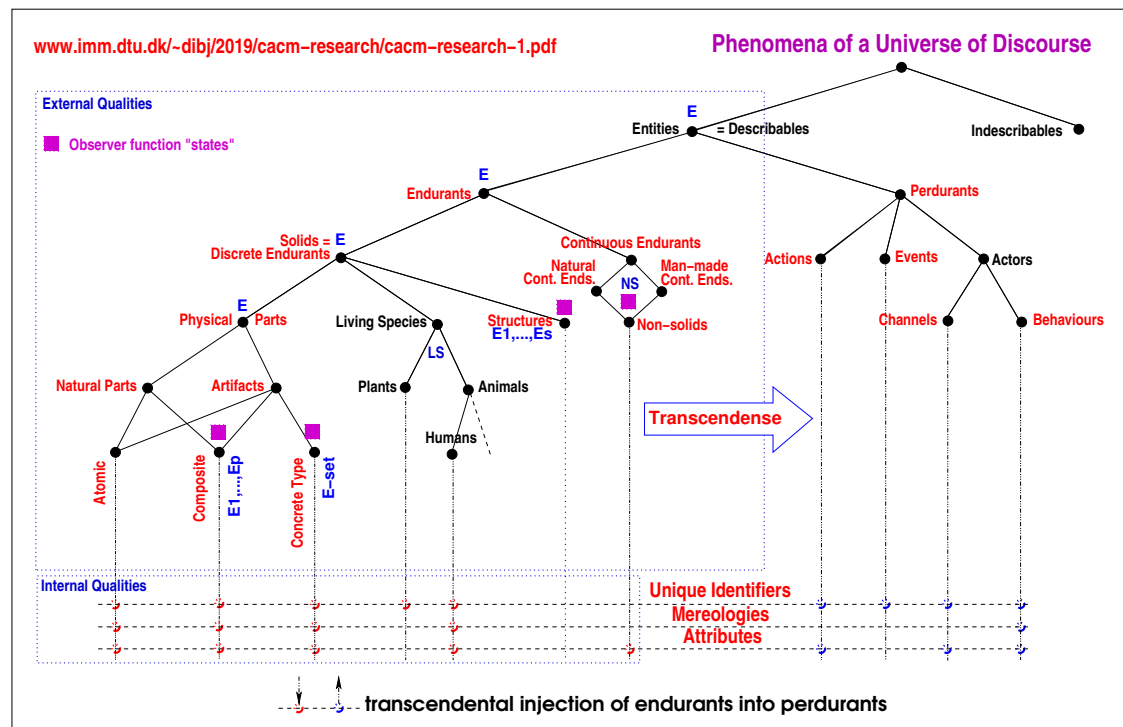


Dines Bjørner

Domain Science & Engineering

A Foundation for Software Development

November 24, 2019: 13:16



A Compendium of Seven Papers and Seven Case Studies

Fredsvej 11, DK-2840 Holte, Denmark

Technical University of Denmark

E-mail: bjorner@gmail.com. URL: www.imm.dtu.dk/~dibj

This monograph was put together in April 2019
Chapter 8 was added September 4
Chapter 7 was added November 22
This version was released November 24, 2019: 13:16

Kari; Charlotte and Nikolaj; Camilla, Marianne, Katrine, Caroline and Jakob
the full meaning of life

Preface

The Triptych Dogma

In order to *specify* **software**,
we must understand its requirements.

In order to *prescribe* **requirements**,
we must understand the **domain**,
so we must **study, analyse** and **describe** it.

General

The thesis of this collection of papers is twofold: (i) that domain engineering is a viable, yes, we would claim, necessary initial phase of software development; and (ii) that domain science & engineering is a worthwhile topic of research. I mean this rather seriously: How can one think of implementing **software**, preferably satisfying some **requirements**, without demonstrating that one understands the **domain**? So in this collection of papers I shall explain what domain engineering is, some of the science that goes with it, and how one can 'derive' requirements prescriptions (for computing systems) from domain descriptions. But there is an altogether different reason, also, for presenting these papers: Software houses may not take up the challenge to develop software that satisfies customers expectations, that is, reflects the domain such as these customers know it, and software that is correct with respect to requirements, with proofs of correctness often having to refer to the domain. But computing scientists are shown, in these papers, that domain science and engineering is a field full of interesting problems to be researched. We consider domain descriptions, requirements prescriptions and software design specifications to be mathematical quantities.

A Brief Guide

I have collected seven documents in this compendium:

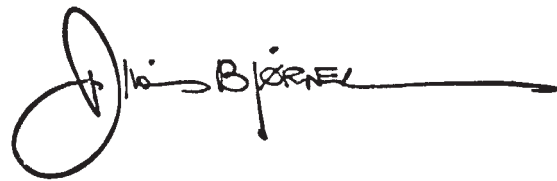
- Chapter 1: [80, 70, Domains Analysis & Description] Pages 3–76
- Chapter 2: [76, 45, Domain Facets: Analysis & Description] Pages 77–106
- Chapter 3: [64, 60, Formal Models of Processes and Prompts – A Sketch] Pages 107–130
- Chapter 4: [78, 41, To Every Manifest Domain Mereology a CSP Expression] Pages 131–151
- Chapter 5: [66, 35, From Domain Descriptions to Requirements Prescriptions] Pages 155–201
- Chapter 6: [65, 52, Demos, Simulators, Monitors and Controllers – A Divertimento] Pages 205–214
- Chapter 7: [82, Sorts, Types, Intents] Pages 217–233
- Chapter 8: [73, Domain Analysis & Description: A Philosophy Basis] Pages 237–253

We urge the reader to study the **Contents** listing and from there to learn that there is a **Bibliography** common to all seven chapters, seven example appendices, **An RSL Primer**, and a set of indexes into definitions, concepts, examples, analysis and description prompts, and **RSL Symbols**.

I have also collected 7 experimental case studies in this compendium:

- Appendix A. **Credit Cards** Pages 279–287
- Appendix B. **Mereorological Information** Pages 289–301
- Appendix C. **Pipelines** Pages 303–320
- Appendix D. **Documents** Pages 321–344
- Appendix E. **Urban Planning** Pages 345–401
- Appendix F. **Swarms of Drones** Pages 403–438
- Appendix G. **Cointainer Terminals** Pages 439–481

Some are still in a stage of “development”.

A handwritten signature in black ink, reading "Dines Bjørner". The signature is stylized, with a large loop for the 'D' and a long horizontal line extending from the end of the name.

*Dines Bjørner. November 24, 2019: 13:16
Fredsevej 11, DK-2840 Holte, Denmark*

Contents

Preface	v
The Triptych Dogma	v
General	v
A Brief Guide	v

Part I The Domain Analysis & Description Method

1 Domain Analysis & Description	3
1.1 Introduction	3
1.1.1 Foreword	3
1.1.2 An Engineering and a Science Viewpoint	4
1.1.3 Some Issues: Metaphysics, Epistemology, Mereology and Ontology	4
1.1.4 The Precursor	6
1.1.5 What is this Chapter About?	6
1.1.6 Structure of this Chapter	7
1.2 Entities: Endurants and Perdurants	7
1.2.1 A Generic Domain Ontology – A Synopsis	7
1.2.2 Universes of Discourse	9
1.2.3 Entities	10
1.2.4 Endurants and Perdurants	10
1.3 Endurants: Analysis of External Qualities	11
1.3.1 Discrete and Continuous Endurants	11
1.3.2 Discrete Endurants	12
1.3.3 Physical Parts	13
1.3.4 Living Species	15
1.3.5 Components	17
1.3.6 Continuous Endurants \equiv Materials	17
1.3.7 Artefacts	17
1.3.8 States	18
1.4 Endurants: The Description Calculus	18
1.4.1 Parts: Natural or Man-made	18
1.4.2 Concrete Part Types	21
1.4.3 On Endurant Sorts	22
1.4.4 Components	22
1.4.5 Materials	23
1.5 Endurants: Analysis & Description of Internal Qualities	24
1.5.1 Unique Identifiers	24
1.5.2 Mereology	26
1.5.3 Attributes	28

1.5.4	Some Axioms and Proof Obligations	35
1.5.5	Discussion of Endurants	36
1.6	A Transcendental Deduction	36
1.6.1	An Explanation	36
1.6.2	Classical Transcendental Deductions	37
1.6.3	Some Special Notation	38
1.7	Space and Time	38
1.7.1	Space	38
1.7.2	Time	39
1.7.3	Whither Attributes?	41
1.7.4	Whither Entities?	41
1.8	Perdurants	41
1.8.1	States, Actors, Actions, Events and Behaviours: A Preview	41
1.8.2	Channels and Communication	44
1.8.3	Perdurant Signatures	47
1.8.4	Discrete Behaviour Definitions	51
1.8.5	Running Systems	55
1.8.6	Concurrency: Communication and Synchronisation	56
1.8.7	Summary and Discussion of Perdurants	57
1.9	The Example Concluded	57
1.9.1	Unique Identifier Concepts	57
1.9.2	Further Transport System Attributes	58
1.9.3	Behaviours	60
1.10	Closing	63
1.10.1	What Have We Achieved?	64
1.10.2	The Four Languages of Domain Analysis & Description	65
1.10.3	Relation to Other Formal Specification Languages	68
1.10.4	Two Frequently Asked Questions	68
1.10.5	On How to Pursue Domain Science & Engineering	69
1.10.6	Domain Science & Engineering	69
1.10.7	Comparison to Related Work	69
1.10.8	Tony Hoare's Summary on 'Domain Modelling'	75
2	Domain Facets: Analysis & Description	77
2.1	Introduction	77
2.1.1	Facets of Domains	77
2.1.2	Relation to Previous Work	78
2.1.3	Structure of Chapter	78
2.2	Intrinsics	78
2.2.1	Conceptual Analysis	78
2.2.2	Requirements	81
2.2.3	On Modeling Intrinsics	81
2.3	Support Technologies	81
2.3.1	Conceptual Analysis	81
2.3.2	Requirements	85
2.3.3	On Modeling Support Technologies	85
2.4	Rules & Regulations	85
2.4.1	Conceptual Analysis	85
2.4.2	Requirements	87
2.4.3	On Modeling Rules and Regulations	87
2.5	Scripts	88
2.5.1	Conceptual Analysis	88
2.5.2	Requirements	89
2.5.3	On Modeling Scripts	90
2.6	License Languages	90
2.6.1	Conceptual Analysis	90

2.6.2	The Pragmatics	91
2.6.3	Schematic Rendition of License Language Constructs	94
2.6.4	Requirements	97
2.6.5	On Modeling License Languages	97
2.7	Management & Organisation	98
2.7.1	Conceptual Analysis	98
2.7.2	Requirements	102
2.7.3	On Modeling Management and Organisation	102
2.8	Human Behaviour	102
2.8.1	Conceptual Analysis	102
2.8.2	Requirements	104
2.8.3	On Modeling Human Behaviour	104
2.9	Conclusion	104
2.9.1	Completion	104
2.9.2	Integrating Formal Descriptions	105
2.9.3	The Impossibility of Describing Any Domain Completely	105
2.9.4	Rôles for Domain Descriptions	105
2.9.5	Grand Challenges of Informatics	106
2.10	Bibliographical Notes	106
3	Towards Formal Models of Processes and Prompts – a Sketch	107
3.1	Introduction	107
3.1.1	Related Work	108
3.1.2	Structure of Chapter	108
3.2	Domain Analysis and Description	108
3.3	Syntax and Semantics	109
3.3.1	Form and Content	109
3.3.2	Syntactic and Semantic Types	109
3.3.3	Names and Denotations	110
3.4	A Model of the Domain Analysis & Description Process	110
3.4.1	Introduction	110
3.4.2	A Model of the Analysis & Description Process	111
3.4.3	Discussion of The Process Model	113
3.5	A Domain Analyser's & Describer's Domain Image	114
3.6	Domain Types	115
3.6.1	Syntactic Types: Parts, Materials and Components	116
3.6.2	Semantic Types: Parts, Materials and Components	119
3.7	From Syntax to Semantics and Back Again !	120
3.7.1	The Analysis & Description Prompt Arguments	120
3.7.2	Some Auxiliary Maps: Syntax to Semantics and Semantics to Syntax	120
3.7.3	M: A Meaning of Type Names	121
3.7.4	The ι Description Function	123
3.8	A Formal Description of a Meaning of Prompts	124
3.8.1	On Function Overloading	124
3.8.2	The Analysis Prompts	124
3.8.3	The Description Prompts	126
3.8.4	Discussion of The Prompt Model	129
3.9	Conclusion	130
3.9.1	What Has Been Achieved ?	130
3.9.2	Are the Models Valid ?	130
3.9.3	Future Work	130

4	To Every Manifest Domain Mereology a CSP Expression	131
4.1	Introduction	131
4.1.1	Mereology	131
4.1.2	From Domains via Requirements to Software	132
4.1.3	Domains: Science and Engineering	132
4.1.4	Contributions of This Chapter	133
4.1.5	Structure of Chapter	133
4.2	Our Concept of Mereology	133
4.2.1	Informal Characterisation	133
4.2.2	Six Examples	134
4.3	An Abstract, Syntactic Model of Mereologies	139
4.3.1	Parts and Subparts	139
4.3.2	No “Infinitely” Embedded Parts	139
4.3.3	Unique Identifications	140
4.3.4	Attributes	142
4.3.5	Mereology	143
4.3.6	The Model	143
4.4	Some Part Relations	144
4.4.1	‘Immediately Within’	144
4.4.2	‘Transitive Within’	144
4.4.3	‘Adjacency’	144
4.4.4	Transitive ‘Adjacency’	145
4.5	An Axiom System	145
4.5.1	Parts and Attributes	145
4.5.2	The Axioms	145
4.6	Satisfaction	146
4.6.1	Some Definitions	146
4.6.2	A Proof Sketch	147
4.7	A Semantic CSP Model of Mereology	147
4.7.1	Parts \simeq Processes	147
4.7.2	Channels	147
4.7.3	Compilation	148
4.7.4	Discussion	149
4.8	Concluding Remarks	149
4.8.1	Relation to Other Work	149
4.8.2	What Has Been Achieved?	151

Part II A Requirements Engineering Method

5	From Domain Descriptions to Requirements Prescriptions	155
5.1	Introduction	155
5.1.1	The Triptych Dogma of Software Development	155
5.1.2	Software As Mathematical Objects	155
5.1.3	The Contribution of Chapter	155
5.1.4	Some Comments	156
5.1.5	Structure of Chapter	156
5.2	An Example Domain: Transport	156
5.2.1	Endurants	157
5.2.2	Domain, Net, Fleet and Monitor	157
5.2.3	Perdurants	163
5.2.4	Domain Facets	167
5.3	Requirements	167
5.3.1	The Three Phases of Requirements Engineering	167
5.3.2	Order of Presentation of Requirements Prescriptions	168
5.3.3	Design Requirements and Design Assumptions	168

5.3.4	Derived Requirements	169
5.4	Domain Requirements	169
5.4.1	Domain Projection	169
5.4.2	Domain Instantiation	173
5.4.3	Domain Determination	178
5.4.4	Domain Extension	180
5.4.5	Requirements Fitting	188
5.4.6	Discussion	188
5.5	Interface and Derived Requirements	189
5.5.1	Interface Requirements	189
5.5.2	Derived Requirements	194
5.5.3	Discussion	197
5.6	Machine Requirements	197
5.7	Conclusion	198
5.7.1	What has been Achieved?	198
5.7.2	Present Shortcomings and Research Challenges	198
5.7.3	Comparison to “Classical” Requirements Engineering:	199
5.8	Bibliographical Notes	200

Part III Some Implications for Software

6	Demos, Simulators, Monitors and Controllers	
	A Divertimento of Ideas and Suggestions.	205
6.1	Introduction	205
6.2	Domain Descriptions	206
6.3	Interpretations	207
6.3.1	What Is a Domain-based Demo?	207
6.3.2	Simulations	208
6.3.3	Monitoring & Control	210
6.3.4	Machine Development	211
6.3.5	Verifiable Software Development	212
6.4	Conclusion	213
6.4.1	Discussion	214

Part IV A Note

7	Sorts, Types, Intents	217
7.1	Introduction	217
7.1.1	Entities, Endurants and Perdurants	217
7.1.2	Discrete and Continuous Endurants	218
7.1.3	A Domain Ontology	218
7.2	Sorts	218
7.2.1	Physical Parts, Living Species and Structures	218
7.2.2	Natural Parts and Artefacts	220
7.2.3	Various Forms of Physical Parts	220
7.2.4	Analysis and Description Prompts	220
7.2.5	An Example: Road Transport	221
7.3	Types	222
7.3.1	Space and Time	222
7.3.2	Internal Qualities	223
7.3.3	Physics Attributes	225
7.3.4	Artefactual Attributes	227
7.4	Intents	228
7.4.1	Expressing Intents	228

7.4.2	Intent Modelling	229
7.4.3	Intentional Pull	229
7.5	Actions, Events, Behaviours	231
7.5.1	Actions	231
7.5.2	Events	231
7.5.3	Behaviours	231
7.5.4	Summary	232
7.6	Conclusion	232
7.6.1	Sort versus Types	232
7.6.2	An Earlier Review of Types	233
7.6.3	Domain Oriented Programming Languages	233
7.6.4	Research Topics	233
7.6.5	Acknowledgements	233

Part V Issues of Philosophy

8	Domain Analysis & Description	
	A Philosophy Basis	237
8.1	Introduction	237
8.1.1	Domain Science & Engineering	238
8.1.2	Some Issues of Philosophy	239
8.1.3	Transcendence	240
8.2	Overview of The Sørlander Philosophy	240
8.2.1	Logical Connectives	240
8.2.2	A Philosophy Basis for Physics and Biology	240
8.3	Phenomena and Entities	243
8.3.1	Endurants	243
8.3.2	Perdurants	243
8.4	Endurants: External Qualities	244
8.4.1	Discrete and Continuous Endurants	244
8.4.2	Solids	245
8.4.3	Non-solids	246
8.4.4	The Analysis Prompts	246
8.5	Endurants: Internal Qualities	246
8.5.1	Unique Identifiers	246
8.5.2	Mereology	247
8.5.3	Attributes	247
8.6	Endurants: Description Prompts	248
8.7	From Parts to Behaviours	249
8.8	Perdurants	250
8.8.1	Behaviour Signatures	250
8.8.2	Behaviour Definition Bodies: B_p	250
8.8.3	From Part Descriptions to Behaviour Definitions	251
8.8.4	Channel Declarations	252
8.8.5	Concrete System	252
8.9	Conclusion	252
8.9.1	What Have We Achieved?	252
8.9.2	Open Problems	253
8.9.3	Acknowledgment	253
8.9.4	Acknowledgment	253

Part VI Conclusion

9	Summing Up	257
9.1	What Have We Achieved?	257
9.1.1	Chapter-by-Chapter Achievement Enumeration	257
9.1.2	Fulfillment of Aim	258
9.2	Acknowledgements	259

Part VII A Common Bibliography

10	Bibliography	263
-----------	---------------------------	-----

Part VIII Experimental Case Studies

A	Credit Card Systems	279
A.1	Introduction	279
A.2	Endurants	279
A.2.1	Credit Card Systems	279
A.2.2	Credit Cards	281
A.2.3	Banks	281
A.2.4	Shops	282
A.3	Perdurants	282
A.3.1	Behaviours	282
A.3.2	Channels	283
A.3.3	Behaviour Interactions	283
A.3.4	Credit Card	284
A.3.5	Banks	285
A.3.6	Shops	287
A.4	Discussion	287
B	Weather Information Systems	289
B.1	On Weather Information Systems	289
B.1.1	On a Base Terminology	289
B.1.2	Some Illustrations	290
B.2	Major Parts of a Weather Information System	291
B.3	Endurants	291
B.3.1	Parts and Materials	291
B.3.2	Unique Identifiers	293
B.3.3	Mereologies	293
B.3.4	Attributes	294
B.4	Perdurants	296
B.4.1	A WIS Context	296
B.4.2	Channels	297
B.4.3	WIS Behaviours	297
B.4.4	Clock	298
B.4.5	Weather Station	298
B.4.6	Weather Data Interpreter	298
B.4.7	Weather Forecast Consumer	300
B.5	Conclusion	301
B.5.1	Reference to Similar Work	301
B.5.2	What Have We Achieved?	301
B.5.3	What Needs to be Done Next?	301
B.5.4	Acknowledgements	301

C	Pipeline Systems	303
C.1	Photos of Pipeline Units and Diagrams of Pipeline Systems	303
C.2	Non-Temporal Aspects of Pipelines	304
C.2.1	Nets of Pipes, Valves, Pumps, Forks and Joins	304
C.2.2	Unit Identifiers and Unit Type Predicates	306
C.2.3	Unit Connections	306
C.2.4	Net Observers and Unit Connections	308
C.2.5	Well-formed Nets, Actual Connections	310
C.2.6	Well-formed Nets, No Circular Nets	310
C.2.7	Well-formed Nets, Special Pairs, wfN_SP	311
C.2.8	Special Routes, I	311
C.2.9	Special Routes, II	312
C.3	State Attributes of Pipeline Units	312
C.3.1	Flow Laws	313
C.3.2	Possibly Desirable Properties	314
C.4	Pipeline Actions	314
C.4.1	Simple Pump and Valve Actions	314
C.4.2	Events	315
C.4.3	Well-formed Operational Nets	316
C.4.4	Orderly Action Sequences	316
C.4.5	Emergency Actions	317
C.5	Connectors	317
C.6	On Temporal Aspects of Pipelines	319
C.7	A CSP Model of Pipelines	319
C.8	Conclusion	320
D	A Document System	321
D.1	Introduction	321
D.2	A System for Managing, Archiving and Handling Documents	321
D.3	Principal Endurants	322
D.4	Unique Identifiers	322
D.5	Documents: A First View	323
D.5.1	Document Identifiers	323
D.5.2	Document Descriptors	323
D.5.3	Document Annotations	323
D.5.4	Document Contents: Text/Graphics	324
D.5.5	Document Histories	324
D.5.6	A Summary of Document Attributes	324
D.6	Behaviours: An Informal, First View	325
D.7	Channels, A First View	326
D.8	An Informal Graphical System Rendition	326
D.9	Behaviour Signatures	327
D.10	Time	327
D.10.1	Time and Time Intervals: Types and Functions	327
D.10.2	A Time Behaviour and a Time Channel	328
D.10.3	An Informal RSL Construct	328
D.11	Behaviour "States"	328
D.12	Inter-Behaviour Messages	329
D.12.1	Management Messages with Respect to the Archive	329
D.12.2	Management Messages with Respect to Handlers	330
D.12.3	Document Access Rights	330
D.12.4	Archive Messages with Respect to Management	330
D.12.5	Archive Message with Respect to Documents	331
D.12.6	Handler Messages with Respect to Documents	331
D.12.7	Handler Messages with Respect to Management	331
D.12.8	A Summary of Behaviour Interactions	331

D.13	A General Discussion of Handler and Document Interactions	332
D.14	Channels: A Final View	332
D.15	An Informal Summary of Behaviours	332
D.15.1	The Create Behaviour: Left Fig. D.3 on Page 333	332
D.15.2	The Edit Behaviour: Right Fig. D.3 on Page 333	333
D.15.3	The Read Behaviour: Left Fig. D.4 on Page 334	333
D.15.4	The Copy Behaviour: Right Fig. D.4 on Page 334	333
D.15.5	The Grant Behaviour: Left Fig. D.5 on Page 334	334
D.15.6	The Shred Behaviour: Right Fig. D.5 on Page 334	335
D.16	The Behaviour Actions	335
D.16.1	Management Behaviour	335
D.16.2	Archive Behaviour	338
D.16.3	Handler Behaviours	340
D.16.4	Document Behaviours	342
D.16.5	Conclusion	343
D.17	Documents in Public Gornment	343
D.18	Documents in Urban Planning	344
E	Urban Planning	345
E.1	Introduction	345
E.1.1	On Urban Planning	346
E.1.2	On the Form of This Research Note	348
E.1.3	On the Structure of this Research Note	348
E.2	An Urban Planning System	349
E.2.1	A First Iteration Overview	349
E.2.2	A Visual Rendition of Urban Planning Development	350
E.3	METHOD	351
E.4	Prelude	351
E.4.1	A Triptych of Software Development	351
E.4.2	On Formality	352
E.4.3	On Describing Domains	352
E.4.4	Reiterating Domain Modeling	353
E.4.5	Partial, Precise, and Approximate Descriptions	353
E.4.6	On Formal Notations	353
E.5	Review & Refinement of the Method	354
E.5.1	Review of Manifest Domains: Analysis & Description	354
E.5.2	Refinement of the Method	355
E.6	ENDURANTS	355
E.7	Structures and Parts	356
E.7.1	The Urban Space, Clock, Analysis & Planning Complex	356
E.7.2	The Analyser Structure and Named Analysers	356
E.7.3	The Planner Structure	357
E.7.4	Atomic Parts	357
E.7.5	Preview of Structures and Parts	358
E.7.6	Planner Names	358
E.7.7	Individual and Sets of Atomic Parts	359
E.8	Unique Identifiers	359
E.8.1	Urban Space Unique Identifier	360
E.8.2	Analyser Unique Identifiers	360
E.8.3	Master Planner Server Unique Identifier	360
E.8.4	Master Planner Unique Identifier	360
E.8.5	Derived Planner Server Unique Identifier	361
E.8.6	Derived Planner Unique Identifier	361
E.8.7	Derived Plan Index Generator Identifier	361
E.8.8	Plan Repository	361
E.8.9	Uniqueness of Identifiers	362

	E.8.10	Indices and Index Sets	362
	E.8.11	Retrieval of Parts from their Identifiers	363
	E.8.12	A Bijection: Derived Planner Names and Derived Planner Identifiers	363
E.9	Mereologies		364
	E.9.1	Clock Mereology	364
	E.9.2	Urban Space Mereology	364
	E.9.3	Analyser Mereology	365
	E.9.4	Analysis Depository Mereology	365
	E.9.5	Master Planner Server Mereology	365
	E.9.6	Master Planner Mereology	365
	E.9.7	Derived Planner Server Mereology	366
	E.9.8	Derived Planner Mereology	366
	E.9.9	Derived Planner Index Generator Mereology	367
	E.9.10	Plan Repository Mereology	367
E.10	Attributes		367
	E.10.1	Clock Attribute	367
	E.10.2	Urban Space Attributes	368
	E.10.3	Scripts	373
	E.10.4	Urban Analysis Attributes	373
	E.10.5	Analysis Depository Attributes	373
	E.10.6	Master Planner Server Attributes	374
	E.10.7	Master Planner Attributes	374
	E.10.8	Derived Planner Server Attributes	374
	E.10.9	Derived Planner Attributes	375
	E.10.10	Derived Planner Index Generator Attributes	375
	E.10.11	Plan Repository Attributes	375
	E.10.12	A System Property of Derived Planner Identifiers	376
E.11	PERDURANTS		376
E.12	The Structure COMPILERS		377
	E.12.1	A UNIVERSE OF DISCOURSE COMPILER	377
	E.12.2	The ANALYSER STRUCTURE COMPILER	377
	E.12.3	The PLANNER STRUCTURE COMPILER	378
E.13	Channel Analysis and Channel Declarations		379
	E.13.1	The <code>clk_ch</code> Channel	379
	E.13.2	The <code>tus_a_ch</code> Channel	380
	E.13.3	The <code>tus_mps_ch</code> Channel	380
	E.13.4	The <code>a_ad_ch</code> Channel	380
	E.13.5	The <code>ad_s_ch</code> Channel	381
	E.13.6	The <code>mps_mp_ch</code> Channel	381
	E.13.7	The <code>p_pr_ch</code> Channel	381
	E.13.8	The <code>p_dpxg_ch</code> Channel	382
	E.13.9	The <code>pr_s_ch</code> Channel	382
	E.13.10	The <code>dps_dp_ch</code> Channel	382
E.14	The Atomic Part TRANSLATORS		383
	E.14.1	The CLOCK TRANSLATOR	383
	E.14.2	The URBAN SPACE TRANSLATOR	383
	E.14.3	The ANALYSER_{ann_i}, $i:[1:n]$ TRANSLATOR	385
	E.14.4	The ANALYSIS DEPOSITORY TRANSLATOR	386
	E.14.5	The DERIVED PLANNER INDEX GENERATOR TRANSLATOR	387
	E.14.6	The PLAN REPOSITORY TRANSLATOR	388
	E.14.7	The MASTER SERVER TRANSLATOR	389
	E.14.8	The MASTER PLANNER TRANSLATOR	391
	E.14.9	The DERIVED SERVER_{nm_i}, $i:[1:p]$ TRANSLATOR	393
	E.14.10	The DERIVED PLANNER_{nm_i}, $i:[1:p]$ TRANSLATOR	394
E.15	Initialisation of The Urban Space Analysis & Planning System		396

E.15.1	Summary of Parts and Part Names	397
E.15.2	Summary of of Unique Identifiers	397
E.15.3	Summary of Channels	397
E.15.4	The Initial System	398
E.15.5	The Derived Planner System	398
E.16	Further Work	398
E.16.1	Reasoning About Deadlock, Starvation, Live-lock and Liveness	398
E.16.2	Document Handling	399
E.16.3	Validation and Verification (V&V)	399
E.16.4	Urban Planning Project Management	400
E.17	Conclusion	401
E.17.1	What Were Our Expectations ?	401
E.17.2	What Have We Achieved ?	401
E.17.3	What Next ?	401
E.17.4	Acknowledgement	401
F	Swarms of Drones	403
F.1	An Informal Introduction	403
F.1.1	Describable Entities	403
F.1.2	The Contribution of [70]	404
F.1.3	The Contribution of This Report	405
F.2	Entities, Endurants	405
F.2.1	Parts, Atomic and Composite, Sorts, Abstract and Concrete Types	405
F.2.2	Unique Identifiers	407
F.2.3	Mereologies	409
F.2.4	Attributes	413
F.3	Operations on Universe of Discourse States	420
F.3.1	The Notion of a State	420
F.3.2	Constants	420
F.3.3	Operations	421
F.4	Perdurants	422
F.4.1	System Compilation	422
F.4.2	An Early Narrative on Behaviours	425
F.4.3	Channels	426
F.4.4	The Atomic Behaviours	430
F.5	Conclusion	438
G	Container Terminals	439
G.1	Introduction	439
G.2	Basic Concepts of Container Terminal Ports	439
G.2.1	Pictorial Renditions: Diagrams	439
G.2.2	Terminology - a Caveat	439
G.2.3	Assumptions	440
G.3	Endurants	441
G.3.1	The Container Line Industry	441
G.3.2	Parts	441
G.3.3	Terminal Ports	442
G.3.4	Unique Identifications	442
G.3.5	Containers	443
G.3.6	Trucks, Cranes, Bay Stacks and Vessels	447
G.3.7	Stacks	447
G.3.8	Terminal Port Command Centers	448
G.3.9	States, Global Values and Constraints	449
G.3.10	Mereology	451
G.3.11	Attributes	454
G.4	Perdurants	457

G.4.1	A Diagram	457
G.4.2	Very Brief Overview	458
G.4.3	Short Behaviour Narratives	458
G.4.4	Actions	460
G.4.5	Channels	464
G.4.6	Behaviour Signatures	465
G.4.7	A Running System	468
G.4.8	Behaviour Definitions	468
G.5	Conclusion	481
G.5.1	Variations of Container Terminal Descriptions	481
G.5.2	A Proper Container Terminal Analysis & Description Project	481

Part IX RSL

H	An RSL Primer	485
H.1	Types	485
H.1.1	Type Expressions	485
H.1.2	Type Definitions	486
H.2	The RSL Predicate Calculus	488
H.2.1	Propositional Expressions	488
H.2.2	Simple Predicate Expressions	488
H.2.3	Quantified Expressions	488
H.3	Concrete RSL Types: Values and Operations	488
H.3.1	Arithmetic	488
H.3.2	Set Expressions	489
H.3.3	Cartesian Expressions	489
H.3.4	List Expressions	489
H.3.5	Map Expressions	490
H.3.6	Set Operations	491
H.3.7	Cartesian Operations	492
H.3.8	List Operations	492
H.3.9	Map Operations	494
H.4	λ-Calculus + Functions	496
H.4.1	The λ-Calculus Syntax	496
H.4.2	Free and Bound Variables	496
H.4.3	Substitution	496
H.4.4	α-Renaming and β-Reduction	497
H.4.5	Function Signatures	497
H.4.6	Function Definitions	497
H.5	Other Applicative Expressions	498
H.5.1	Simple let Expressions	498
H.5.2	Recursive let Expressions	498
H.5.3	Predicative let Expressions	498
H.5.4	Pattern and “Wild Card” let Expressions	498
H.5.5	Conditionals	499
H.5.6	Operator/Operand Expressions	499
H.6	Imperative Constructs	500
H.6.1	Statements and State Changes	500
H.6.2	Variables and Assignment	500
H.6.3	Statement Sequences and skip	500
H.6.4	Imperative Conditionals	500
H.6.5	Iterative Conditionals	501
H.6.6	Iterative Sequencing	501
H.7	Process Constructs	501
H.7.1	Process Channels	501

H.7.2	Process Composition	501
H.7.3	Input/Output Events	501
H.7.4	Process Definitions	502
H.8	Simple RSL Specifications	502

Part X **Indexes**

I	Indexes	505
I.1	Definitions	505
I.2	Concepts	510
I.3	Examples	516
I.4	Analysis Prompts	517
I.5	Description Prompts	517
I.6	Attribute Categories	517
I.7	RSL Symbols	517

The Domain Analysis & Description Method

Domain Analysis & Description

We¹ present a *method* for **analysing and describing manifest (discrete dynamics) domains**.

1.1 Introduction

By a **domain** we shall understand a **rationally describable** segment of a **discrete dynamics** segment of a **human assisted** reality, i.e., of the world, its **physical parts: natural** [“God-given”] and **artifactual** [“man-made”], and **living species: plants** and **animals** including, notably, **humans**. These are **endurants** (“still”), as well as **perdurants** (“alive”). Emphasis is placed on **“human-assistedness”**, that is, that there is *at least one (man-made) artifact* and, therefore, that **humans** are a primary cause for change of **endurant states** as well as **perdurant behaviours**.

Please note the ‘delimiter’: *discrete dynamics*. Control theory, the study of the control of continuously operating dynamical systems in engineered processes and machines, is one thing; domain engineering is “a different thing”. Where control theory builds upon classical physics, and uses classical mathematics, partial differential equations, etc., to model phenomena of physics and therefrom engineered ‘machines’; domain science & engineering, in some contrast, builds upon mathematical logic, and, to some extent, modern algebra, to model phenomena of mostly artefactual systems.

Domain science & engineering marks a new area of *computing science*. Just as we are *formalising the syntax and semantics of programming languages*, so we are *formalising the syntax and semantics of human-assisted domains*. Just as *physicists* are studying the *natural physical world*, endowing it with *mathematical models*, so we, *computing scientists*, are studying these *domains*, endowing them with *mathematical models*. A difference between the endeavours of physicists and ours lies in the tools: the physics models are based on *classical mathematics, differential equations and integrals*, etc.; our models are based on *mathematical logic, set theory, and algebra*.

Where physicists thus classically use a variety of *differential and integral calculi* to model the physical world, we shall be using the *analysis & description calculi* presented in this chapter to model primarily artefactual domains.

1.1.1 Foreword

Dear reader! You are about to embark on a journey. The chapter in front of you is long! But it is not the number of pages, 76, or duration of your studying the chapter that I am referring to. It is the mind that should be prepared for a journey. It is a journey into a new realm. A realm where we confront the computer

¹ Chapter 1 is primarily based on [80]. That paper was based on [70]. Section 1.5.2’s Part Relations is changed wrt. [80, Sect. 4.2.1]. Section 9.7 of [80] has here been replaced by Sect. 1.10.7 which is taken from [70]. Remaining editing changes are of syntactics art.

& computing scientists with a new universe: a universe in which we build a bridge between the *informal* world, that we live in, the context for eventual, *formal* software, and that *formal* software.

The bridge involves a novel construction, new in computing science: a **transcendental deduction**. We are going to present you, we immodestly claim, with a new way of looking at the “origins” of software, the domain in which it is to serve. We shall show a method, a set of principles and techniques and a set of languages, some formal, some “almost” formal, and the informal language of usual computing science papers for a systematic to rigorous way of *analysing & describing domains*. We immodestly claim that such a method has not existed before.

1.1.2 An Engineering and a Science Viewpoint

A Triptych of Software Development

It seems reasonable to expect that before **software** can be designed we must have a reasonable grasp of its **requirements**; before **requirements** can be expressed we must have a reasonable grasp of the underlying **domain**. It therefore seems reasonable to structure software development into: **domain engineering**, in which “the underlying” domain is *analysed and described*²; **requirements engineering**, in which requirements are *analysed and prescribed* – such as we suggest it [35, 66] – based on a domain description³; and **software design**, in which the software is *rigorously “derived”* from a requirements prescription⁴. Our interest, in this chapter, lies solely in domain analysis & description.

Domain Science & Engineering:

The present chapter outlines a *methodology* for an aspect of software development. Domain analysis & description can be pursued in isolation, for example, without any consideration of any other aspect of software development. As such domain analysis & description represents an aspect of **domain science & engineering**. Other aspects are covered in: Chap. 2 [76, *Domain Facets*], Chap. 3 [64, *An Analysis & Description Process Model*], Chap. 4 [78, *From Mereologies to Lambda-Expressions*], Chap. 5 [66, *Requirements Engineering*], and in Chap. 6 [65, 52, *Domains: Their Simulation, Monitoring and Control*]. This work is over-viewed in [77, *Domain Science & Engineering – A Review of 10 Years Work*]. They are all facets of an emerging **domain science & engineering**. *We consider the present chapter to outline the basis for this science and engineering.*

1.1.3 Some Issues: Metaphysics, Epistemology, Mereology and Ontology

But there is an even more fundamental issue “at play” here. It is that of philosophy. Let us briefly review some aspects of philosophy.

Metaphysics is a branch of *philosophy* that explores fundamental questions, including the nature of concepts like *being*, *existence*, and *reality* ■⁵

Traditional metaphysics seeks to answer, in a “suitably abstract and fully general manner”, the questions: *What is there?* and *And what is it like?*⁶. Topics of metaphysical investigation include existence, objects and their properties, space and time, cause and effect, and possibility.

Epistemology is the branch of philosophy concerned with the theory of knowledge⁷ ■

² including the statement and possible proofs of properties of that which is denoted by the domain description

³ including the statement and possible proofs of properties of that which is denoted by the requirements prescription with respect also to the domain description

⁴ including the statement and possible proofs of properties of that which is specified by the software design with respect to both the requirements prescription and the domain description

⁵ ■ is used to signal the end of a characterisation, a definition, or an example.

⁶ <https://en.wikipedia.org/wiki/Metaphysics>

⁷ <https://en.wikipedia.org/wiki/Epistemology>

Epistemology studies the nature of knowledge, justification, and the rationality of belief. Much of the debate in epistemology centers on four areas: (1) the philosophical analysis of the nature of knowledge and how it relates to such concepts as truth, belief, and justification, (2) various problems of skepticism, (3) the sources and scope of knowledge and justified belief, and (4) the criteria for knowledge and justification. A central branch of *epistemology* is *ontology*.⁸

Ontology: An *ontology* encompasses a representation, formal naming, and definition of the categories, properties, and relations of the entities that substantiate one, many, or all domains.⁹ An *upper ontology* (also known as a top-level ontology or foundation ontology) is an ontology which consists of very general terms (such as *entity*, *endurant*, *attribute*) that are common across all domains¹⁰ ■

Mereology (from the Greek *μερος* ‘part’) is the theory of part-hood relations: of the relations of part to whole and the relations of part to part within a whole [104]¹¹ ■

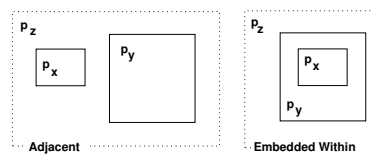


Fig. 1.1. Immediately ‘Adjacent’ and ‘Embedded Within’ Parts

Accordingly two parts, p_x and p_y , (of a same “whole”) are either “adjacent”, or are “embedded within”, one within the other, as loosely indicated in Fig. 1.1. ‘Adjacent’ parts are direct parts of a same third part, p_z , i.e., p_x and p_y are “embedded within” p_z ; or one (p_x) or the other (p_y) or both (p_x and p_y) are parts of a same third part, p'_z “embedded within” p_z ; et cetera; as loosely indicated in Fig. 1.2, or one is “embedded within” the other — etc. as loosely indicated in Fig. 1.2. Parts, whether ‘adjacent’ or ‘embedded

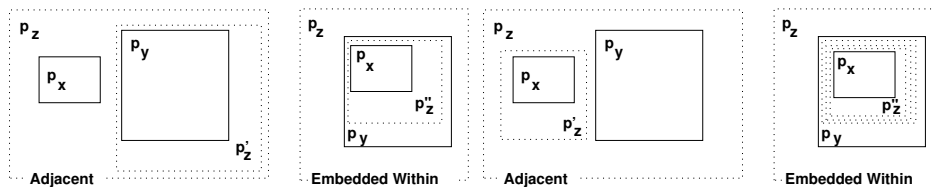


Fig. 1.2. Transitively ‘Adjacent’ and ‘Embedded Within’ Parts

within’, can share properties. For adjacent parts this sharing seems, in the literature, to be diagrammatically expressed by letting the part rectangles “intersect”. Usually properties are not spatial hence ‘intersection’ seems confusing. We refer to Fig. 1.3 on the next page. Instead of depicting parts sharing properties as in Fig. 1.3 on the following page[L]eft, where shaded, dashed rounded-edge rectangles stands for ‘sharing’, we shall (eventually) show parts sharing properties as in Fig. 1.3 on the next page[R]ight where ●—● connections connect those parts.

We refer to [78, *From Mereologies to Lambda-Expressions*].

Mereology is basically the contribution [172, 247] of the Polish philosopher, logician and mathematician Stanisław Leśniewski (1886–1939).

⁸ <https://en.wikipedia.org/wiki/Metaphysics>
⁹ [https://en.wikipedia.org/wiki/On-tology_\(information_science\)](https://en.wikipedia.org/wiki/On-tology_(information_science))
¹⁰ https://en.wikipedia.org/wiki/Upper_ontology
¹¹ <https://plato.stanford.edu/entries/mereology>

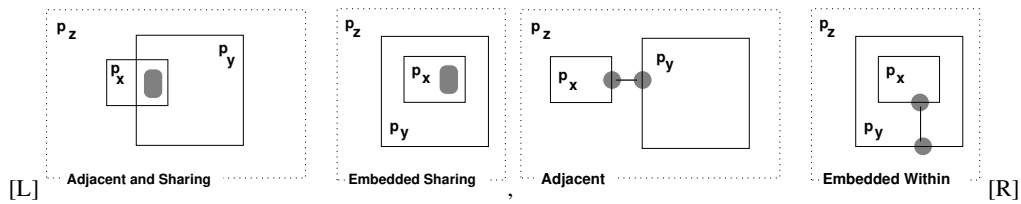


Fig. 1.3. Two models, [L,R], of parts sharing properties

Kai Sørlander's Philosophy:

We shall base some of our modelling decisions of Kai Sørlander's Philosophy [238, 239, 241, 245]. A main contribution of Kai Sørlander is, on the philosophical basis of the *possibility of truth* (in contrast to Kant's *possibility of self-awareness*), to *rationally and transcendentally deduce the absolutely necessary conditions for describing any world*.

These conditions presume a *principle of contradiction* and lead to the *ability to reason using logical connectives* and to *handle asymmetry, symmetry and transitivity*. *Transcendental deductions* then lead to *space and time*, not as *priory assumptions*, as with Kant, but *derived facts of any world*. From this basis Kai Sørlander then, by further *transcendental deductions*, arrive at *kinematics, dynamics and the bases for Newton's Laws*. And so forth.

We build on Sørlander's basis to argue that the domain analysis & description calculi are necessary and sufficient for the analysis & description of domains and that a number of relations between domain entities can be understood transcendentally and as "variants" of laws of physics, biology, etc. !

1.1.4 The Precursor

The present chapter is based on a revision of the published [70] – as published in [80, April 2019]. The revision considerably simplifies and considerably extends the domain analysis & description calculi of [70]. The major revision that prompts this complete rewrite is due to a serious study of Kai Sørlander's Philosophy. As a result we extend [70]'s ontology of *endurants*: describable phenomena to not only cover those of **physical phenomena**, but also those of **living species**, notably **humans**, and, as a result of that, our understanding of discrete endurants is refined into those of **natural parts** and **artifacts**. A new contribution is that of **intentional "pull"** akin to the *gravitational pull* of physics. Both this paper and [70] are the result of extensive "non-toy" example case studies, see the example: *Universes of Discourse* – on Page 9. The last half of these were carried out in the years since [70] was first submitted (i.e., 2014). The present paper omits the extensive introduction¹² and closing of [70, Sects. 1 and 5]. Most notably, however, is a clarified view on the transition from **parts** to **behaviours**, a **transcendental deduction** from *domain space* to *domain time*.

1.1.5 What is this Chapter About ?

We present a *method for analysing &*¹³ *describing domains*.

Definition 1 Domain: By a **domain** we shall understand a **rationally describable** segment of a **discrete dynamics** segment of a **human assisted** reality, i.e., of the world, its **physical parts**, **natural** ["God-given"] and **artefactual** ["man-made"], and **living species**: **plants** and **animals** including, predominantly,

¹² **Note added in proof:** Omitted from the extensive, five page, literature survey of [70] was [110, Section 5.3]. It is an interesting study of the domain of geography.

¹³ By *A&B* we mean one topic, the confluence of topics *A* and *B*.

humans. These are **endurants** (“still”) as well as **perdurants** (“alive”). Emphasis is placed on “**human-assistedness**”, that is, that there is *at least one (man-made) artefact* and that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours** ■

Definition 2 Domain Description: By a **domain description** we shall understand a combination of **narration** and **formalisation** of a domain. A **formal specification** is a collection of *sort*, or *type* definitions, *function* and *behaviour* definitions, together with *axioms* and *proof obligations* constraining the definitions. A **specification narrative** is a natural language text which in terse statements introduces the names of (in this case, the domain), and, in cases, also the definitions, of sorts (types), functions, behaviours and axioms; not anthropomorphically, but by emphasizing their properties ■

Domain descriptions are (to be) void of any reference to future, contemplated software, let alone IT systems, that may support entities of the domain. As such *domain models*¹⁴ can be studied separately, for their own sake, for example as a basis for investigating possible domain theories, or can, subsequently, form the basis for requirements engineering with a view towards development of (‘future’) software, etc. *Our aim is to provide a method for the precise analysis and the formal description of domains.*

1.1.6 Structure of this Chapter

Sections 1.2–1.8 form the core of this chapter. Section 1.2 introduces the first concepts of domain phenomena: *endurants* and *perdurants*. Their characterisation, in the form of “definitions”, cannot be mathematically precise, as is usual in computer science papers. Section 1.3 analyses the so-called *external qualities* of *endurants* into *natural parts*, *structures*, *components*, *materials*, *living species* and *artefacts*. In doing so it covers the *external quality analysis prompts*. Section 1.4 covers the *external quality description prompts*. Section 1.5 analyses the so-called *internal qualities* of *endurants* into *unique identification*, *mereology* and *attributes*. In doing so it covers both the *internal quality analysis prompts* and the *internal quality description prompts*. Sections 1.3–1.5 cover what this chapter has to say about *endurants*. Section 1.6 “bridges” Sects. 1.3–1.5 and Sect. 1.8 by introducing the concept of *transcendental deduction*. These deductions allow us to “transform” *endurants* into *perdurants*: “passive” entities into “active” ones. The essence of Sects. 1.6–1.8 is to “translate” endurant parts into perdurant behaviours. Section 1.8 – although “only” half as long as the three sections on *endurants* – covers the analysis & description method for *perdurants*. We shall model perdurants, notably *behaviours*, in the form of CSP [148]. Hence we introduce the CSP notions of *channels* and channel *input/output*. Section 1.8 then “derives” the types of the behaviour arguments from the internal endurant qualities. Section 1.10 summarises the achievements and discusses open issues. Section 1.10.2 on Page 65 summarises the four languages used in this chapter.

Framed texts either delineate major figures, so-called *observer* and *behaviour* schemes.

One major example, that of the domain analysis & description of a road transport system, intersperses the methodology presentation of 38 examples. Section 1.9 completes that road transport system example.

1.2 Entities: Endurants and Perdurants

1.2.1 A Generic Domain Ontology – A Synopsis

Figure 1.4 on the next page shows an *upper ontology* for domains such a defined in Defn. 1 on the facing page.

Kai Sørlander’s Philosophy justifies our organising the *entities* of any describable domain, for example¹⁵, as follows: We shall review Fig. 1.4 by means of a top-down, left-traversal of the tree (whose root

¹⁴ We use the terms ‘*domain descriptions*’ and ‘*domain models*’ interchangeably.

¹⁵ We could organise the ontology differently: entities are either naturals, artefacts or living species, et cetera. If an upper node (●) satisfies a predicate \mathcal{P} then all descendant nodes do likewise.

is at the top). There are *describable* phenomena and there are phenomena that we cannot describe. The former we shall call *entities*. The *entities* are either *endurants*, “still” entities – existing in *space*, or *perdurants*, “alive” entities – existing also in *time*. *Endurants* are either *discrete* or *continuous* – in which

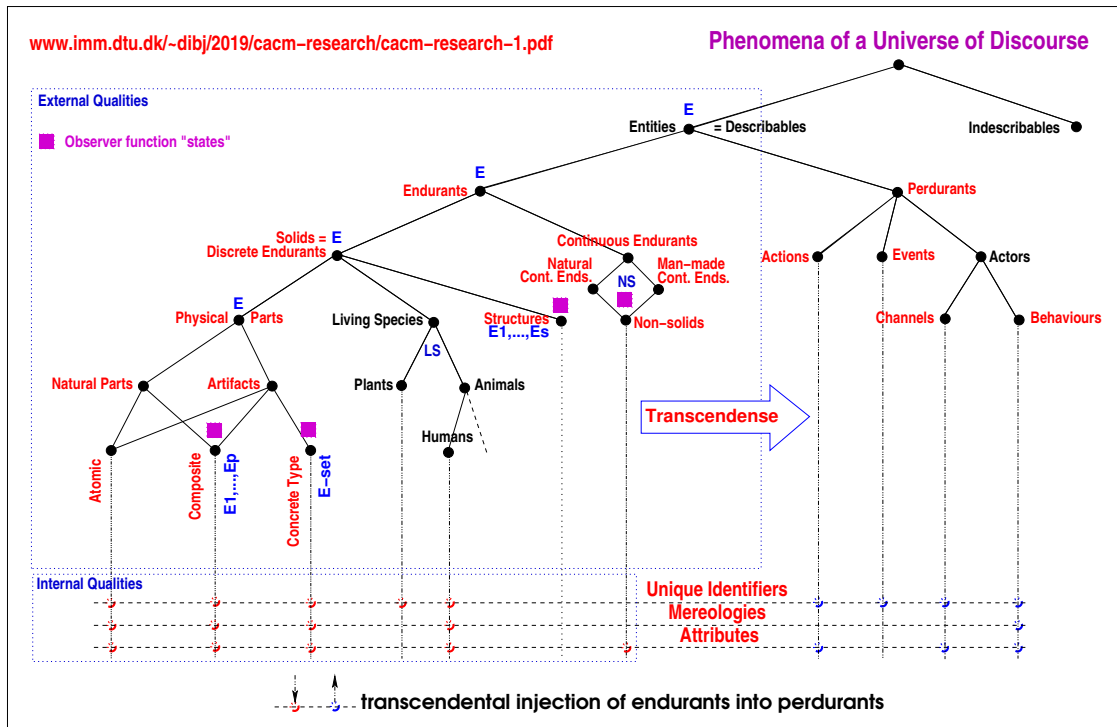


Fig. 1.4. An Upper Ontology for Domains

latter case we call them *materials*¹⁶. *Discrete endurants* are *physical parts*, *living species*, or are *structures*. *Physical parts* are either *naturals*, *artefacts*, i.e. man-made. Natural and man-made parts are either *atomic* or *composite*. We additionally analyse artefacts into either *components*¹⁷, or *sets of identically typed parts*. That additional analysis could also be expressed for natural parts but as we presently find no use for that we omit such further analysis. *Living Species* are either *plants* or *animals*. Among animals we have the *humans*. *Structures* consist of one or more endurants. Structures and components really are parts, but for pragmatic reasons we choose to not model them as [full fledged] parts. The categorisation into structures, natural parts, artefactual parts, plants, animals, and components is thus partly based in Sørlander’s Philosophy, partly pragmatic. The distinction between endurants and perdurants, are necessitated by Sørlander as being in space, respectively in space **and** time; discrete and continuous are motivated by arguments of natural sciences; structures and components are purely pragmatic; plants and animals, including humans, are necessitated by Kai Sørlander’s Philosophy. The distinction between natural, physical parts, and artefacts is not necessary in Sørlander’s Philosophy, but, we claim, necessary, philosophically, in order to perform the *intentional “pull”*, a transcendental deduction.

¹⁶ Please observe that *materials* were either *natural* or *artefactual*, but that we do not “bother” in this chapter. You may wish to slightly change the ontology diagram to reflect a distinction.

¹⁷ Whether a discrete endurant as we shall soon see, is treated as a part or a component is a matter of pragmatics. Again cf. Footnote 16.

On Pragmatics: We have used the term ‘pragmatic’ a few times. On one hand there is philosophy’s need for absolute clarity. On the other hand, when applying the natural part, artefactual part, and living species, concepts in practice, there can be a need for “loosening” up. As for example: a structure really is a collection of parts and relations between them. As we shall later see, parts are transcendently to be understood as behaviours. We know that modelling is imperative when we model a domain, but we may not wish to model a discrete endurant as a behaviour so we decide, pragmatically, to model it as a structure.

Our reference, here, to Kai Sørlander’s Philosophy, is very terse. We refer to a detailed research report: *A Philosophy of Domain Science & Engineering*¹⁸ for carefully reasoned arguments. That report is under continued revision: It reviews the domain analysis & description method; translates many of Sørlander’s arguments and relates, in detail, the “options” of the domain analysis & description approach to Sørlander’s Philosophy.

1.2.2 Universes of Discourse

By a **universe of discourse** we shall understand the same as the **domain of interest**, that is, the *domain* to be analysed & described ■

Example 1: Universes of Discourse

We refer to a number of Internet accessible experimental reports¹⁹ of descriptions of the following domains:

- **railways** [23, 88, 26],
- **container shipping** [33],
- **stock exchange** [48],
- **oil pipelines** [56],
- **“The Market”** [24],
- **Web systems** [47],
- **weather information** [67],
- **credit card systems** [63],
- **document systems** [72],
- **urban planning** [95],
- **swarms of drones** [69],
- **container terminals** [74]

It may be a **“large” domain**, that is, consist of many, as we shall see, *endurants* and *perdurants*, of many *parts*, *components* and *materials*, of many *humans* and *artefacts*, and of many *actors*, *actions*, *events* and *behaviours*.

Or it may be a **“small” domain**, that is, consist of a few such entities.

The choice of “boundaries”, that is, of how much or little to include, and of how much or little to exclude is entirely the choice of the domain engineer cum scientist: the choice is crucial, and is not always obvious. The choice delineates an *interface*, that is, that which is within the boundary, i.e., is in the domain, and that which is without, i.e., outside the domain, i.e., is the **context of the domain**, that is, the **external domain interfaces**. Experience helps set reasonable boundaries.

There are two “situations”: Either a domain analysis & description endeavour is pursued in order to prepare for a subsequent development of *requirements modelling*, in which case one tends to choose a **“narrow” domain**, that is, one that “fits”, includes, but not much more, the domain of interest for the requirements. Or a domain analysis & description endeavour is pursued in order to research a domain. *Either* one that can form the basis for subsequent engineering studies aimed, eventually at requirements development; in this case “wider” boundaries may be sought. *Or* one that experimentally “throws a larger net”, that is, seeks a “large” domain so as to explore interfaces between what is thought of as **internal system interfaces**.

Where, then, to start the *domain analysis & description* ? Either one can start “bottom-up”, that is, with atomic entities: endurants or perdurants, one-by-one, and work one’s way “out”, to include composite entities, again endurants or perdurants, to finally reach some satisfaction: *Eureka*, a goal has been reached.

¹⁸ <http://www.imm.dtu.dk/~dibj/2018/philosophy/filo.pdf>

Or one can start “top-down”, that is, “casting a wide net”. The choice is yours. Our presentation, however, is “top down”: most general domain aspects first.

Example 2: Universe of Discourse

The universe of discourse is road transport systems. We analyse & describe not the class of all road transport systems but a representative subclass, UoD, is structured into such notions as a road net, RN, of hubs, H, (nodes, i.e., street intersections) and links, L, (edges, i.e., street segments between intersections); a fleet of vehicles, FV, structured into companies, BC, of buses, B, and pools, PA, of private automobiles, A (et cetera); et cetera.

1.2.3 Entities

Characterisation 1 Entity: By an **entity** we shall understand a **phenomenon**, i.e., something that can be *observed*, i.e., be seen or touched by humans, or that can be *conceived* as an *abstraction* of an entity; alternatively, a phenomenon is an entity, *if it exists, it is “being”, it is that which makes a “thing” what it is: essence, essential nature* [170, Vol. I, pg. 665] ■

Analysis Prompt 1 *is_entity*: *The domain analyser analyses “things” (θ) into entities or non-entities. The method provides the domain analysis prompt:*

- *is_entity* – where *is_entity*(θ) holds if θ is an entity²⁰ ■

is_entity is said to be a *prerequisite prompt* for all other prompts.

To sum up: *An entity is what we can analyse and describe using the analysis & description prompts outlined in this chapter.*

The *entities* that we are concerned with are those with which Kai Sørlander’s Philosophy is likewise concerned. They are the ones that are *unavoidable* in any description of any possible world. And then, which are those entities? In both [238] and [245] Kai Sørlander rationally deduces that these entities must be in *space* and *time*, must satisfy laws of physics – like those of Newton and Einstein, but among them are also *living species: plants and animals* and hence *humans*. The *living species*, besides still being in *space* and *time*, and satisfying laws of physics, must satisfy further properties – which we shall outline in Sects. 1.3.4 on Page 15 and 1.5.3 on Page 32.

1.2.4 Endurants and Perdurants

The concepts of endurants and perdurants are not present in, that is, are not essential to Sørlander’s Philosophy. Since our departure point is that of *computing science* where, eventually, conventional computing performs operations on, i.e. processes data, we shall, however, introduce these two notions: *endurant* and *perdurant*. The former, in a rough sense, “corresponds” to data; the latter, similarly, to processes.

Characterisation 2 Endurant: By an **endurant** we shall understand an entity that can be observed, or conceived and described, as a “complete thing” at no matter which given snapshot of time; alternatively an entity is *endurant* if it is capable of *enduring*, that is *persist*, “*hold out*” [170, Vol. I, pg. 656]. Were we to “freeze” time we would still be able to observe the entire *endurant* ■

Example 3: Endurants

Geography Endurants: *The geography of an area, like some island, or a country, consists of its geography – “the lay of the land”, the geodetics of this land, the meteorology of it, et cetera.* **Railway System Endurants:** *Example railway system endurants are: a railway system, its net, its individual tracks, switch points, trains, their individual locomotives, et cetera.*

Analysis Prompt 2 *is_endurant*: *The domain analyser analyses an entity, ϕ , into an *endurant* as prompted by the domain analysis prompt:*

²⁰ Analysis prompt definitions and description prompt definitions and schemes are delimited by ■

- *is_endurant* – ϕ is an endurant if *is_endurant*(ϕ) holds.

is_entity is a prerequisite prompt for *is_endurant* ■

Characterisation 3 Perdurant: By a **perdurant** we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time. Were we to freeze time we would only see or touch a fragment of the perdurant, alternatively an entity is perdurant if it endures continuously, over time, persists, lasting [170, Vol. II, pg. 1552] ■

Example 4: Perdurants

Geography: Example geography perdurants are: the continuous changing of the weather (meteorology); the erosion of coast lines; the rising of some land and the “sinking” of other land areas; volcano eruptions; earth quakes; et cetera.

Railway Systems: Example railway system perdurants are: the ride of a train from one railway station to another; and the stop of a train at a railway station from some arrival time to some departure time.

Analysis Prompt 3 *is_perdurant*: The domain analyser analyses an entity *e* into perdurants as prompted by the **domain analysis prompt**:

- *is_perdurant* – *e* is a perdurant if *is_perdurant*(*e*) holds.

is_entity is a prerequisite prompt for *is_perdurant* ■

Occurrent is a synonym for perdurant.

1.3 Endurants: Analysis of External Qualities

1.3.1 Discrete and Continuous Endurants

Characterisation 4 Discrete Endurant: By a **discrete endurant** we shall understand an endurant which is separate, individual or distinct in form or concept ■

To simplify matters we shall allow separate elements of a discrete endurant to be continuous!

Example 5: Discrete Endurants

The individual endurants of the above example of railway system endurants were all discrete. Here are examples of discrete endurants of pipeline systems. A pipeline and its individual units: pipes, valves, pumps, forks, etc.

Analysis Prompt 4 *is_discrete*: The domain analyser analyses endurants *e* into discrete entities as prompted by the **domain analysis prompt**:

- *is_discrete* – *e* is discrete if *is_discrete*(*e*) holds ■

Characterisation 5 Continuous Endurant: By a **continuous endurant** we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern ■

We shall prefer to refer to continuous endurants as *materials* and otherwise cover materials in Sect. 1.3.6 on Page 17.

Example 6: Materials

Examples of materials are: water, oil, gas, compressed air, etc. A container, which we consider a discrete endurant, may contain a material, like a gas pipeline unit may contain gas.

Analysis Prompt 5 *is_continuous*: The domain analyser analyses endurants *e* into continuous entities as prompted by the **domain analysis prompt**:

- *is_continuous* – *e* is continuous if *is_continuous*(*e*) holds ■

Continuity shall here not be understood in the sense of mathematics. Our definition of ‘continuity’ focused on *prolonged, without interruption, in an unbroken series or pattern*. In that sense materials shall be seen as ‘continuous’. The mathematical notion of ‘continuity’ is an abstract one. The endurant notion of ‘continuity’ is physical one.

1.3.2 Discrete Endurants

We analyse discrete endurants into *physical parts*, *living species* and *structures*. Physical parts and living species can be identified as separate entities – following Kai Sørlander’s Philosophy. To model discrete endurants as structures represent a pragmatic choice which relieves the domain describer from transcendently considering structures as behaviours.

Physical Parts

Characterisation 6 Physical Parts: By a *physical part* we shall understand a discrete endurant existing in time and subject to laws of physics, including the *causality principle* and *gravitational pull*²¹ ■

Analysis Prompt 6 *is_physical_part*: The domain analyser analyses “things” (η) into physical part. The method provides the **domain analysis prompt**:

- *is_physical_part* – where *is_physical_part*(η) holds if η is a physical part ■

Section 1.3.3 continues our treatment of physical parts.

Living Species

Definition 3 Living Species, I: By a *living species* we shall understand a discrete endurant, subject to laws of physics, and additionally subject to *causality of purpose*.²² [Defn. 9 on Page 15 elaborates further on this point] ■

Analysis Prompt 7 *is_living_species*: The domain analyser analyses “things” (e) into living species. The method provides the **domain analysis prompt**:

- *is_living_species* – where *is_living_species*(e) holds if e is a living species ■

Living species have a *form* they can *develop* to reach; they are *causally* determined to *maintain* this form; and they do so by *exchanging matter* with an *environment*. We refer to [73] for details. Section 1.3.4 continues our treatment of living species.

Structures

Definition 4 Structure: By a **structure** we shall understand a discrete endurant which the domain engineer chooses to describe as consisting of one or more endurants, whether discrete or continuous, but to **not** endow with **internal qualities**: unique identifiers, mereology or attributes ■

Structures are “conceptual endurants”. A *structure* “gathers” one or more endurants under “one umbrella”, often simplifying a presentation of some elements of a domain description. Sometimes, in our domain modelling, we choose to model an endurant as a *structure*, sometimes as a *physical part*; it all depends on what we wish to focus on in our domain model. As such structures are “compounds” where we are

²¹ This characterisation is the result of our study of relations between philosophy and computing science, notably influenced by Kai Sørlander’s Philosophy. We refer to our research report [73, www.imm.dtu.dk/~dibj/2018/philosophy-filo.pdf].

²² See Footnote 21.

interested only in the (external and internal) qualities of the elements of the compound, but not in the qualities of the structure itself.

Example 7: Structures

A transport system is modelled as structured into a road net structure and an automobile structure. The road net structure is then structured as a pair: a structure of hubs and a structure of links. These latter structures are then modelled as set of hubs, respectively links.

Example 8: Structures – Contd.

We could have modelled the road net structure as a composite part with unique identity, mereology and attributes which could then serve to model a road net authority. We could have modelled the automobile structure as a composite part with unique identity, mereology and attributes which could then serve to model a department of vehicles. ■

The concept of *structure* is new. Whether to analyse & describe a discrete endurant into a structure or a physical part is a matter of choice. If we choose to analyse a discrete endurant into a *physical part* then it is because we are interested in endowing the part with *qualities*, the unique identifiers, mereology and one or more attributes. If we choose to analyse a discrete endurant into a *structure* then it is because we are **not** interested in endowing the endurant with *qualities*. When we choose that an endurant sort should be modelled as a part sort with unique identification, mereology and proper attributes, then it is because we eventually shall consider the part sort as being the basis for transcendently deduced behaviours.

Analysis Prompt 8 *is_structure*: The domain analyser analyse endurants, *e*, into structure entities as prompted by the **domain analysis prompt**:

- *is_structure* *e* is a structure if *is_structure*(*e*) holds ■

We shall now treat the external qualities of discrete endurants: *physical parts* (Sect. 1.3.3) and *living species* (Sect. 1.3.4). After that we cover *components* (Sect. 1.3.5), *materials* (Sect. 1.3.6) and *artefacts* (physical man-made parts, Sect. 1.3.3). We remind the reader that in this section, i.e. Sect. 1.3, we cover only the *analysis calculus* for external qualities; the *description calculus* for external qualities is treated in Sect. 1.4. The analysis and description calculi for internal qualities is covered in Sect. 1.5.

1.3.3 Physical Parts

Physical parts are either *natural parts*, or *components*, or *sets of parts* of the same type, or are *artefacts* i.e. man-made parts. The categorisation of physical parts into these four is pragmatic. *Physical parts* follow from Kai Sørlander’s Philosophy. *Natural parts* are what Sørlander’s Philosophy is initially about. *Artefacts* follow from *humans* acting according to their *purpose* in making “physical parts”. *Components* is a simplification of natural and man-made parts. *Set of parts* is a simplification of composite natural and composite man-made parts as will be made clear in Sect. 1.4.2.

Natural Parts

Characterisation 7 Natural Parts: Natural parts are in *space* and *time*; are subject to the *laws of physics*, and also subject to the *principle of causality* and *gravitational pull* ■

The above is a factual characterisation of natural parts. The below is our definition – such as we shall model natural parts.

Definition 5 Natural Part: By a **natural part** we shall understand a *physical part* which the domain engineer chooses to endow with all three **internal qualities**: unique identification, mereology, and one or more attributes ■

Artefacts

Characterisation 8 Man-made Parts: Artefacts: Artefacts are man-made either discrete or continuous durants. In this section we shall only consider discrete durants. Man-made continuous durants are not treated separately but are lumped with natural materials. Artefacts are subject to the *laws of physics* ■

The above is a factual characterisation of discrete artefacts. The below is our definition – such as we shall model discrete artefacts.

Definition 6 Artefact: By an *artefact* we shall understand a *man-made physical part* which, like for *natural parts*, the domain engineer chooses to endow with all three *internal qualities*: unique identification, mereology, and one or more attributes ■

We shall assume, cf. Sect. 1.5.3 [*Attributes*], that *artefacts* all come with an *attribute* of kind *intent*, that is, a set of purposes for which the artefact was constructed, and for which it is intended to serve. We continue our treatment of artefacts in Sect. 1.3.7 below.

Parts

We revert to our treatment of parts.

Example 9: Parts

The geography examples (of Page 10) of are all natural parts. The railway system examples (of Page 10) are all artefacts ■

Except for the *intent* attribute of artefacts, we shall, in the following, treat *natural* and *artefactual* parts on par, i.e., just as physical parts.

Analysis Prompt 9 *is_part*: The domain analyser analyse durants, *e*, into part entities as prompted by the **domain analysis prompt**:

- *is_part e* is a part if *is_part (e)* holds ■

Atomic and Composite Parts:

A distinguishing quality of natural and artefactual parts is whether they are atomic or composite. Please note that we shall, in the following, examine the concept of parts in quite some detail. That is, parts become the domain durants of main interest, whereas components, structures and materials become of secondary interest. This is a choice. The choice is based on pragmatics. It is still the domain analyser cum describers' choice whether to consider a discrete durant a part or a component, or a structure. If the domain engineer wishes to investigate the details of a discrete durant then the domain engineer chooses to model²³ the discrete durant as a part otherwise as a component.

Atomic Parts

Definition 7 Atomic Part: Atomic parts are those which, in a given context, are deemed to *not* consist of meaningful, separately observable proper *sub-parts*. A **sub-part** is a part ■

Analysis Prompt 10 *is_atomic*: The domain analyser analyses a discrete durant, i.e., a part *p* into an atomic durant:

²³ We use the term *to model* interchangeably with the composite term *to analyse & describe*; similarly a *model* is used interchangeably with an *analysis & description*.

- *is_atomic*: p is an atomic endurant if *is_atomic*(p) holds ■

Example 10: Atomic Road Net Parts

From one point of view all of the following can be considered atomic parts: hubs, links²⁴, and automobiles.

Composite Parts

Definition 8 Composite Part: Composite parts are those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts ■

Analysis Prompt 11 *is_composite*: The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:

- *is_composite*: p is a composite endurant if *is_composite*(p) holds ■

is_discrete is a prerequisite prompt of both *is_atomic* and *is_composite*.

Example 11: Composite Automobile Parts

From another point of view all of the following can be considered composite parts: an automobile, consisting of, for example, the following parts: the engine train, the chassis, the car body, the doors and the wheels. These can again be considered composite parts.

1.3.4 Living Species

We refer to Sect. 1.3.2 for our first characterisation (Page 12) of the concept of *living species*²⁵: a discrete endurant existing in time, subject to laws of physics, and additionally subject to *causality of purpose*²⁶

Definition 9 Living Species, II: Living species must have some *form they can be developed to reach*; which they must be *causally determined to maintain*. This *development and maintenance* must further in an *exchange of matter with an environment*. It must be possible that living species occur in one of two forms: one form which is characterised by *development, form and exchange*; another form which, **additionally**, can be characterised by the *ability to purposeful movement* The first we call **plants**, the second we call **animals** ■

Analysis Prompt 12 *is_living_species*: The domain analyser analyse discrete endurants, ℓ , into living species entities as prompted by the **domain analysis prompt**:

- *is_living_species* – where *is_living_species* ℓ holds if ℓ is a living species ■

Plants

We start with some examples.

Example 12: Plants

Although we have not yet come across domains for which the need to model the living species of plants were needed, we give some examples anyway: grass, tulip, rhododendron, oak tree.

Analysis Prompt 13 *is_plant*: The domain analyser analyses “things” (ℓ) into a plant. The method provides the **domain analysis prompt**:

- *is_plant* – where *is_plant*(ℓ) holds if ℓ is a plant ■

The predicate *is_living_species*(ℓ) is a prerequisite for *is_plant*(ℓ).

²⁴ Hub \equiv street intersection; link \equiv street segments with no intervening hubs.

²⁵ See analysis prompt 7 on Page 12.

²⁶ See Footnote 21 on Page 12.

Animals

Definition 10 Animal: We refer to the initial definition of *living species* above – while ephasizing the following traits: (i) *form animals can be developed to reach*; (ii) *causally determined to maintain*. (iii) *development and maintenance in an exchange of matter with an environment*, and (iv) *ability to purposeful movement* ■

Analysis Prompt 14 *is_animal*: *The domain analyser analyses “things” (ℓ) into an animal. The method provides the domain analysis prompt:*

- *is_animal – where $is_animal(\ell)$ holds if ℓ is an animal* ■

The predicate $is_living_species(\ell)$ is a prerequisite for $is_animal(\ell)$.

Example 13: Animals

Although we have not yet come across domains for which the need to model the living species of animals, in general, were needed, we give some examples anyway: *dolphin, goose cow dog, lion, fly*.

We have not decided, for this chapter, whether to model animals singly or as sets²⁷ of such.

Humans

Definition 11 Human: A *human* (a *person*) is an *animal*, cf. Definition 10, with the additional properties of having *language*, being *conscious of having knowledge* (of its own situation), and *responsibility* ■

Analysis Prompt 15 *is_human*: *The domain analyser analyses “things” (ℓ) into a human. The method provides the domain analysis prompt:*

- *is_human – where $is_human(\ell)$ holds if ℓ is a human* ■

The predicate $is_animal(\ell)$ is a prerequisite for $is_human(\ell)$.

We refer to [73, Sects. 10.4–10.5] for a specific treatment of living species, animals and humans, and to [73] in general for the philosophy background for rationalising the treatment of living species, animals and humans.

We have not, in our many experimental domain modelling efforts had occasion to model humans; or rather: we have modelled, for example, automobiles as possessing human qualities, i.e., “subsuming humans”. We have found, in these experimental domain modelling efforts that we often confer anthropomorphic qualities on artefacts²⁸, that is, that these artefacts have human characteristics. You, the reader are reminded that when some programmers try to explain their programs they do so using such phrases as *and here the program does ... so-and-so* !

1.3.5 Components

Definition 12 Component: By a **component** we shall understand a discrete endurant which we, the domain analyser cum describer chooses to **not** endow with **mereology** ■

Components are discrete endurants. Usually they come in sets. That is, sets of sets of components of different sorts (cf. Sect. 1.4.4 on Page 22). A discrete endurant can (itself) “be” a set of components. But physical parts may contain ($has_components$) components: natural parts may contain natural components,

²⁷ school of dolphins, flock of geese, herd of cattle, pack of dogs, pride of lions, swarm of flies,

²⁸ Cf. Sect. 1.3.7 below.

artefacts may contain natural and artefactual components. We leave it to the reader to provide analysis predicates for natural and artefactual “componentry”.

Example 14: Components

A natural part, say a land area may contain gravel pits of sand, clay pits tar pits and other “pits”. An artefact, say a postal letter box may contain letters, small parcels, newspapers and advertisement brochures.

Analysis Prompt 16 *has_components*: The domain analyser analyses discrete endurants e into component entities as prompted by the domain analysis prompt:

- *has_components*(p) holds if part p potentially may contain components ■

We refer to Sect. 1.4.4 on Page 22 for further treatment of the concept of *components*.

1.3.6 Continuous Endurants \equiv Materials

Definition 13 Material: By a **material** we shall understand a continuous endurant ■

Materials are continuous endurants. Usually they come in sets. That is, sets of materials of different sorts (cf. Sect. 1.4.5 on Page 23). So an endurant can (itself) “be” a set of materials. But physical parts may contain (*has_materials*) materials: natural parts may contain natural materials, artefacts may contain natural and artefactual materials. We leave it to the reader to provide analysis predicates for natural and artefactual “materials”.

Example 15: Natural and Man-made Materials

A natural part, say a land area, may contain lakes, rivers, irrigation dams and border seas.
An artefact, say an automobile, usually contains gasoline, lubrication oil, engine cooler liquid and window screen washer water.

Analysis Prompt 17 *has_materials*: The domain analysis prompt:

- *has_materials*(p) yields **true** if part $p:P$ potentially may contain materials otherwise false ■

We refer to Sect. 1.4.5 on Page 23 for further treatment of the concept of *materials*. We shall define the terms unique identification, mereology and attributes in Sects. 1.5.1–1.5.3.

1.3.7 Artefacts

Definition 14 Artefacts: By artefacts we shall understand a man-made physical part or a man-made material ■

Example 16: More Artefacts

From the shipping industry: ship, container vessels, container, container stack, container terminal port, harbour.

Analysis Prompt 18 *is_artefact*: The domain analyser analyses “things” (p) into artefacts. The method provides the domain analysis prompt:

- *is_artefact* – where *is_artefact*(p) holds if p is an artefact ■

1.3.8 States

Definition 15 State: By a *state* we shall understand any number of physical parts and/or materials each possessing as we shall later introduce them at least one dynamic attribute. There is no need to introduce time at this point ■

Example 17: Artefactual States

The following endurants are examples of states (including being elements of state compounds): pipe units (pipes, valves, pumps, etc.) of pipe-lines; hubs and links of road nets (i.e., street intersections and street segments); automobiles (of transport systems).

The notion of *state* becomes relevant in Sect. 1.8. We shall there exemplify states further: example *Constants and States [Indexed States]* Page 42.

1.4 Endurants: The Description Calculus

1.4.1 Parts: Natural or Man-made

The observer functions of this section apply to both natural parts and man-made parts (i.e., artefacts).

On Discovering Endurant Sorts

Our aim now is to present the basic principles that let the domain analyser decide on *part sorts*. We observe parts one-by-one.

(α) *Our analysis of parts concludes when we have “lifted” our examination of a particular part instance to the conclusion that it is of a given sort²⁹, that is, reflects a formal concept.*

Thus there is, in this analysis, a “eureka”, a step where we shift focus from the concrete to the abstract, from observing specific part instances to postulating a sort: from one to the many. If p is a part of sort P , then we express that as: $p:P$.

Analysis Prompt 19 *observe_endurant_sorts*: The **domain analysis prompt**:

- *observe_endurant_sorts*

directs the domain analyser to observe the sub-endurants of an endurant e and to suggest their sorts. Let $observe_endurant_sorts(e) = \{e_1:E_1, e_2:E_2, \dots, e_m:E_m\}$ ■

(β) *The analyser analyses, for each of these endurants, e_i , which formal concept, i.e., sort, it belongs to; let us say that it is of sort E_k ; thus the sub-parts of p are of sorts $\{E_1, E_2, \dots, E_m\}$. Some E_k may be natural parts, other artefacts (man-made parts) or structures, and yet others may be components or materials. And parts may be either atomic or composite.*

The domain analyser continues to examine a finite number of other composite parts: $\{p_j, p_\ell, \dots, p_n\}$. It is then “discovered”, that is, decided, that they all consists of the same number of sub-parts $\{e_{i_1}, e_{i_2}, \dots, e_{i_m}\}, \{e_{j_1}, e_{j_2}, \dots, e_{j_m}\}, \{e_{\ell_1}, e_{\ell_2}, \dots, e_{\ell_m}\}, \dots, \{e_{n_1}, e_{n_2}, \dots, e_{n_m}\}$, of the same, respective, endurant sorts.

(γ) *It is therefore concluded, that is, decided, that $\{e_i, e_j, e_\ell, \dots, e_n\}$ are all of the same endurant sort P with observable part sub-sorts $\{E_1, E_2, \dots, E_m\}$.*

²⁹ We use the term ‘sort’ for abstract types, i.e., for the type of values whose concrete form we are not describing. The term ‘sort’ is commonly used in algebraic semantics [231].

Above we have *type-font-highlighted* three sentences: (α, β, γ) . When you analyse what they “prescribe” you will see that they entail a “depth-first search” for part sorts. The β sentence says it rather directly: “*The analyser analyses, for each of these parts, p_k , which formal concept, i.e., part sort it belongs to.*” To do this analysis in a proper way, the analyser must (“recursively”) analyse structures into sub-structures, parts, components and materials, and parts “down” to their atomicity. Components and materials are considered “atomic”, i.e., to not contain further analysable endurants. For the structures, parts (whether natural or man-made), components and materials of the structure the analyser cum describer decides on their sort, and work (“recurse”) their way “back”, through possibly intermediate endurants, to the p_k s. Of course, when the analyser starts by examining atomic parts, components and materials, then their endurant structure and part analysis “recursion” is not necessary.

Endurant Sort Observer Functions:

The above analysis amounts to the analyser first “applying” the *domain analysis* prompt `is_composite(e)` to a discrete endurant, e , where we now assume that the obtained truth value is **true**. Let us assume that endurants $e:E$ consist of sub-endurants of sorts $\{E_1, E_2, \dots, E_m\}$. Since we cannot automatically guarantee that our domain descriptions secure that E and each E_i ($1 \leq i \leq m$) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 1 *observe_endurant_sorts*: *If `is_composite(p)` holds, then the analyser “applies” the domain description prompt*

- `observe_endurant_sorts(p)`

resulting in the analyser writing down the endurant sorts and endurant sort observers domain description text according to the following schema:

1. *observe_endurant_sorts* Observer schema

Narration:

- [s] ... narrative text on sorts ...
- [o] ... narrative text on sort observers ...
- [p] ... narrative text on proof obligations ...

Formalisation:

type

- [s] E ,
- [s] E_i $i:[1..m]$ **comment:** E_i $i:[1..m]$ abbreviates E_1, E_2, \dots, E_m

value

- [o] **obs_** E_i : $E \rightarrow E_i$ $i:[1..m]$

proof obligation [Disjointness of endurant sorts]

- [p] $\mathcal{P}\mathcal{O} : \forall e:(E_1|E_2|\dots|E_m) \cdot \wedge \{\mathbf{is_}E_i(e) \equiv \wedge \{\sim \mathbf{is_}E_j(e)|j:[1..m] \setminus \{i\}\}|i:[1..m]\}$

The $\mathbf{is_}E_j(e)$ is defined by E_i $i:[1..m]$. `is_composite` is a **prerequisite prompt** of `observe_endurant_sorts`. That is, the composite may satisfy `is_natural` or `is_artefact` ■

Note: The above schema as well as the following schemes introduce, i.e., define in terms of a function signature, a number of functions whose names begin with bold-faced **obs_...**, **uid_...**, **mereo_...**, **attr_...** et cetera. These observer functions are one of the bases of domain descriptions.

We do not here state techniques for discharging proof obligations.³⁰

Example 18: Composite Endurant Sorts

1 There is the universe of discourse, UoD .

It is structured into

³⁰ – such techniques are given in standard texts on formal specification languages.

2 a road net, *RN*, and
 3 a fleet of vehicles, *FV*.

Both are structures.

type
 1 UoD **axiom** $\forall uod:UoD \cdot is_structure(uod)$.
 2 RN **axiom** $\forall rn:RN \cdot is_structure(rn)$.
 3 FV **axiom** $\forall fv:FV \cdot is_structure(fv)$.

value
 2 obs_RN: UoD \rightarrow RN
 3 obs_FV: UoD \rightarrow FV ■

Note: A proper description has two texts, a *narrative* and a *formalisation* each is itemised and items are pairwise numbered.

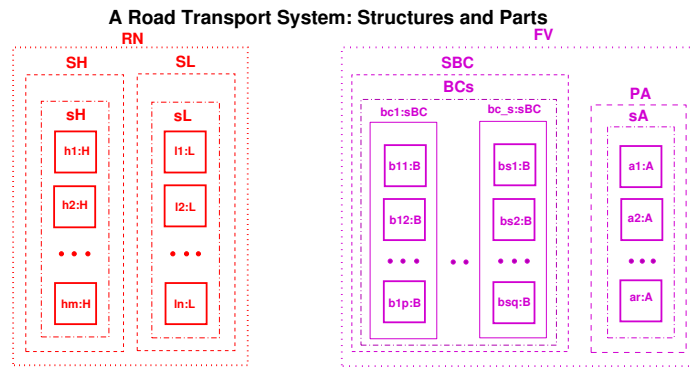


Fig. 1.5. A Road Transport System ■

Example 19: Structures

4 The road net consists of
 a a structure, *SH*, of hubs and
 b a structure, *SL*, of links.
 5 The fleet of vehicles consists of
 a a structure, *SBC*, of bus companies, and
 b a structure, *PA*, a pool of automobiles.

type
 4a SH **axiom** $\forall sh:SH \cdot is_structure(sh)$
 4b SL **axiom** $\forall sl:SL \cdot is_structure(sl)$
 5a SBC **axiom** $\forall sbc:SBC \cdot is_structure(bc)$
 5b PA **axiom** $\forall pa:PA \cdot is_structure(pa)$

value
 4a obs_SH: RN \rightarrow SH
 4b obs_SL: RN \rightarrow SL
 5a obs_BC: FV \rightarrow BC
 5b obs_PA: FV \rightarrow PA

1.4.2 Concrete Part Types

Sometimes it is expedient to ascribe concrete types to sorts.

Analysis Prompt 20 *has_concrete_type*: The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P , whether atomic or composite, as a concrete type, T . That decision is prompted by the holding of the **domain analysis prompt**:

- *has_concrete_type*.

is_discrete is a prerequisite prompt of *has_concrete_type* ■

The reader is reminded that the decision as to whether an abstract type is (also) to be described concretely is entirely at the discretion of the domain engineer.

Domain Description Prompt 2 *observe_part_type*: Then the domain analyser applies the **domain description prompt**:

- *observe_part_type*(p)³¹

to parts $p:P$ which then yield the part type and part type observers domain description text according to the following schema:

2. *observe_part_type* Observer schema

Narration:

- [t₁] ... narrative text on sorts and types S_i ...
- [t₂] ... narrative text on types T ...
- [o] ... narrative text on type observers ...

Formalisation:

type

- [t₁] $S_1, S_2, \dots, S_m, \dots, S_n,$
- [t₂] $T = \mathcal{E}(S_1, S_2, \dots, S_n)$

value

- [o] **obs_T**: $P \rightarrow T$ ■

Here $S_1, S_2, \dots, S_m, \dots, S_n$ may be any types, including part sorts, where $0 \leq m \leq n \geq 1$, where m is the number of new (atomic or composite) sorts, and where $n - m$ is the number of concrete types (like **Bool**, **Int**, **Nat**) or sorts already analysed & described. and $\mathcal{E}(S_1, S_2, \dots, S_n)$ is a type expression Usually it is wise to restrict the part type definitions, $T_i = \mathcal{E}_i(Q, R, \dots, S)$, to simple type expressions.³² The type name, T , of the concrete type, as well as those of the auxiliary types, S_1, S_2, \dots, S_m , are chosen by the domain describer: they may have already been chosen for other sort-to-type descriptions, or they may be new.

Example 20: Concrete Part Types

- 6 The structure of hubs is a set, sH , of atomic hubs, H .
- 7 The structure of links is a set, sL , of atomic links, L .
- 8 The structure of buses is a set, sBC , of composite bus companies, BC .
- 9 The composite bus companies, BC , are sets of buses, sB .
- 10 The structure of private automobiles is a set, sA , of atomic automobiles, A .

6 $H, sH = H\text{-set}$ axiom $\forall h:H \cdot \text{is_atomic}(h)$

³¹ *has_concrete_type* is a prerequisite prompt of *observe_part_type*.

³² $T = A\text{-set}$ or $T = A^*$ or $T = \text{ID} \rightarrow_n A$ or $T = A_t | B_t | \dots | C_t$ where ID is a sort of unique identifiers, $T = A_t | B_t | \dots | C_t$ defines the disjoint types $A_t ::= \text{mk}A_t(s:A_s)$, $B_t ::= \text{mk}B_t(s:B_s)$, ..., $C_t ::= \text{mk}C_t(s:C_s)$, and where A, A_s, B_s, \dots, C_s are sorts. Instead of $A_t ::= \text{mk}A_t(a:A_s)$, etc., we may write $A_t :: A_s$ etc.

```

7 L, sL = L-set axiom  $\forall l:L \cdot \text{is\_atomic}(l)$ 
8 BC, BCs = BC-set axiom  $\forall bc:BC \cdot \text{is\_composite}(bc)$ 
9 B, Bs = B-set axiom  $\forall b:B \cdot \text{is\_atomic}(b)$ 
10 A, sA = A-set axiom  $\forall a:A \cdot \text{is\_atomic}(a)$ 
value
6 obs_sH: SH  $\rightarrow$  sH
7 obs_sL: SL  $\rightarrow$  sL
8 obs_sBC: SBC  $\rightarrow$  BCs
9 obs_Bs: BCs  $\rightarrow$  Bs
10 obs_sA: SA  $\rightarrow$  sA ■

```

1.4.3 On Endurant Sorts

Derivation Chains

Let E be a composite sort. Let E_1, E_2, \dots, E_m be the part sorts “discovered” by means of `observe_endurant_sorts(e)` where $e:E$. We say that E_1, E_2, \dots, E_m are (immediately) **derived** from E . If E_k is derived from E_j and E_j is derived from E_i , then, by transitivity, E_k is **derived** from E_i .

No Recursive Derivations:

We “mandate” that if E_k is derived from E_j then there E_j is different from E_k and there can be no E_k derived from E_j , that is, E_k cannot be derived from E_k . That is, we do not “provide for” recursive domain sorts. It is not a question, actually of allowing recursive domain sorts. It is, we claim to have observed, in very many *analysis & description* experiments, that there are no recursive domain sorts!³³

Names of Part Sorts and Types:

The domain analysis & description text prompts `observe_endurant_sorts`, as well as the below-defined `observe_part_type`, `observe_component_sorts` and `observe_material_sorts`, – as well as the further below defined `attribute_names`, `observe_material_sorts`, `observe_unique_identifier`, `observe_mereology` and `observe_attributes` prompts introduced below – “yield” type names. That is, it is as if there is a reservoir of an indefinite-size set of such names from which these names are selected, and once obtained are never again selected. There may be domains for which two distinct part sorts may be composed from identical part sorts. *In this case the domain analyser indicates so by prescribing a part sort already introduced.*

1.4.4 Components

We refer to Sect. 1.3.5 on Page 17 for our initial treatment of ‘components’.

Domain Description Prompt 3 *observe_component_sorts*: The domain description prompt:

- *observe_component_sorts(p)*

³³ Some readers may object, but we insist! If *trees* are brought forward as an example of a recursively definable domain, then we argue: Yes, trees can be recursively defined, but it is not recursive. Trees can, as well, be defined as a variant of graphs, and you wouldn’t claim, would you, that graphs are recursive?

yields the component sorts and component sort observer domain description text according to the following schema – whether or not the actual part p contains any components:

3. observe_component_sorts Observer schema

Narration:

- [s] ... narrative text on component sorts ...
- [o] ... narrative text on component observers ...
- [p] ... narrative text on component sort proof obligations ...

Formalisation:

type

- [s] K_1, K_2, \dots, K_n
- [s] $K = K_1 | K_2 | \dots | K_n$
- [s] $KS = K\text{-set}$

value

- [o] **obs_components_P**: $P \rightarrow KS$

Proof Obligation: [Disjointness of Component Sorts]

- [p] $\mathcal{PO}: \forall k_i: (K_1 | K_2 | \dots | K_n) \bullet \wedge \text{is_}K_i(k_i) \equiv \wedge \{ \sim \text{is_}K_j(k_j) | j: [1..n] \setminus \{i\} \} \ i: [1..n]$ ■

The $\text{is_}K_j(e)$ is defined by $K_i, i: [1..n]$.

Example 21: Components

To illustrate the concept of components we describe timber yards, waste disposal areas, road material storage yards, automobile scrap yards, and the like as special “cul de sac” hubs with components. Here we describe road material storage yards.

- 11 Hubs may contain components, but only if the hub is connected to exactly one link.
- 12 These “cul-de-sac” hub components may be such things as Sand, Gravel, Cobble Stones, Asphalt, Cement or other.

value

- 11 **has_components**: $H \rightarrow \text{Bool}$

type

- 12 Sand, Gravel, Stones, Asphalt, Cement, ...
- 12 $KS = (\text{Sand} | \text{Gravel} | \text{Stones} | \text{Asphalt} | \text{Cement} | \dots)\text{-set}$

value

- 11 **obs_components_H**: $H \rightarrow KS$
- 11 **pre**: $\text{obs_components_H}(h) \equiv \text{card mereo}(h) = 1$ ■

We have presented one way of tackling the issue of describing components. There are other ways. We leave those ‘other ways’ to the reader. We are not going to suggest techniques and tools for analysing, let alone ascribing qualities to components. We suggest that conventional abstract modelling techniques and tools be applied.

1.4.5 Materials

We refer to Sect. 1.3.6 on Page 17 for our initial treatment of ‘materials’. Continuous endurants (i.e., **materials**) are entities, m , which satisfy:

- $\text{is_material}(e) \equiv \text{is_continuous}(e)$

If $\text{is_material}(e)$ holds then we can apply the **domain description prompt**: $\text{observe_material_sorts}(e)$.

Domain Description Prompt 4 *observe_material_sorts*: The domain description prompt:

- *observe_material_sorts(e)*

yields the material sorts and material sort observers' domain description text according to the following schema whether or not part *p* actually contains materials:

4. *observe_material_sorts* Observer schema**Narration:**

- [s] ... narrative text on material sorts ...
- [o] ... narrative text on material sort observers ...
- [p] ... narrative text on material sort proof obligations ...

Formalisation:**type**

- [s] M_1, M_2, \dots, M_n
- [s] $M = M_1 \mid M_2 \mid \dots \mid M_n$
- [s] $MS = M\text{-set}$

value

- [o] $\text{obs}_{M_i}: P \rightarrow M, [i:1..n]$

proof obligation [Disjointness of Material Sorts]

- [p] $\mathcal{P}\mathcal{O}: \forall m_i: M \cdot \bigwedge \{ \text{is}_{M_i}(m_i) \equiv \bigwedge \{ \sim \text{is}_{M_j}(m_j) \mid j \in \{1..m\} \setminus \{i\} \} \mid i: [1..n] \}$

The $\text{is}_{M_j}(e)$ is defined by $M_i, i: [1..n]$.

Let us assume that parts $p:P$ embody materials of sorts $\{M_1, M_2, \dots, M_n\}$. Since we cannot automatically guarantee that our domain descriptions secure that each M_i ($[1 \leq i \leq n]$) denotes disjoint sets of entities we must prove it ■

Example 22: Materials

To illustrate the concept of materials we describe waterways (river, canals, lakes, the open sea) along links as links with material of type water.

- 13 Links may contain material.
- 14 That material is water, *W*.

type

- 14 *W*

value

- 13 $\text{obs_material}: L \rightarrow W$
- 13 **pre:** $\text{obs_material}(l) \equiv \text{has_material}(h)$ ■

1.5 Endurants: Analysis & Description of Internal Qualities

We remind the reader that internal qualities cover *unique Identifiers* (Sect. 1.5.1), *mereology* (Sect. 1.5.2) and *attributes* (Sect. 1.5.3).

1.5.1 Unique Identifiers

We introduce a notion of unique identification of parts and components. We assume (i) that all parts and components, *p*, of any domain *P*, have *unique identifiers*, (ii) that *unique identifiers* (of parts and components $p:P$) are *abstract values* (of the *unique identifier* sort *PI* of parts $p:P$), (iii) such that distinct part

or component sorts, P_i and P_j , have distinctly named *unique identifier* sorts, say PI_i and PI_j , (iv) that all $\pi_i:PI_i$ and $\pi_j:PI_j$ are distinct, and (v) that the observer function **uid_P** applied to p yields the unique identifier, $\pi:PI$, of p . The description language function **type_name** applies to unique identifiers, $p_{ui}:P_{UI}$, and yield the name of the type, P , of the parts having unique identifiers of type P_{UI} .

Representation of Unique Identifiers: Unique identifiers are abstractions. When we endow two parts (say of the same sort) with distinct unique identifiers then we are simply saying that these two parts are distinct. We are not assuming anything about how these identifiers otherwise come about.

Domain Description Prompt 5 *observe_unique_identifier*: We can therefore apply the **domain description prompt**:

- *observe_unique_identifier*

to parts $p:P$ resulting in the analyser writing down the unique identifier type and observer domain description text according to the following schema:

5. *observe_unique_identifier* Observer schema

Narration:

- [s] ... narrative text on unique identifier sort PI ...
- [u] ... narrative text on unique identifier observer **uid_P** ...
- [a] ... axiom on uniqueness of unique identifiers ...

Formalisation:

type

- [s] PI

value

- [u] **uid_P**: $P \rightarrow PI$

axiom [Disjointness of Domain Identifier Types]

- [a] $\mathcal{A}: \mathcal{U}(PI, PI_i, PI_j, \dots, PI_k)$

Example 23: Unique Identifiers

- 15 We assign unique identifiers to all parts.
- 16 By a road identifier we shall mean a link or a hub identifier.
- 17 By a vehicle identifier we shall mean a bus or an automobile identifier.
- 18 Unique identifiers uniquely identify all parts.
 - a All hubs have distinct [unique] identifiers.
 - b All links have distinct identifiers.
 - c All bus companies have distinct identifiers.
 - d All buses of all bus companies have distinct identifiers.
 - e All automobiles have distinct identifiers.
 - f All parts have distinct identifiers.

type

15 $H_{UI}, L_{UI}, BC_{UI}, B_{UI}, A_{UI}$

16 $R_{UI} = H_{UI} \mid L_{UI}$

17 $V_{UI} = B_{UI} \mid A_{UI}$

value

18a **uid_H**: $H \rightarrow H_{UI}$

18b **uid_L**: $H \rightarrow L_{UI}$

18c **uid_{BC}**: $H \rightarrow BC_{UI}$

18d **uid_B**: $H \rightarrow B_{UI}$

18e uid_A: H → A_UI

Section 1.9.1 on Page 57 presents some auxiliary functions related to unique identifiers ■

We ascribe, in principle, unique identifiers to all parts whether natural or artefactual, and to all components. We find, from our many experiments, cf. the *Universes of Discourse* example, Page 9, that we really focus on those domain entities which are artefactual endurants and their behavioural “counterparts”.

1.5.2 Mereology

Mereology is the study and knowledge of parts and part relations. Mereology, as a logical/philosophical discipline, can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [104, 58].

Part Relations:

Which are the relations that can be relevant for part-hood? There are basically two relations: (i) a physical one, and (ii) a conceptual one.

(i) Physically two or more parts may be topologically either adjacent to one another, like rails of a line, or within a part, like links and hubs of a road net.

(ii) Conceptually some parts, like automobiles, “belong” to an embedding part, like to an automobile club, or are registered in the local department of vehicles.

Part Mereology: Types and Functions

Analysis Prompt 21 *has_mereology*: To discover necessary, sufficient and pleasing “mereology-hoods” the analyser can be said to endow a truth value, **true**, to the **domain analysis prompt**:

- *has_mereology*

When the domain analyser decides that some parts are related in a specifically enunciated mereology, the analyser has to decide on suitable *mereology types* and *mereology observers* (i.e., part relations).

- 19 We may, to illustration, define a **mereology type** of a part $p:P$ as a triplet type expression over set of unique [part] identifiers.
- 20 There is the identification of all those part types $P_{i_1}, P_{i_2}, \dots, P_{i_m}$ where at least one of whose properties "is_of_interest" to parts $p:P$.
- 21 There is the identification of all those part types $P_{io_1}, P_{io_2}, \dots, P_{io_n}$ where at least one of whose properties "is_of_interest" to parts $p:P$ and vice-versa.
- 22 There is the identification of all those part types $P_{o_1}, P_{o_2}, \dots, P_{o_o}$ for whom properties of $p:P$ "is_of_interest" to parts of types $P_{o_1}, P_{o_2}, \dots, P_{o_o}$.
- 23 The the mereology triplet sets of unique identifiers are disjoint and are all unique identifiers of the universe of discourse.

The three part mereology is just a suggestion. As it is formulated here we mean the three ‘sets’ to be disjoint. Other forms of expressing a mereology should be considered for the particular domain and for the particular parts of that domain. We leave out further characterisation of the seemingly vague notion "is_of_interest".

type

- 20 $iPI = iPI1 \mid iPI2 \mid \dots \mid iPI_m$
- 21 $ioPI = ioPI1 \mid ioPI2 \mid \dots \mid ioPI_n$
- 22 $oPI = oPI1 \mid oPI2 \mid \dots \mid oPI_o$

19 $MT = iPI\text{-set} \times ioPI\text{-set} \times oPI\text{-set}$
axiom
 23 $\forall (iset,ioiset,oset):MT \bullet$
 23 $\mathbf{card} iset + \mathbf{card} ioiset + \mathbf{card} oset = \mathbf{card} \cup\{iset,ioiset,oset\}$
 23 $\cup\{iset,ioiset,oset\} \subseteq \mathbf{unique_identifiers}(uod)$
value
 23 $\mathbf{unique_identifiers}: P \rightarrow UI\text{-set}$
 23 $\mathbf{unique_identifiers}(p) \equiv \dots$

Domain Description Prompt 6 *observe_mereology*: *If has_mereology(p) holds for parts p of type P, then the analyser can apply the domain description prompt:*

- *observe_mereology*

to parts of that type and write down the mereology types and observer domain description text according to the following schema:

6. observe_mereology Observer schema

Narration:
 [t] ... narrative text on mereology type ...
 [m] ... narrative text on mereology observer ...
 [a] ... narrative text on mereology type constraints ...

Formalisation:
type
 [t] MT^{34}
value
 [m] $\mathbf{obs_mereo_P}: P \rightarrow MT$
axiom [Well-formedness of Domain Mereologies]
 [a] $\mathcal{A}: \mathcal{A}(MT)$

$\mathcal{A}(MT)$ is a predicate over possibly all unique identifier types of the domain description. To write down the concrete type definition for MT requires a bit of analysis and thinking. *has_mereology* is a prerequisite prompt for *observe_mereology* ■

Example 24: Mereology

24 The mereology of hubs is a pair: (i) the set of all bus and automobile identifiers³⁵, and (ii) the set of unique identifiers of the links that it is connected to and the set of all unique identifiers of all vehicle (buses and private automobiles).³⁶.

25 The mereology of links is a pair: (i) the set of all bus and automobile identifiers, and (ii) the set of the two distinct hubs they are connected to.

26 The mereology of a bus company is a set the unique identifiers of the buses operated by that company.

27 The mereology of a bus is a pair: (i) the set of the one single unique identifier of the bus company it is operating for, and (ii) the unique identifiers of all links and hubs³⁷.

28 The mereology of an automobile is the set of the unique identifiers of all links and hubs³⁸.

type
 24 $H_Mer = V_UI\text{-set} \times L_UI\text{-set}$
 24 **axiom** $\forall (vuis,luis):H_Mer \bullet luis \subseteq l_{uis} \wedge vuis = v_{uis}$
 25 $L_Mer = V_UI\text{-set} \times H_UI\text{-set}$
 25 **axiom** $\forall (vuis,huis):L_Mer \bullet$
 25 $vuis = v_{uis} \wedge huis \subseteq h_{uis} \wedge \mathbf{card}huis = 2$
 26 $BC_Mer = B_UI\text{-set}$
 26 **axiom** $\forall buis:H_Mer \bullet buis = b_{uis}$

³⁴ The mereology descriptor, MT will be referred to in the sequel.

```

27 B_Mer = BC_UI × R_UI-set
27   axiom  $\forall (bc\_ui, ruis): H\_Mer \bullet bc\_ui \in bc\_uis \wedge ruis = r\_uis$ 
28 A_Mer = R_UI-set
28   axiom  $\forall ruis: A\_Mer \bullet ruis = r\_uis$ 
value
24 mereo_H: H  $\rightarrow$  H_Mer
25 mereo_L: L  $\rightarrow$  L_Mer
26 mereo_BC: BC  $\rightarrow$  BC_Mer
27 mereo_B: B  $\rightarrow$  B_Mer
28 mereo_A: A  $\rightarrow$  A_Mer

```

We can express some additional axioms, in this case for relations between hubs and links:

```

29 If hub,  $h$ , and link,  $l$ , are in the same road net,
30 and if hub  $h$  connects to link  $l$  then link  $l$  connects to hub  $h$ .

```

```

axiom
29  $\forall h: H, l: L \bullet h \in hs \wedge l \in ls \Rightarrow$ 
   let  $(\_, luis) = mereo\_H(h)$ ,  $(\_, huis) = mereo\_L(l)$ 
30   in  $uid\_L(l) \in luis \Rightarrow uid\_H(h) \in huis$  end

```

More mereology axioms need be expressed – but we leave, to the reader, to narrate and formalise those ■

Formulation of Mereologies:

The `observe_mereology` domain descriptor, Page 27, may give the impression that the mereo type MT can be described “at the point of issue” of the `observe_mereology` prompt. Since the MT type expression may, in general, depend on any part sort the mereo type MT can, for some domains, “first” be described when all part sorts have been dealt with. In [61] we present a model of one form of evaluation of the TripTych analysis and description prompts, see also Sect. 1.10.2 on Page 67.

Some Modelling Observations:

It is, in principle, possible to find examples of mereologies of natural parts: rivers: their confluence, lakes and oceans; and geography: mountain ranges, flat lands, etc. But in our experimental case studies, cf. Example on Page 9, we have found no really interesting such cases. All our experimental case studies appears to focus on the mereology of artefacts. And, finally, in modelling humans, we find that their mereology encompass all other humans and all artefacts! Humans cannot be tamed to refrain from interacting with everyone and everything.

Some domain models may emphasize *physical mereologies* based on spatial relations, others may emphasize *conceptual mereologies* based on logical “connections”.

1.5.3 Attributes

To recall: there are three sets of **internal qualities**: unique part identifiers, part mereology and attributes. Unique part identifiers and part mereology are rather definite kinds of internal enduring qualities. Part attributes form more “free-wheeling” sets of **internal qualities**.

Technical Issues:

We divide Sect. 1.5.3 into two subsections: *technical issues*, the present one, and *modelling issues*, Sect. 1.5.3.

Inseparability of Attributes from Parts and Materials:

Parts and materials are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), are physical, tangible, in the sense of being spatial [or being abstractions, i.e., concepts, of spatial endurants], attributes are intangible: cannot normally be touched³⁹, or seen⁴⁰, but can be objectively measured⁴¹. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective “attributes”: feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. We equate all endurants which, besides possible type of unique identifiers (i.e., excepting materials) and possible type of mereologies (i.e., excepting components and materials), have the same types of attributes, with one sort. Thus removing a quality from an endurant makes no sense: the endurant of that type either becomes an endurant of another type or ceases to exist (i.e., becomes a non-entity)!

Attribute Quality and Attribute Value: We distinguish between an attribute (as a logical proposition, of a name, i.e.) type, and an attribute value, as a value in some value space.

Analysis Prompt 22 *attribute types*: One can calculate the set of attribute types of parts and materials with the following **domain analysis prompt**:

- *attribute_types*

Thus for a part p we may have $attribute_types(p) = \{A_1, A_2, \dots, A_m\}$.

Whether by $attribute_types(p)$ we mean the names of the types $\{A_1, A_2, \dots, A_m\}$ for example $\{\eta A_1, \eta A_2, \dots, \eta A_m\}$ where η is some meta-function which applies to a type and yields its name, or we mean the [full] types themselves, i.e., some possibly infinite, suitably structured set of values (of that type), we shall here leave open!

Attribute Types and Functions:

Let us recall that attributes cover qualities other than unique identifiers and mereology. Let us then consider that parts and materials have one or more attributes. These attributes are qualities which help characterise “what it means” to be a part or a material. Note that we expect every part and material to have at least one attribute. The question is now, in general, how many and, particularly, which.

Domain Description Prompt 7 *observe_attributes*: The domain analyser experiments, thinks and reflects about part attributes. That process is initiated by the **domain description prompt**:

- *observe_attributes*.

The result of that **domain description prompt** is that the domain analyser cum describer writes down the attribute (sorts or) types and observers domain description text according to the following schema:

7. *observe_attributes* Observer schema

Narration:

[t]	... narrative text on attribute sorts ...
[o]	... narrative text on attribute sort observers ...
[p]	... narrative text on attribute sort proof obligations ...

³⁹ One can see the red colour of a wall, but one touches the wall.

⁴⁰ One cannot see electric current, and one may touch an electric wire, but only if it conducts high voltage can one know that it is indeed an electric wire.

⁴¹ That is, we restrict our domain analysis with respect to attributes to such quantities which are observable, say by mechanical, electrical or chemical instruments. Once objective measurements can be made of human feelings, beauty, and other, we may wish to include these “attributes” in our domain descriptions.

Formalisation:

```

type
[t]  $A_i$  [ $1 \leq i \leq n$ ]
value
[o] attr $_A_i$ :  $P \rightarrow A_i$   $i:[1..n]$ 
proof obligation [Disjointness of Attribute Types]
[p]  $\mathcal{P}\mathcal{O}$ : let  $P$  be any part sort in [the domain description]
[p] let  $a:(A_1|A_2|\dots|A_n)$  in  $\text{is}_{A_i}(a) \neq \text{is}_{A_j}(a)$  end end [ $i \neq j, i,j:[1..n]$ ]

```

The $\text{is}_{A_j}(e)$ is defined by $A_i, i:[1..n]$.

The **type** (or rather sort) definitions: A_1, A_2, \dots, A_n , inform us that the domain analyser has decided to focus on the distinctly named A_1, A_2, \dots, A_n attributes.⁴² And the **value** clauses $\text{attr}_{A_1}:P \rightarrow A_1, \text{attr}_{A_2}:P \rightarrow A_2, \dots, \text{attr}_{A_n}:P \rightarrow A_n$ are then “automatically” given: if a part, $p:P$, has an attribute A_i then there is postulated, “by definition” [eureka] an attribute observer function $\text{attr}_{A_i}:P \rightarrow A_i$ etcetera ■

We cannot automatically, that is, syntactically, guarantee that our domain descriptions secure that the various attribute types for a part sort denote disjoint sets of values. Therefore we must prove it.

Attribute Categories: Michael A. Jackson [157] has suggested a hierarchy of attribute categories: static or dynamic values – and within the dynamic value category: inert values or reactive values or active values – and within the dynamic active value category: autonomous values or biddable values or programmable values. We now review these attribute value types. The review is based on [157, M.A. Jackson]. *Part attributes* are either constant or varying, i.e., **static** or **dynamic** attributes.

Attribute Category: 1 By a **static attribute**, $a:A$, $\text{is_static_attribute}(a)$, we shall understand an attribute whose values are constants, i.e., cannot change.

Attribute Category: 2 By a **dynamic attribute**, $a:A$, $\text{is_dynamic_attribute}(a)$, we shall understand an attribute whose values are variable, i.e., can change. Dynamic attributes are either *inert*, *reactive* or *active* attributes.

Attribute Category: 3 By an **inert attribute**, $a:A$, $\text{is_inert_attribute}(a)$, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe new values.

Attribute Category: 4 By a **reactive attribute**, $a:A$, $\text{is_reactive_attribute}(a)$, we shall understand a dynamic attribute whose values, if they vary, change in response to external stimuli, where these stimuli come from outside the domain of interest.

Attribute Category: 5 By an **active attribute**, $a:A$, $\text{is_active_attribute}(a)$, we shall understand a dynamic attribute whose values change (also) of its own volition. Active attributes are either *autonomous*, *biddable* or *programmable* attributes.

Attribute Category: 6 By an **autonomous attribute**, $a:A$, $\text{is_autonomous_attribute}(a)$, we shall understand a dynamic active attribute whose values change only “on their own volition”. The values of an autonomous attributes are a “law unto themselves and their surroundings”.

Attribute Category: 7 By a **biddable attribute**, $a:A$, $\text{is_biddable_attribute}(a)$ we shall understand a dynamic active attribute whose values *are prescribed but may fail to be observed as such*.

Attribute Category: 8 By a **programmable attribute**, $a:A$, $\text{is_programmable_attribute}(a)$, we shall understand a dynamic active attribute whose values can be prescribed.

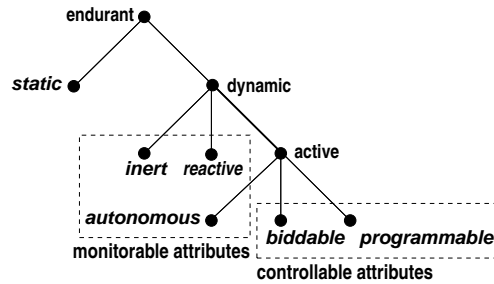


Fig. 1.6. Attribute Value Ontology

Figure 1.6 on the next page captures an attribute value ontology.

We treat part attributes, sort by sort. **Example 25: Attributes.** We show just a few attributes:

- 31 There is a hub state. It is a set of pairs, (l_f, l_r) of link identifiers, where these link identifiers are in the mereology of the hub. The meaning of the hub state, in which, e.g., (l_f, l_r) is an element, is that the hub is open, “green”, for traffic from link l_f to link l_r . If a hub state is empty then the hub is closed, i.e., “red” for traffic from any connected links to any other connected links.
- 32 There is a hub state space. It is a set of hub states. The meaning of the hub state space is that its states are all those the hub can attain. The current hub state must be in its state space.
- 33 Since we can think rationally about it, it can be described, hence it can model, as an attribute of hubs a history of its traffic: the recording, per unique bus and automobile identifier, of the time ordered presence in the hub of these vehicles.
- 34 The link identifiers of hub states must be in the set, $l_{ui}s$, of the road net's link identifiers.

type

31 $H\Sigma = (L_UI \times L_UI)\text{-set}$

axiom

31 $\forall h:H \cdot \text{obs_H}\Sigma(h) \in \text{obs_H}\Omega(h)$

type

32 $H\Omega = H\Sigma\text{-set}$

33 $H_Traffic$

33 $H_Traffic = (A_UI|B_UI) \rightarrow_{\mathcal{H}} (\mathcal{T} \times VPos)^*$

axiom

33 $\forall ht:H_Traffic, ui:(A_UI|B_UI) \cdot$

33 $ui \in \text{dom } ht \Rightarrow \text{time_ordered}(ht(ui))$

value

31 $\text{attr_H}\Sigma: H \rightarrow H\Sigma$

32 $\text{attr_H}\Omega: H \rightarrow H\Omega$

33 $\text{attr_H_Traffic}: H \rightarrow H_Traffic$

axiom

34 $\forall h:H \cdot h \in hs \Rightarrow$

34 **let** $h\sigma = \text{attr_H}\Sigma(h)$ **in**

34 $\forall (l_{ui}i, l_{ui}i'): (L_UI \times L_UI) \cdot$

34 $(l_{ui}i, l_{ui}i') \in h\sigma$

34 $\Rightarrow \{l_{ui}i, l_{ui}i'\} \subseteq l_{ui}s$ **end**

value

33 $\text{time_ordered}: \mathcal{T}^* \rightarrow \text{Bool}$

⁴² The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena.

33 `time_ordered(tvpl) ≡ ...`

Attributes for remaining sorts are shown in Sect. 1.9.2 on Page 58.

Calculating Attributes:

- 35 Given a part p we can *meta-linguistically*⁴³ calculate names for its static attributes.
 36 Given a part p we can *meta-linguistically* calculate name for its monitorable attributes attributes.
 37 Given a part p we can *meta-linguistically* calculate name for its monitorable and controllable attributes.
 38 Given a part p we can *meta-linguistically* calculate names for its controllable attributes.
 39 These three sets make up all the attributes of part p .

The type names nSA , nMA $nMCA$, nCA designate sets of names.

value

- 35 `stat_attr_typs: P → nSA-set`
 36 `mon_attr_typs: P → nMA-set`
 37 `mon_ctrl_attr_typs: P → nMCA-set`
 38 `ctrl_attr_typs: P → nCA-set`

axiom

- 39 $\forall p:P \cdot$
 35 **let** `stat_nms = stat_attr_typs(p),`
 36 `mon_nms = mon_attr_typs(p),`
 37 `mon_ctrl_nms = mon_ctrl_attr_typs(p),`
 38 `ctrl_nms = mon_ctrl_typs(p) in`
 39 **card** `stat_nms + card mon_nms + card mon_ctrl_nms + card ctrl_nms`
 39 **= card**(`stat_nms ∪ mon_nms ∪ mon_ctrl_nms ∪ ctrl_nms`) **end**

The above formulas are indicative, like mathematical formulas, they are not computable.

- 40 Given a part p we can *meta-linguistically* calculate its static attribute values.
 41 Given a part p we can *meta-linguistically* calculate its controllable, i.e., programmable attribute values.

Et cetera for monitorable and monitorable & controllable attribute values.

The type names $sa1$, ..., cac refer to the types denoted by the corresponding types name $nsa1$, ..., $ncac$.

value

- 40 `stat_attr_vals: P → SA1×SA2×...×SAs`
 40 `stat_attr_vals(p) ≡ let {nsa1,nsa2,...,nsas}`
 40 **= stat_attr_typs(p) in (attr_sa1(p),attr_sa2(p),...,attr_sas(p)) end**
 41 `ctrl_attr_vals: P → CA1×CA2×...×CAc`
 41 `ctrl_attr_vals(p) ≡ let {nca1,nca2,...,ncac}`
 41 **= ctrl_attr_typs(p) in (attr_ca1(p),attr_ca2(p),...,attr_cac(p)) end**

The “ordering” of type values, $(attr_sa1(p), \dots, attr_sas(p))$, respectively $(attr_ca1(p), \dots, attr_cac(p))$, is arbitrary.

⁴³ By using the term *meta-linguistically* here we shall indicate that we go outside what is computable – and thus appeal to the reader’s forbearance.

Basic Principles for Ascribing Attributes:

Section 1.5.3 dealt with technical issues of expressing attributes. This section will indicate some modelling principles.

Natural Parts: are subject to laws of physics. So basic attributes focus on physical (including chemical) properties. These attributes cover the full spectrum of attribute categories outlined in Sect. 1.5.3.

Materials: are subject to laws of physics. So basic attributes focus on physical, especially chemical properties. These attributes cover the full spectrum of attribute categories outlined in Sect. 1.5.3.

The next paragraphs, **living species**, **animate entities** and **humans**, reflect Sørlander's Philosophy [245, pp 14–182].



Causality of Purpose: If there is to be *the possibility of language and meaning* then there must exist primary entities which are *not entirely encapsulated within the physical conditions*; that they are stable and can influence one another. This is only possible if such primary entities are subject to a *supplementary causality directed at the future: a causality of purpose*.

Living Species: These primary entities are here called *living species*. What can be deduced about them? They are characterised by *causality of purpose*: they *have some form they can be developed to reach*; and which *they must be causally determined to maintain*; this development and maintenance must further in *an exchange of matter with an environment*. It must be possible that living species occur in one of two forms: one form which is characterised by *development, form and exchange*, and another form which, additionally, can be characterised by the ability to *purposeful movements*. The first we call *plants*, the second we call *animals*.

Animate Entities: For an animal to purposefully move around there must be “additional conditions” for such self-movements to be in accordance with the principle of causality: they must have *sensory organs* sensing among others the immediate purpose of its movement; they must have *means of motion* so that it can move; and they must have *instincts, incentives and feelings* as causal conditions that what it senses can drive it to movements. And all of this in accordance with the laws of physics.

Animals: To possess these three kinds of “additional conditions”, must be built from special units which have an inner relation to their function as a whole; Their *purposefulness* must be built into their physical building units, that is, as we can now say, their *genomes*. That is, animals are built from genomes which give them the *inner determination* to such building blocks for *instincts, incentives and feelings*. Similar kinds of deduction can be carried out with respect to plants. Transcendentally one can deduce basic principles of evolution but not its details.

Humans: Consciousness and Learning: The existence of animals is a necessary condition for there being language and meaning in any world. That there can be *language* means that animals are capable of *developing language*. And this must presuppose that animals can *learn from their experience*. To learn implies that animals can *feel* pleasure and distaste and can *learn*. One can therefore deduce that animals must possess such building blocks whose inner determination is a basis for learning and consciousness.

Language: Animals with higher social interaction uses *signs*, eventually developing a *language*. These languages adhere to the same system of defined concepts which are a prerequisite for any description of any world: namely the system that philosophy lays bare from a basis of transcendental deductions and the *principle of contradiction* and its *implicit meaning theory*. A *human* is an animal which has a *language*.

Knowledge: Humans must be *conscious* of having *knowledge* of its concrete situation, and as such that human can have knowledge about what he feels and eventually that human can know whether what he feels is true or false. Consequently a *human can describe his situation correctly*.

Responsibility: In this way one can deduce that humans can thus have *memory* and hence can have *responsibility, be responsible*. Further deductions lead us into *ethics*.

We shall not develop the theme of *living species: plants and animals*, thus excluding, most notably *humans*, much further in this chapter. We claim that the present chapter, due to its foundation in Kai Sørlander’s Philosophy, provides a firm foundation with which we, or others, can further develop this theme: *analysis & description of living species*.

Intentionality: *Intentionality is a philosophical concept and is defined by the Stanford Encyclopedia of Philosophy⁴⁴ as “the power of minds to be about, to represent, or to stand for, things, properties and states of affairs.”*

Definition 16 Intentional Pull: Two or more artefactual parts of different sorts, but with overlapping sets of intents may exert an *intentional “pull”* on one another ■

This *intentional “pull”* may take many forms. Let $p_x : X$ and $p_y : Y$ be two parts of *different sorts* (X, Y), and with *common intent*, ι . *Manifestations* of these, their common intent must somehow be *subject to constraints*, and these must be *expressed predicatively*.

Example 26: Intentional Pull

We illustrate the concept of intentional “pull”:

42 *automobiles include the intent of ‘transport’,*
43 *and so do hubs and links.*

42 **attr_Intent:** $A \rightarrow (\text{'transport' | ...})\text{-set}$
43 **attr_Intent:** $H \rightarrow (\text{'transport' | ...})\text{-set}$
43 **attr_Intent:** $L \rightarrow (\text{'transport' | ...})\text{-set}$

Manifestations of ‘transport’ is reflected in *automobiles* having the automobile position attribute, APos, Item 125 Pg. 60, *hubs* having the *hub traffic* attribute, H_Traffic, Item 33 Pg. 31, and in *links* having the *link traffic* attribute, L_Traffic, Item 117 Pg. 58.

44 Seen from the point of view of an automobile there is its own traffic history, A_Hist, which is a (time ordered) sequence of timed automobile’s positions;
45 seen from the point of view of a hub there is its own traffic history, H_Traffic Item 33 Pg. 31, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions; and
46 seen from the point of view of a link there is its own traffic history, L_Traffic Item 117 Pg. 58, which is a (time ordered) sequence of timed maps from automobile identities into automobile positions.

The *intentional “pull”* of these manifestations is this:

47 The union, i.e. proper merge of all automobile traffic histories, AllATH, must now be identical to the same proper merge of all hub, AllHTH, and all link traffic histories, AllLTH.

type

44 $A_Hi = (\mathbb{T} \times APos)^*$
33 $H_Trf = A_UI \rightarrow_m (\mathbb{T} \times APos)^*$
117 $L_Trf = A_UI \rightarrow_m (\mathbb{T} \times APos)^*$
47 $AllATH = \mathbb{T} \rightarrow_m (AUI \rightarrow_m APos)$
47 $AllHTH = \mathbb{T} \rightarrow_m (AUI \rightarrow_m APos)$
47 $AllLTH = \mathbb{T} \rightarrow_m (AUI \rightarrow_m APos)$

axiom

47 **let** allA = mrg_AllATH({(a, attr_A_Hi(a)) | a: A • a ∈ as}),
47 allH = mrg_AllHTH({attr_H_Trf(h) | h: H • h ∈ hs}),
47 allL = mrg_AllLTH({attr_L_Trf(l) | l: L • l ∈ ls}) **in**
47 allA = mrg_HLT(allH, allL) **end**

⁴⁴ Jacob, P. (Aug 31, 2010). *Intentionality*. Stanford Encyclopedia of Philosophy (<https://seop.illc.uva.nl/entries/intentionality/>) October 15, 2014, retrieved April 3, 2018.

We leave the definition of the four merge functions to the reader !

Discussion: We endow each automobile with its history of timed positions and each hub and link with their histories of timed automobile positions. These histories are facts ! They are not something that is laboriously recorded, where such recordings may be imprecise or cumbersome⁴⁵. The facts are there, so we can (but may not necessarily) talk about these histories as facts. It is in that sense that the purpose (‘transport’) for which man let automobiles, hubs and link be made with their ‘transport’ intent are subject to an *intentional* “pull”. *It can be no other way: if automobiles “record” their history, then hubs and links must together “record” identically the same history!*

Artefacts: Humans create artefacts – for a reason, to serve a purpose, that is, with **intent**. Artefacts are like parts. They satisfy the laws of physics – and serve a *purpose*, fulfill an *intent*.

Assignment of Attributes: So what can we deduce from the above, a little more than two pages ?

The attributes of **natural parts** and **natural materials** are generally of such concrete types – expressible as some **real** with a dimension⁴⁶ of the International System of Units: <https://physics.nist.gov/cuu/Units/units.html>. Attribute values usually enter *differential equations* and *integrals*, that is, classical calculus.

The attributes of **humans**, besides those of parts, significantly includes one of a usually non-empty set of *intents*. In directing the creation of artefacts humans create these with an intent.

Example 27: Intentional Pull

These are examples of human intents: they create roads and automobiles with the intent of transport, they create houses with the intents of living, offices, production, etc., and they create pipelines with the intent of oil or gas transport ■

Human attribute values usually enter into *modal logic* expressions.

Artefacts, including Man-made Materials: Artefacts, besides those of parts, significantly includes a usually singleton set of *intents*.

Example 28: Intents

Roads and automobiles possess the intent of transport; houses possess either one of the intents of living, offices, production; and pipelines possess the intent of oil or gas transport.

Artefact attribute values usually enter into *mathematical logic* expressions.

We leave it to the reader to formulate attribute assignment principles for plants and non-human animals.

1.5.4 Some Axioms and Proof Obligations

To remind you, an **axiom** – in the *context* of domain analysis & description – means a logical expression, usually a predicate, that constrains the types and values, including unique identifiers and mereologies of domain models. Axioms, together with the sort, including type definitions, and the unique identifier, mereology and attribute observer functions, define the domain value spaces. We refer to axioms in Item [a] of domain description prompts of *unique identifiers*: 5 on Page 25 and of *mereologies*: 6 on Page 27.

Another reminder: a **proof obligation** – in the *context* of domain analysis & description – means a logical expression that predicates relations between the types and values, including unique identifiers, mereologies and attributes of domain models, where these predicates must be shown, i.e., proved, to hold. Proof obligations supplement axioms. We refer to proof obligations in Item [p] of domain description prompts about *endurant sorts*: 1 on Page 19, about *components sorts*: 3 on Page 23, about *materials sorts*: 4 on Page 24, and about *attribute types*: 7 on Page 30.

The difference between expressing axioms and expressing proof obligations is this:

⁴⁶ Basic units are *meter, kilogram, second, Ampere, Kelvin, mole, and candela*. Some derived units are: *Newton: $kg \times m \times s^{-2}$, Weber: $kg \times m^2 \times s^{-2} \times A^{-1}$* , etc.

- **We use axioms** when our formula cannot otherwise express it simply, but when physical or other *properties of the domain*⁴⁷ dictates property consistency.
- **We use proof obligations** where necessary constraints are not necessarily physically impossible.
- **Proof obligations** finally arise in the transition from endurants to perdurants where endurant axioms become properties that must be proved to hold.

When considering *endurants* we interpret these as stable, i.e., that although they may have, for example, programmable attributes, when we observe them, we observe them at any one moment, but *we do not consider them over a time*. That is what we turn to next: *perdurants*. When considering a part with, for example, a programmable attribute, at two different instances of time we expect the particular programmable attribute to enjoy any expressed well-formedness properties. We shall, in Sect. 1.8, see how these programmable attributes re-occur as explicit behaviour parameters, “programmed” to possibly new values passed on to recursive invocations of the same behaviour. If well-formedness axioms were expressed for the part on which the behaviour is based, then a *proof obligation* arises, one that must show that new values of the programmed attribute satisfies the part attribute axiom. This is, but one relation between *axioms* and *proof obligations*. We refer to remarks made in the bullet (•) named **Biddable Access** Page 49.

1.5.5 Discussion of Endurants

Domain descriptions are, as we have already shown, formulated, both informally and formally, by means of abstract types, that is, by sorts for which no concrete models are usually given. Sorts are made to denote possibly empty, possibly infinite, rarely singleton, sets of entities on the basis of the qualities defined for these sorts, whether external or internal. By **junk** we shall understand that the domain description unintentionally denotes undesired entities. By **confusion** we shall understand that the domain description unintentionally have two or more identifications of the same entity or type. The question is *can we formulate a [formal] domain description such that it does not denote junk or confusion*? The short answer to this is no! So, since one naturally wishes “no junk, no confusion” what does one do? The answer to that is *one proceeds with great care!*

1.6 A Transcendental Deduction

1.6.1 An Explanation

It should be clear to the reader that in domain analysis & description we are reflecting on a number of philosophical issues. First and foremost on those of *epistemology*, especially *ontology*. In this section on a sub-field of epistemology, namely that of a number of issues of *transcendental* nature, we refer to [152, Oxford Companion to Philosophy, pp 878–880] [6, The Cambridge Dictionary of Philosophy, pp 807–810] [100, The Blackwell Dictionary of Philosophy, pp 54–55 (1998)].

Definition 17 Transcendental: By **transcendental** we shall understand the philosophical notion: **the a priori or intuitive basis of knowledge, independent of experience** ■

A priori knowledge or intuition is central: By *a priori* we mean that it not only precedes, but also determines rational thought.

Definition 18 Transcendental Deduction: By a **transcendental deduction** we shall understand the philosophical notion: **a transcendental “conversion” of one kind of knowledge into a seemingly different kind of knowledge** ■

⁴⁷ – examples of such properties are: (i) topologies of the domain makes certain compositions of parts physically impossible, and (ii) conservation laws of the domain usually dictates that endurants cannot suddenly arise out of nothing.

Example 29: Some Transcendental Deductions

We give some intuitive examples of transcendental deductions. They are from the “domain” of programming languages. There is the syntax of a programming language, and there are the programs that supposedly adhere to this syntax. Given that, the following are now transcendental deductions. The software tool, a syntax checker, that takes a program and checks whether it satisfies the syntax, including the statically decidable context conditions, i.e., the statics semantics – that tool is one of several forms of transcendental deductions; The software tools, an automatic theorem prover⁴⁸ and a model checker, for example SPIN [151], that takes a program and some theorem, respectively a Promela statement, and proves, respectively checks, the program correct with respect the theorem, or the statement. A compiler and an interpreter for any programming language. Yes, indeed, any abstract interpretation [114, 96] reflects a transcendental deduction: First these examples show that there are many transcendental deductions. Secondly they show that there is no single-most preferred transcendental deduction.

A transcendental deduction, crudely speaking, is just any abstraction that can be “linked” to another, not by logical necessity, but by logical (and philosophical) possibility !

Definition 19 Transcendentality: By **transcendentality** we shall here mean the philosophical notion: the state or condition of being transcendental ■

Example 30: Transcendentality

We can speak of a bus in at least three senses:

- (i) The bus as it is being "maintained, serviced, refueled";
- (ii) the bus as it "speeds" down its route; and
- (iii) the bus as it "appears" (listed) in a bus time table.

The three senses are:

- (i) as an **endurant** (here a part),
- (ii) as a **perdurant** (as we shall see a behaviour), and
- (iii) as an **attribute**⁴⁹ ■

The above example, we claim, reflects *transcendentality* as follows:

- (i) We have knowledge of an endurant (i.e., a part) being an endurant.
- (ii) We are then to assume that the perdurant referred to in (ii) is an aspect of the endurant mentioned in (i) – where perdurants are to be assumed to represent a different kind of knowledge.
- (iii) And, finally, we are to further assume that the attribute mentioned in (iii) is somehow related to both (i) and (ii) – where at least this attribute is to be assumed to represent yet a different kind of knowledge.

In other words: two (i–ii) kinds of different knowledge; that they relate *must indeed* be based on a *a priori knowledge*. Someone claims that they relate ! The two statements (i–ii) are claimed to relate transcendently.⁵⁰

1.6.2 Classical Transcendental Deductions

We present a few of the transcendental deductions of [245, Kai Sørlander: *Introduction to The Philosophy*, 2016]

⁵⁰ – the attribute statement was “thrown” in “for good measure”, i.e., to highlight the issue !

Space:

[245, pp 154] “The two relations asymmetric and symmetric, by a transcendental deduction, can be given an interpretation: The relation (spatial) *direction* is asymmetric; and the relation (spatial) *distance* is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation *in-between*. Hence we must conclude that *primary entities exist in space*. Space is therefore an unavoidable characteristic of any possible world”

Time:

[245, pp 159] “Two different states must necessarily be ascribed different incompatible predicates. But how can we ensure so? Only if states stand in an asymmetric relation to one another. This state relation is also transitive. So that is an indispensable property of any world. By a transcendental deduction we say that *primary entities exist in time*. So every possible world must exist in time”

1.6.3 Some Special Notation

The *transcendentality* that we are referring to is one in which we “**translate**” enduring descriptions of parts and their *unique identifiers*, *mereologies* and *attributes* into descriptions of perdurants, i.e., transcendental interpretations of parts as *behaviours*, part mereologies as *channels*, and part attributes as *attribute value accesses*. The *translations* referred to above, *compile* enduring descriptions into RSL⁺Text. We shall therefore first explain some aspects of this translation.

Where in the function definition bodies we enclose some RSL⁺Text, e.g., `rsl+_text`, in $\langle\langle\rangle\rangle$ s, i.e., $\langle\langle$ `rsl+_text` $\rangle\rangle$ we mean that text. Where in the function definition bodies we write $\langle\langle$ `rsl+_text` $\rangle\rangle$ `function_expression` we mean that `rsl+_text` concatenated to the RSL⁺Text emanating from `function_expression`. Where in the function definition bodies we write $\langle\langle\rangle\rangle$ `function_expression` we mean just `rsl+_text` emanating from `function_expression`. That is: $\langle\langle\rangle\rangle$ `function_expression` \equiv `function_expression` and $\langle\langle\rangle\rangle$ $\langle\langle\rangle\rangle$ \equiv $\langle\langle\rangle\rangle$. Where in the function definition bodies we write $\{ \langle\langle$ `f(x)` $\rangle\rangle | x:\text{RSL}^+\text{Text} \}$ we mean the “expansion” of the RSL⁺Text `f(x)`, in arbitrary, linear text order, for appropriate RSL⁺Texts `x`.

1.7 Space and Time

This section is a necessary prelude to our treatment of perdurants.

Following Kai Sørlander’s Philosophy we must accept that space and time are rationally potentially mandated in any domain description. It is, however not always necessary to model space and time. We can talk about space and time; **and** when we do, we must model them.

1.7.1 Space**General:**

Mathematicians and physicists model space in, for example, the form of Hausdorff (or topological) space⁵¹; or a metric space which is a set for which distances between all members of the set are defined; Those distances, taken together, are called a metric on the set; a metric on a space induces topological properties like open and closed sets, which lead to the study of more abstract topological spaces; or Euclidean space, due to *Euclid of Alexandria*.

⁵¹ Armstrong, M. A. (1983) [1979]. Basic Topology. Undergraduate Texts in Mathematics. Springer. ISBN 0-387-90839-0.

Space Motivated Philosophically

Characterisation 9 Indefinite Space: We motivate the concept of indefinite space as follows: [245, pp154] “*The two relations asymmetric and symmetric, by a transcendental deduction, can be given an interpretation: The relation (spatial) direction is asymmetric; and the relation (spatial) distance is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation in-between. Hence we must conclude that primary entities exist in space. Space is therefore an unavoidable characteristic of any possible world*” ■

From the direction and distance relations one can derive *Euclidean Geometry*.

Characterisation 10 Definite Space: By a **definite space** we shall understand a space with a definite metric ■

There is but just one space. It is all around us, from the inner earth to the farthest galaxy. It is not manifest. We can not observe it as we observe a road or a human.

Space Types

The Spatial Value:

48 There is an abstract notion of (definite) SPACE(s) of further unanalysable points; and
49 there is a notion of POINT in SPACE.

type

48 SPACE
49 POINT

Space is not an attribute of endurants. Space is just there. So we do not define an observer, `observe_space`. For us, bound to model mostly artifactual worlds on this earth there is but one space. Although SPACE, as a type, could be thought of as defining more than one space we shall consider these isomorphic!

Spatial Observers

50 A point observer, `observe_POINT`, is a function which applies to physical endurants, e , and yield a point, $\ell : \text{POINT}$.

value

50 `observe_POINT`: $E \rightarrow \text{POINT}$

1.7.2 Time

Concepts of time⁵² continue to fascinate thinkers [252, 124, 179, 202, 207, 208, 209, 210, 211, 212, 224] and [128, Mandrioli et al.].

⁵² Time:

- (i) a moving image of eternity;
 - (ii) the number of the movement in respect of the before and the after;
 - (iii) the life of the soul in movement as it passes from one stage of act or experience to another;
 - (iv) a present of things past: memory, a present of things present: sight, and a present of things future: expectations.
- [6, (i) Plato, (ii) Aristotle, (iii) Plotinus, (iv) Augustine].

Time Motivated Philosophically

Characterisation 11 Indefinite Time: We motivate the abstract notion of time as follows. [245, pp 159] “Two different states must necessarily be ascribed different incompatible predicates. But how can we ensure so? Only if states stand in an asymmetric relation to one another. This state relation is also transitive. So that is an indispensable property of any world. By a transcendental deduction we say that primary entities exist in time. So every possible world must exist in time” ■

Characterisation 12 Definite Time: By a **definite time** we shall understand an abstract representation of time such as for example year, month, day, hour, minute, second, et cetera ■

Example 31: Temporal Notions of Endurants

By temporal notions of endurants we mean time properties of endurants, usually modelled as attributes. Examples are: (i) the time stamped link traffic, cf. Item 117 on Page 58 and (ii) the time stamped hub traffic, cf. Item 33 on Page 31.

Time Values

We shall not be concerned with any representation of time. That is, we leave it to the domain analyser cum describer to choose an own representation [128]. Similarly we shall not be concerned with any representation of time intervals.⁵³

- | | |
|---|---|
| <p>51 So there is an abstract type Time,</p> <p>52 and an abstract type TI: $\mathit{TimeInterval}$.</p> <p>53 There is no Time origin, but there is a “zero” TIme interval.</p> <p>54 One can add (subtract) a time interval to (from) a time and obtain a time.</p> <p>55 One can add and subtract two time intervals and obtain a time interval – with subtraction respecting that the subtrahend is smaller than or equal to the minuend.</p> <p>56 One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.</p> <p>57 One can multiply a time interval with a real and obtain a time interval.</p> | <p>58 One can compare two times and two time intervals.</p> <p>type</p> <p>51 T</p> <p>52 TI</p> <p>value</p> <p>53 $\mathbf{0}:\mathit{TI}$</p> <p>54 $+, -: \mathit{T} \times \mathit{TI} \rightarrow \mathit{T}$</p> <p>55 $+, -: \mathit{TI} \times \mathit{TI} \xrightarrow{\sim} \mathit{TI}$</p> <p>56 $-: \mathit{T} \times \mathit{T} \rightarrow \mathit{TI}$</p> <p>57 $*: \mathit{TI} \times \mathbf{Real} \rightarrow \mathit{TI}$</p> <p>58 $<, \leq, =, \neq, \geq, >: \mathit{T} \times \mathit{T} \rightarrow \mathbf{Bool}$</p> <p>58 $<, \leq, =, \neq, \geq, >: \mathit{TI} \times \mathit{TI} \rightarrow \mathbf{Bool}$</p> <p>axiom</p> <p>54 $\forall t:\mathit{T} \cdot t + \mathbf{0} = t$</p> |
|---|---|

Temporal Observers

59 We define the signature of the meta-physical time observer.

type

59 T

value

59 $\mathbf{record_TIME}(): \mathbf{Unit} \rightarrow \mathit{T}$

⁵³ – but point out, that although a definite time interval may be referred to by number of years, number of days (less than 365), number of hours (less than 24), number of minutes (less than 60) number of seconds (less than 60), et cetera, this is not a time, but a time interval.

The time recorder applies to nothing and yields a time. `record_TIME()` can only occur in action, event and behavioural descriptions.

Models of Time:

Modern models of time, by mathematicians and physicists evolve around spacetime⁵⁴ We shall not be concerned with this notion of time. Models of time related to computing differs from those of mathematicians and physicists in focusing on divergence and convergence, zero (Zenon) time and interleaving time [262] are relevant in studies of real-time, typically distributed computing systems. We shall also not be concerned with this notion of time.

Spatial and Temporal Modelling:

It is not always that we are compelled to endow our domain descriptions with those of spatial and/or temporal properties. In our experimental domain descriptions, for example, [67, 95, 72, 69, 33, 48, 63, 24], we have either found no need to model space and/or time, or we model them explicitly, using slightly different types and observers than presented above.

1.7.3 Whither Attributes ?

Are space and time attributes of endurants ? Of course not ! Space and time surround us. Every endurant is in the one-and-only space we know of. Every endurant is “somewhere” in that space. We represent that ‘somewhere’ by a point in space. Every endurant point can be recorded. And every such recording can be time-stamped.

1.7.4 Whither Entities ?

Are space and time entities ? Of course not ! They are simply abstract concepts that apply to any entity.

1.8 Perdurants

The main transcendental deduction of this chapter is that of associating with each part a behaviour. This section shows the details of this association. A main conjecture of this chapter is this:

Perdurants are understood in terms of a notion of *time* and a notion of *state*. We covered the notion of state in Sect. 1.3.8 on Page 18 and time in Sect. 1.7.2 on Page 39.

1.8.1 States, Actors, Actions, Events and Behaviours: A Preview

Example 32: Constants and States

Constants:

60 Let there be given a universe of discourse, *rs*. It is an example of a state.

From that state we can calculate other states.

61 The set of all hubs, *hs*.

62 The set of all links, *ls*.

⁵⁴ The concept of **Spacetime** was first “announced” by Hermann Minkowski, 1907–08 – based on work by Henri Poincaré, 1905–06, https://en.wikisource.org/wiki/Translation:The_Fundamental_Equations_for_Electromagnetic_Processes_in_Moving_Bodies

- 63 The set of all hubs and links, hls .
 64 The set of all bus companies, bcs .
 65 The set of all buses, bs .
 66 The map from the unique bus company identifiers to the set of all the identifies bus company's buses, $bc_{ui}bs$.
 67 The set of all private automobiles, as .
 68 The set of all parts, ps .

value

- 60 $rts:UoD$ [60]
 61 $hs:H\text{-set} \equiv H\text{-set} \equiv \text{obs_sH}(\text{obs_SH}(\text{obs_RN}(rts)))$
 62 $ls:L\text{-set} \equiv L\text{-set} \equiv \text{obs_sL}(\text{obs_SL}(\text{obs_RN}(rts)))$
 63 $hls:(H|L)\text{-set} \equiv hs \cup ls$
 64 $bcs:BC\text{-set} \equiv \text{obs_BCs}(\text{obs_SBC}(\text{obs_FV}(\text{obs_RN}(rts))))$
 65 $bs:B\text{-set} \equiv \cup \{ \text{obs_Bs}(bc) \mid bc:BC \bullet bc \in bcs \}$
 66 $as:A\text{-set} \equiv \text{obs_BCs}(\text{obs_SBC}(\text{obs_FV}(\text{obs_RN}(rts))))$

Indexed States:

We shall

- 69 index bus companies,
 70 index buses, and
 71 index automobiles

using the unique identifiers of these parts.

type

- 69 BC_{ui}
 70 B_{ui}
 71 A_{ui}

value

- 69 $ibcs:BC_{ui}\text{-set} \equiv$
 69 $\{ bc_{ui} \mid bc:BC, bc:BC_{ui}:BC_{ui} \bullet bc \in bcs \wedge ui = \text{uid_BC}(bc) \}$
 70 $ibs:B_{ui}\text{-set} \equiv$
 70 $\{ b_{ui} \mid b:B, b:B_{ui}:B_{ui} \bullet b \in bs \wedge ui = \text{uid_B}(b) \}$
 71 $ias:A_{ui}\text{-set} \equiv$
 71 $\{ a_{ui} \mid a:A, a:A_{ui}:A_{ui} \bullet a \in as \wedge ui = \text{uid_A}(a) \}$

Actors, Actions, Events, Behaviours and Channels

To us perdurants are further, pragmatically, analysed into *actions*, *events*, and *behaviours*. We shall define these terms below. Common to all of them is that they potentially change a state. Actions and events are here considered atomic perdurants. For behaviours we distinguish between discrete and continuous behaviours.

Time Considerations

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may “occur” during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can be analysed into “constituent” actions, events and “sub-behaviours”. We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal monographs: Specifying Systems [167, Leslie Lamport] and Duration Calculus: A Formal Approach to Real-Time Systems [264, Zhou ChaoChen and Michael Reichhardt Hansen] (and [32, Chapter 15]). For a seminal book on “time in computing” we refer to the eclectic [128, Mandrioli et al., 2012]. And for seminal book on time at the epistemology level we refer to [252, J. van Benthem, 1991].

Actors

Definition 20 Actor: By an **actor** we shall understand something that is capable of initiating and/or **carrying out** actions, events or behaviours ■

The notion of “*carrying out*” will be made clear in this overall section. We shall, in principle, associate an actor with each part⁵⁵. These actors will be described as behaviours. These behaviours evolve around a state. The state is the set of qualities, in particular the dynamic attributes, of the associated parts and/or any possible components or materials of the parts.

Discrete Actions

Definition 21 Discrete Action: By a **discrete action** [258, Wilson and Shpall] we shall understand a foreseeable thing which deliberately and potentially changes a well-formed state, in one step, usually into another, still well-formed state, for which an actor can be made responsible ■

An action is what happens when a function invocation changes, or potentially changes a state.

Discrete Events

Definition 22 Event: By an **event** we shall understand some unforeseen thing, that is, some ‘not-planned-for’ “action”, one which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular domain actor can be made responsible ■

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a *time* or *time interval*. The notion of event continues to puzzle philosophers [120, 216, 182, 117, 139, 12, 103, 200, 102].

Discrete Behaviours

Definition 23 Discrete Behaviour: By a **discrete behaviour** we shall understand a set of sequences of potentially interacting sets of discrete actions, events and behaviours ■

Discrete behaviours now become the *focal point* of our investigation. To every part we associate, by transcendental deduction, a behaviour. We shall express these behaviours as CSP *processes* [148] For those behaviours we must therefore establish their means of *communication* via *channels*; their *signatures*; and their *definitions* – as *translated* from enduring parts.

Example 33: Behaviours

In the figure of the Channels example of Page 45 we “symbolically”, i.e., the “...”, show the following parts: each individual hub, each individual link, each individual bus company, each individual bus, and each individual automobile – and all of these. The idea is that those are the parts for which we shall define behaviours. That figure, however, and in contrast to Fig. 1.5 on Page 20, shows the composite parts as not containing their atomic parts, but as if they were “free-standing, atomic” parts. That shall visualise the transcendental interpretation as atomic part behaviours not being somehow embedded in composite behaviours, but operating concurrently, in parallel ■

⁵⁵ This is an example of a *transcendental deduction*.

1.8.2 Channels and Communication

We choose to exploit the CSP [148] subset of RSL since CSP is a suitable vehicle for expressing suitably abstract synchronisation and communication between behaviours.

The mereology of domain parts induces channel declarations.

CSP channels are loss-free. That is: two CSP processes, of which one offers and the other offers to accept a message do so synchronously and without forgetting that message. If you model actual, so-called “real-life” communication via queues or allowing “channels” to forget, then you must model that explicitly in CSP. We refer to [148, 225, 233].

The CSP Story:

CSP processes (models of domain behaviours), P_i, P_j, \dots, P_k can proceed in parallel:

$$P_i \parallel P_j \parallel \dots \parallel P_k$$

Behaviours sometimes synchronise and usually communicate. Synchronisation and communication is abstracted as the sending ($ch ! m$) and receipt ($ch ?$) of messages, $m:M$, over channels, ch .

```
type M
channel ch:M
```

Communication between (unique identifier) indexed behaviours have their channels modeled as similarly indexed channels:

```
out:    ch[idx]!m
in:     ch[idx]?
channel {ch[ide]:M|ide:IDE}
```

where IDE typically is some type expression over unique identifier types.

The expression

$$P_i \sqcap P_j \sqcap \dots \sqcap P_k$$

can be understood as a choice: either P_i , or P_j , or ... or P_k is *non-deterministically internally* chosen with no stipulation as to why!

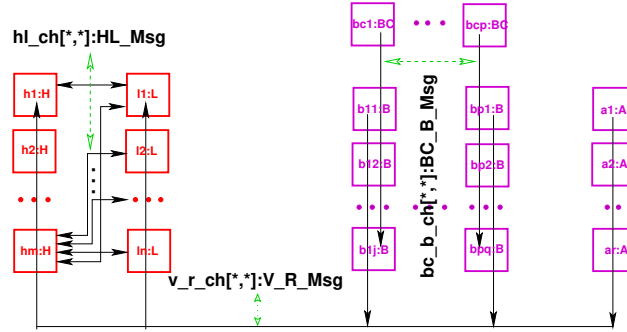
The expression

$$P_i \sqcap P_j \sqcap \dots \sqcap P_k$$

can be understood as a choice: either P_i , or P_j , or ... or P_k is *deterministically externally* chosen on the basis that the one chosen offers to participate in either an input, $ch ?$, or an output, $ch ! \text{msg}$, event. If

more than one P_i offers a communication then one is arbitrarily chosen. If no P_i offers a communication the behaviour halts till some P_j offers a communication.

Example 34: Channels



We shall argue for hub-to-link channels based on the mereologies of those parts. Hub parts may be topologically connected to any number, 0 or more, link parts. Only instantiated road nets knows which. Hence there must be channels between any hub behaviour and any link behaviour. Vice versa: link parts will be connected to exactly two hub parts. Hence there must be channels from any link behaviour to two hub behaviours. See the figure above.

Channel Message Types:

We ascribe types to the messages offered on channels.

- 72 Hubs and links communicate, both ways, with one another, over channels, hl_ch , whose indexes are determined by their mereologies.
- 73 Hubs send one kind of messages, links another.
- 74 Bus companies offer timed bus time tables to buses, one way.
- 75 Buses and automobiles offer their current, timed positions to the road element, hub or link they are on, one way.

type

- 73 H_L_Msg, L_H_Msg
- 72 $HL_Msg = H_L_Msg \mid L_F_Msg$
- 74 $BC_B_Msg = T \times BusTimTbl$
- 75 $V_R_Msg = T \times (BPos \mid APos)$

Channel Declarations:

76 This justifies the channel declaration which is calculated to be:

channel

- 76 { $hl_ch[h_ui, l_ui]: H_L_Msg$
- 76 | $h_ui: H_UI, l_ui: L_UI \mid i \in h_uis \wedge j \in lh_ui m(h_ui)$ }
- 76 \cup
- 76 { $hl_ch[h_ui, l_ui]: L_H_Msg$
- 76 | $h_ui: H_UI, l_ui: L_UI \mid l_ui \in l_uis \wedge i \in lh_ui m(l_ui)$ }

We shall argue for bus company-to-bus channels based on the mereologies of those parts. Bus companies need communicate to all its buses, but not the buses of other bus companies. Buses of a bus company need communicate to their bus company, but not to other bus companies.

77 This justifies the channel declaration which is calculated to be:

channel

- 77 { $bc_b_ch[bc_ui, b_ui] \mid bc_ui: BC_UI, b_ui: B_UI$
- 77 • $bc_ui \in bc_uis \wedge b_ui \in b_uis$ } : BC_B_Msg

We shall argue for vehicle to road element channels based on the mereologies of those parts. Buses and automobiles need communicate to all hubs and all links.

78 This justifies the channel declaration which is calculated to be:

channel

78 { $v_r_ch[v_ui,r_ui]$ | $v_ui:V_UI,r_ui:R_UI$

78 • $v_ui \in v_uis \wedge r_ui \in r_uis$ } : V_R_Msg

The channel calculations are described on Pages 49–51 ■

From Mereologies to Channel Declarations:

The fact that a part, p of sort P with unique identifier p_i , has a mereology, for example the set of unique identifiers $\{q_a, q_b, \dots, q_d\}$ identifying parts $\{q_a, q_b, \dots, q_d\}$ of sort Q , may mean that parts p and $\{q_a, q_b, \dots, q_d\}$ may wish to exchange – for example, attribute – values, one way (from p to the q_s) or the other (vice versa) or in both directions. Figure 1.7 shows two dotted rectangle box diagrams. The left fragment of the figure

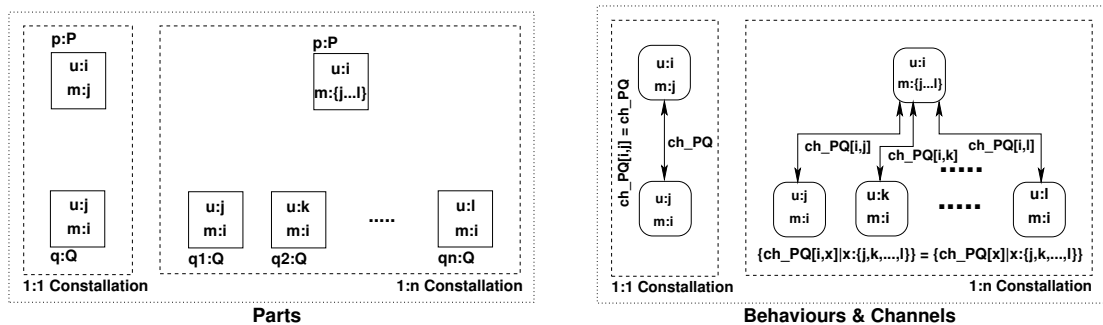


Fig. 1.7. Two Part and Channel Constallations. $u:p$ unique id. p ; $m:p$ mereology p

intends to show a 1:1 Constallation of a single $p:P$ box and a single $q:Q$ part, respectively, indicating, within these parts, their unique identifiers and mereologies. The right fragment of the figure intends to show a 1:n Constallation of a single $p:P$ box and a set of $q:Q$ parts, now with arrowed lines connecting the p part with the q parts. These lines are intended to show channels. We show them with two way arrows. We could instead have chosen one way arrows, in one or the other direction. The directions are intended to show a direction of value transfer. We have given the same channel names to all examples, ch_PQ . We have ascribed channel message types MPQ to all channels.⁵⁶ Figure 1.8 shows an arrangement similar to that of Fig. 1.7, but for an $m:n$ Constallation.

The channel declarations corresponding to Figs. 1.7 and 1.8 are:

channel

- [1] $ch_PQ[i,j]:MPQ$
- [2] { $ch_PQ[i,x]:MPQ$ | $x:\{j,k,\dots,l\}$ }
- [3] { $ch_PQ[p,q]:MPQ$ | $p:\{x,y,\dots,z\}, q:\{j,k,\dots,l\}$ }

Since there is only one index i and j for channel [1], its declaration can be reduced. Similarly there is only one i for declaration [2]:

⁵⁶ Of course, these names and types would have to be distinct for any one domain description.

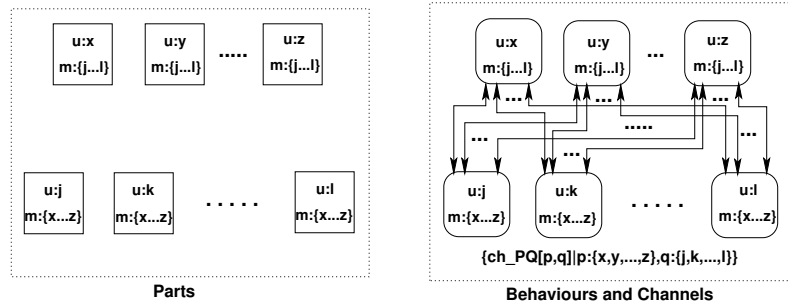


Fig. 1.8. Multiple Part and Channel Arrangements: $u:p$ unique id. p ; $m:p$ mereology p

channel

- [1] $ch_PQ:MPQ$
 [2] $\{ ch_PQ[x]:MPQ \mid x:\{j,k,\dots,l\} \}$

79 The following description identities holds:

$$79 \quad \{ ch_PQ[x]:MPQ \mid x:\{j,k,\dots,l\} \} \equiv ch_PQ[j], ch_PQ[k], \dots, ch_PQ[l],$$

$$79 \quad \{ ch_PQ[p,q]:MPQ \mid p:\{x,y,\dots,z\}, q:\{j,k,\dots,l\} \} \equiv$$

$$79 \quad ch_PQ[x,j], ch_PQ[x,k], \dots, ch_PQ[x,l],$$

$$79 \quad ch_PQ[y,j], ch_PQ[y,k], \dots, ch_PQ[y,l],$$

$$79 \quad \dots,$$

$$79 \quad ch_PQ[z,j], ch_PQ[z,k], \dots, ch_PQ[z,l]$$

We can sketch a diagram similar to Figs. 1.7 on the preceding page and 1.8 for the case of composite parts.

Continuous Behaviours:

By a **continuous behaviour** we shall understand a *continuous time* sequence of *state changes*. We shall not go into what may cause these *state changes*. And we shall not go into continuous behaviours in this chapter.

1.8.3 Perdurant Signatures

We shall treat perdurants as function invocations. In our cursory overview of perdurants we shall focus on one perdurant quality: function signatures.

Definition 24 Function Signature: By a **function signature** we shall understand a *function name* and a *function type expression* ■

Definition 25 Function Type Expression: By a **function type expression** we shall understand a pair of *type expressions*, separated by a *function type constructor* either \rightarrow (for **total function**) or $\tilde{\rightarrow}$ (for **partial function**) ■

The *type expressions* are part sort or type, or material sort or type, or component sort or type, or attribute type names, but may, occasionally be expressions over respective type names involving **-set**, \times , $*$, \rightarrow_m and $|$ type constructors.

Action Signatures and Definitions:

Actors usually provide their initiated actions with arguments, say of type VAL. Hence the schematic function (action) signature and schematic definition:

$$\begin{aligned} \text{action: } & \text{VAL} \rightarrow \Sigma \xrightarrow{\sim} \Sigma \\ \text{action}(v)(\sigma) & \text{ as } \sigma' \\ \text{pre: } & \mathcal{P}(v, \sigma) \\ \text{post: } & \mathcal{Q}(v, \sigma, \sigma') \end{aligned}$$

expresses that a selection of the domain, as provided by the Σ type expression, is acted upon and possibly changed. The partial function type operator $\xrightarrow{\sim}$ shall indicate that $\text{action}(v)(\sigma)$ may not be defined for the argument, i.e., initial state σ and/or the argument $v:\text{VAL}$, hence the precondition $\mathcal{P}(v, \sigma)$. The post condition $\mathcal{Q}(v, \sigma, \sigma')$ characterises the “after” state, $\sigma':\Sigma$, with respect to the “before” state, $\sigma:\Sigma$, and possible arguments ($v:\text{VAL}$). Which could be the argument values, $v:\text{VAL}$, of actions? Well, there can basically be only the following kinds of argument values: parts, components and materials, respectively unique part identifiers, mereologies and attribute values.

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the “smallest” relevant state⁵⁷, ascribing signatures to action functions, and determining action pre-conditions and action post-conditions. Of these, ascribing signatures is the most crucial: In the process of determining the action signature one oftentimes discovers that part or component or material attributes have been left (“so far”) “undiscovered”.

Event Signatures and Definitions:

Events are usually characterised by the absence of known actors and the absence of explicit “external” arguments. Hence the schematic function (event) signature:

value

$$\begin{aligned} \text{event: } & \Sigma \times \Sigma \xrightarrow{\sim} \mathbf{Bool} \\ \text{event}(\sigma, \sigma') & \text{ as } \text{tf} \\ \text{pre: } & P(\sigma) \\ \text{post: } & \text{tf} = Q(\sigma, \sigma') \end{aligned}$$

The event signature expresses that a selection of the domain as provided by the Σ type expression is “acted” upon, by unknown actors, and possibly changed. The partial function type operator $\xrightarrow{\sim}$ shall indicate that $\text{event}(\sigma, \sigma')$ may not be defined for some states σ . The resulting state may, or may not, satisfy axioms and well-formedness conditions over Σ – as expressed by the post condition $Q(\sigma, \sigma')$. Events may thus cause well-formedness of states to fail. Subsequent actions, once actors discover such “disturbing events”, are therefore expected to remedy that situation, that is, to restore well-formedness. We shall not illustrate this point.

Discrete Behaviour Signatures

Signatures: We shall only cover behaviour signatures when expressed in RSL/CSP [131]. The behaviour functions are now called processes. That a behaviour function is a never-ending function, i.e., a process, is “revealed” by the “trailing” **Unit**:

$$\text{behaviour: } \dots \rightarrow \dots \mathbf{Unit}$$

That a process takes no argument is “revealed” by a “leading” Unit:

⁵⁷ By “smallest” we mean: containing the fewest number of parts. Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

behaviour: **Unit** \rightarrow ...

That a process accepts channel, viz.: ch, inputs, is “revealed” as follows:

behaviour: ... \rightarrow **in** ch ...

That a process offers channel, viz.: ch, outputs is “revealed” as follows:

behaviour: ... \rightarrow **out** ch ...

That a process accepts other arguments is “revealed” as follows:

behaviour: ARG \rightarrow ...

where ARG can be any type expression:

T, $T \rightarrow T$, $T \rightarrow T \rightarrow T$, etcetera

where T is any type expression.

Attribute Access, An Interpretation:

We shall only be concerned with part attributes. And we shall here consider them in the context of part behaviours. Part behaviour definitions embody part attributes.

- **Static attributes** designate constants. As such they can be “compiled” into behaviour definitions. We choose, instead to list them as arguments.
- **Inert attributes** designate values provided by external stimuli, that is, must be obtained by channel input: `attr_Inert_A_ch ?`, i.e., are considered monitorable.
- **Reactive attributes** are functions of other attribute values.
- **Autonomous attributes** must be input, like inert attributes: `attr_Autonomous_A_ch ?`, i.e., are considered monitorable.
- **Programmable attribute** values are calculated by their behaviours. We list them as behaviour arguments. The behaviour definitions may then specify new values. These are provided in the position of the programmable attribute arguments in *tail recursive* invocations of these behaviours.
- **Biddable attributes** are now considered programmable attributes, but when provided, in possibly tail recursive invocations of their behaviour, the calculated biddable attribute value is *modified*, usually by some *perturbation* of the calculated value – to reflect that although they *are prescribed* they *may fail to be observed as such*.

Calculating In/Output Channel Signatures:

Given a part p we can calculate the $RSL^+ \text{Text}$ that designates the input channels on which part p behaviour obtains monitorable attribute values. For each monitorable attribute, A, the text $\llcorner \text{attr_A_ch} \lrcorner$ is to be “generated”. One or more such channel declaration contributions is to be preceded by the text $\llcorner \text{in} \lrcorner$. If there are no monitorable attributes then no text is to be yielded.

80 The function `calc_i.o.chn_refs` apply to parts and yield $RSL^+ \text{Text}$.

- From p we calculate its unique identifier value, its mereology value, and its monitorable attribute values.
- If there the mereology is not void and/or there are monitorable values then a (Currying⁵⁸) right pointing arrow, \rightarrow , is inserted.⁵⁹

⁵⁸ <https://en.wikipedia.org/wiki/Currying>

⁵⁹ We refer to the three parts of the mereology value as the input, the input/output and the output mereology (values).

- c If there is an input mereology and/or there are monitorable values then the keyword **in** is inserted in front of the monitorable attribute values and input mereology.
- d Similarly for the input/output mereology;
- e and for the output mereology.

value

```

80 calc_i_o_chn_refs: P → RSL+Text
80 calc_i_o_chn_refs(p) ≡ ;
80a   let ui = uid_P(p), (ics,iocs,ocs) = obs_mereo_(p), atrvs = obs_attrib_values_P(p) in
80b   if ics ∪ iocs ∪ ocs ∪ atrvs ≠ {} then ⋈ → ⋈ end ;
80c   if ics ∪ atrvs ≠ {} then ⋈in⋈ calc_attr_chn_refs(ui,atrvs), calc_chn_refs(ui,ichs) end ;
80d   if iocs≠{} then ⋈in,out⋈ calc_chn_refs(ui,iochs) end ;
80e   if ocs≠{} then ⋈out⋈ calc_chn_refs(ui,ochs) end end

```

81 The function `calc_attr_chn_refs`

- a apply to a set, `mas`, of monitorable attribute types and yield `RSL+Text`.
- b If `achs` is empty no text is generated. Otherwise a channel declaration `attr_A_ch` is generated for each attribute type whose name, `A`, which is obtained by applying η to an observed attribute value, ηa .

```

81a calc_attr_chn_refs: UI × A-set → RSL+Text
81b calc_attr_chn_refs(ui,mas) ≡ { ⋈ attr_ηa_ch[ui] ⋈ | a:A•a ∈ mas }

```

82 The function `calc_chn_refs`

- a apply to a pair, `(ui,uis)` of a unique part identifier and a set of unique part identifiers and yield `RSL+Text`.
- b If `uis` is empty no text is generated. Otherwise an array channel declaration is generated.

```

82a calc_chn_refs: P_UI × Q_UI-set → RSL+Text
82b calc_chn_refs(pui,quis) ≡ { ⋈ η(pui,qui)_ch[pui,qui] ⋈ | qui:Q_UI•qui ∈ quis }

```

83 The function `calc_all_chn_dcls`

- a apply to a pair, `(pui,quis)` of a unique part identifier and a set of unique part identifiers and yield `RSL+Text`.
- b If `quis` is empty no text is generated. Otherwise an array channel declaration
 - { ⋈ η(pui,qui)_ch[pui,qui]:η(pui,qui)M ⋈ | qui:Q_UI•qui ∈ quis } is generated.

```

83a calc_all_chn_dcls: P_UI × Q_UI-set → RSL+Text
83a calc_all_chn_dcls(pui,quis) ≡ { ⋈ η(pui,qui)_ch[pui,qui]:η(pui,qui)M ⋈ | qui:Q_UI•qui ∈ quis }

```

The $\eta(pui,qui)$ invocation serves to prefix-name both the channel, $\eta(pui,qui)_ch[pui,qui]$, and the channel message type, $\eta(pui,qui)M$.

84 The overloaded η operator⁶⁰ is here applied to a pair of unique identifiers.

```

84 η: (UI → RSL+Text)|((X_UI × Y_UI) → RSL+Text)
84 η(x-ui,y-ui) ≡ (⋈ η x-ui η y-ui ⋈)

```

Repeating these channel calculations over distinct parts p_1, p_2, \dots, p_n of the same part type `P` will yield “similar” behaviour signature channel references:

⁶⁰ The η operator applies to a type and yields the η name of the type.

$$\begin{aligned} & \{ \text{PQ_ch}[p_{1_{ui}}, \text{qui}] \mid p_{1_{ui}} : \text{P_UI}, \text{qui} : \text{Q_UI} \cdot \text{qui} \in \text{quis} \} \\ & \{ \text{PQ_ch}[p_{2_{ui}}, \text{qui}] \mid p_{2_{ui}} : \text{P_UI}, \text{qui} : \text{Q_UI} \cdot \text{qui} \in \text{quis} \} \\ & \dots \\ & \{ \text{PQ_ch}[p_{n_{ui}}, \text{qui}] \mid p_{n_{ui}} : \text{P_UI}, \text{qui} : \text{Q_UI} \cdot \text{qui} \in \text{quis} \} \end{aligned}$$

These distinct single channel references can be assembled into one:

$$\begin{aligned} & \{ \text{PQ_ch}[p_{ui}, \text{qui}] \mid p_{ui} : \text{P_UI}, \text{qui} : \text{Q_UI} : -p_{ui} \in \text{puis}, \text{qui} \in \text{quis} \} \\ & \text{where } \text{puis} = \{ p_{1_{ui}}, p_{2_{ui}}, \dots, p_{n_{ui}} \} \end{aligned}$$

As an example we have already calculated the array channels for Fig. 1.8 Pg. 47– cf. the left, the **Parts**, of that figure – cf. Items [1–3] Pages 46–47. The identities Item 79 Pg. 47 apply.

1.8.4 Discrete Behaviour Definitions

We associate with each part, $p:P$, a behaviour name \mathcal{M}_p . Behaviours have as first argument their unique part identifier: $\text{uid}_P(p)$. Behaviours evolves around a state, or, rather, a set of values: its possibly changing mereology, $\text{mt}:\text{MT}$ and the attributes of the part.⁶¹ A behaviour signature is therefore:

$$\mathcal{M}_p: \text{ui}:\text{UI} \times \text{me}:\text{MT} \times \text{stat_attr_typs}(p) \rightarrow \text{ctrl_attr_typs}(p) \rightarrow \text{calc_i_o_chn_refs}(p) \text{ Unit}$$

where (i) $\text{ui}:\text{UI}$ is the unique identifier value and type of part p ; (ii) $\text{me}:\text{MT}$ is the value and type mereology of part p , $\text{me} = \text{obs_mereo}_P(p)$; (iii) $\text{stat_attr_typs}(p)$: static attribute types of part $p:P$; (iv) $\text{ctrl_attr_typs}(p)$: controllable attribute types of part $p:P$; (v) $\text{calc_i_o_chn_refs}(p)$ calculates references to the **input**, the **input/output** and the **output** channels serving the attributes shared between part p and the parts designated in its mereology me . Let P be a composite sort defined in terms of *endurant*⁶² sub-sorts E_1, E_2, \dots, E_n . The behaviour description *translated* from $p:P$, is composed from a behaviour description, \mathcal{M}_p , relying on and handling the unique identifier, mereology and attributes of part p to be *translated* with behaviour descriptions $\beta_1, \beta_2, \dots, \beta_n$ where β_1 is *translated* from $e_1:E_1$, β_2 is *translated* from $e_2:E_2$, ..., and β_n is *translated* from $e_n:E_n$. The domain description *translation* schematic below “formalises” the above.

Abstract `is_composite(p)` Behaviour schema

```

value
  Translatep: P → RSL+Text
  Translatep(p) ≡
    let ui = uidP(p), me = obs_mereoP(p),
        sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
        MT = mereo_type(p), ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
        IOR = calc_i_o_chn_refs(p), IOD = calc_all_ch_dcls(p) in
    ⋈ channel
      IOD
      value
        Mp: P_UI × MT × ST CT IOR Unit
        Mp(ui, me, sta)(pa) ≡ Bp(ui, me, sta)ca
        , ⋈ Translatep1(obs_endurant_sortsE1(p))
        ⋈ Translatep2(obs_endurant_sortsE2(p))
        ⋈ ...
        ⋈ Translatepn(obs_endurant_sortsEn(p))
      end
end

```

⁶¹ We leave out consideration of possible components and materials of the part.

⁶² – structures or composite

Expression $\mathcal{B}_P(ui, me, sta, pa)$ stands for the *behaviour definition body* in which the names ui , me , sta , pa are bound to the *behaviour definition head*, i.e., the left hand side of the \equiv . Endurant sorts E_1, E_2, \dots, E_n are obtained from the `observe_endurant_sorts` prompt, Page 19. We informally explain the `TranslatePi` function. It takes endurants and produces RSL^+ Text. Resulting texts are bracketed: $\langle\langle rsl_text \rangle\rangle$

Example 35: Signatures

We first decide on names of behaviours. In Sect. 1.8.4, Pages 51–54, we gave schematic names to behaviours of the form \mathcal{M}_P . We now assign mnemonic names: from part names to names of transcendently interpreted behaviours and then we assign signatures to these behaviours.

85 $hub_{h_{ui}}$:

- a there is the usual “triplet” of arguments: unique identifier, mereology and static attributes;
- b then there are the programmable attributes;
- c and finally there are the input/output channel references: first those allowing communication between hub and link behaviours,
- d and then those allowing communication between hub and vehicle (bus and automobile) behaviours.

value

85 $hub_{h_{ui}}$:

85a $h_{ui}: H_UI \times (vuis, luis, _): H_Mer \times H\Omega$

85b $\rightarrow (H\Sigma \times H_Traffic)$

85c $\rightarrow \mathbf{in, out} \{ h_l_ch[h_{ui}, l_{ui}] \mid l_{ui}: L_UI \bullet l_{ui} \in luis \}$

85d $\{ ba_r_ch[h_{ui}, v_{ui}] \mid v_{ui}: V_UI \bullet v_{ui} \in vuis \}$ **Unit**

85a **pre:** $vuis = v_{uis} \wedge luis = l_{uis}$

86 $link_{l_{ui}}$:

- a there is the usual “triplet” of arguments: unique identifier, mereology and static attributes;
- b then there are the programmable attributes;
- c and finally there are the input/output channel references: first those allowing communication between hub and link behaviours,
- d and then those allowing communication between link and vehicle (bus and automobile) behaviours.

value

86 $link_{l_{ui}}$:

86a $l_{ui}: L_UI \times (vuis, huis, _): L_Mer \times L\Omega$

86b $\rightarrow (L\Sigma \times L_Traffic)$

86c $\rightarrow \mathbf{in, out} \{ h_l_ch[h_{ui}, l_{ui}] \mid h_{ui}: H_UI \bullet h_{ui} \in huis \}$

86d $\{ ba_r_ch[l_{ui}, v_{ui}] \mid v_{ui}: (B_UI \mid A_UI) \bullet v_{ui} \in vuis \}$ **Unit**

86a **pre:** $vuis = v_{uis} \wedge huis = h_{uis}$

87 $bus_company_{bc_{ui}}$:

- a there is here just a “doublet” of arguments: unique identifier and mereology;
- b then there is the one programmable attribute;
- c and finally there are the input/output channel references allowing communication between the bus company and buses.

value

87 $bus_company_{bc_{ui}}$:

87a $bc_{ui}: BC_UI \times (_, _, buis): BC_Mer$

87b $\rightarrow BusTimTbl$

87c $\mathbf{in, out} \{ bc_b_ch[bc_{ui}, b_{ui}] \mid b_{ui}: B_UI \bullet b_{ui} \in buis \}$ **Unit**


```

87a  pre: buis =  $b_{uis}$   $\wedge$  huis =  $h_{uis}$ 

.....

88  bus $_{b_{ui}}$ :
    a  there is here just a “doublet” of arguments: unique identifier and mereology;
    b  then there are the programmable attributes;
    c  and finally there are the input/output channel references: first the input/output allowing communication
       between the bus company and buses,
    d  and the input/output allowing communication between the bus and the hub and link behaviours.

value
88  bus $_{b_{ui}}$ :
88a  b $_{ui}$ :B_UI  $\times$  (bc $_{ui}$ ,__,ruis):B_Mer
88b   $\rightarrow$  (LN  $\times$  BTT  $\times$  BPOS)
88c   $\rightarrow$  out bc $_{b\_ch}$ [bc $_{ui}$ ,b $_{ui}$ ],
88d      {ba $_{r\_ch}$ [r $_{ui}$ ,b $_{ui}$ ]|r $_{ui}$ :(H_UI|L_UI) $\cdot$ ui $\in v_{uis}$ } Unit
88a  pre: ruis =  $r_{uis}$   $\wedge$  bc $_{ui} \in bc_{uis}$ 

.....

89  automobile $_{a_{ui}}$ :
    a  there is the usual “triplet” of arguments: unique identifier, mereology and static attributes;
    b  then there is the one programmable attribute;
    c  and finally there are the input/output channel references allowing communication between the automobile
       and the hub and link behaviours.

value
89  automobile $_{a_{ui}}$ :
89a  a $_{ui}$ :A_UI  $\times$  (__,__,ruis):A_Mer  $\times$  rn:RegNo
89b   $\rightarrow$  apos:APos
89c  in,out {ba $_{r\_ch}$ [a $_{ui}$ ,r $_{ui}$ ]|r $_{ui}$ :(H_UI|L_UI) $\cdot$ r $_{ui} \in ruis$ } Unit
89a  pre: ruis =  $r_{uis}$   $\wedge$  a $_{ui} \in a_{uis}$  ■

```

For the case that an enduring is a structure there is only its elements to compile; otherwise Schema 2 is as Schema 1.

Abstract `is_structure(e)` Behaviour schema

```

value
  Translate $_E$ (e)  $\equiv$ 
    Translate $_{E_1}$ (obs_endurant_sorts $_{E_1}$ (e))
    <<> Translate $_{E_2}$ (obs_endurant_sorts $_{E_2}$ (e))
    <<> ...
    <<> Translate $_{E_n}$ (obs_endurant_sorts $_{E_n}$ (e))

```

Let P be a composite sort defined in terms of the concrete type Q -set. The process definition compiled from $p:P$, is composed from a process, \mathcal{M}_p , relying on and handling the unique identifier, mereology and attributes of process p as defined by P operating in parallel with processes $q:\mathbf{obs_part_Qs}(p)$. The domain description “compilation” schematic below “formalises” the above.

Concrete `is_composite(p)` Behaviour schema

```

type
  Qs = Q-set

```

```

value
qs:Q-set = obs_part_Qs(p)
Translatep(p) ≡
  let ui = uid_P(p), me = obs_mereo_P(p),
    sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
    ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
    IOR = calc_i_o_chn_refs(p), IOD = calc_all_ch_dcls(p) in
  ⋈ channel
    IOD
    value
       $\mathcal{M}_p: P\_UI \times MT \times ST \ CT \ IOR \ \mathbf{Unit}$ 
       $\mathcal{M}_p(ui,me,sa)ca \equiv \mathcal{B}_p(ui,me,sa)ca \ \bowtie$ 
      { ⋈, ⋉ Translateq(q) | q:Q•q ∈ qs }
    end

```

Atomic **is_atomic**(p) Behaviour schema

```

value
Translatep(p) ≡
  let ui = uid_P(p), me = obs_mereo_P(p),
    sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
    ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
    IOR = calc_i_o_chn_refs(p), IOD = calc_all_chs(p) in
  ⋈ channel
    IOD
    value
       $\mathcal{M}_p: P\_UI \times MT \times ST \ PT \ IOR \ \mathbf{Unit}$ 
       $\mathcal{M}_p(ui,me,sa)ca \equiv \mathcal{B}_p(ui,me,sa)ca \ \bowtie$ 
    end

```

The core processes can be understood as never ending, “tail recursively defined” processes:

Core Behaviour schema

```

 $\mathcal{B}_p: uid:P\_UI \times me:MT \times sa:SA \rightarrow ct:CT \rightarrow \mathbf{in} \ \mathbf{in\_chns}(p) \ \mathbf{in, out} \ \mathbf{in\_out\_chns}(me) \ \mathbf{Unit}$ 
 $\mathcal{B}_p(p)(ui,me,sa)(ca) \equiv \mathbf{let} \ (me',ca') = \mathcal{F}_p(ui,me,sa)ca \ \mathbf{in} \ \mathcal{M}_p(ui,me',sa)ca' \ \mathbf{end}$ 
 $\mathcal{F}_p: P\_UI \times MT \times ST \rightarrow CT \rightarrow \mathbf{in\_out\_chns}(me) \rightarrow MT \times CT$ 

```

We refer to [70, Process Schema V: Core Process (II), Page 40] for possible forms of \mathcal{F}_p .

Example 36: Automobile Behaviour (at a hub)

We define the behaviours in a different order than the treatment of their signatures. We “split” definition of the automobile behaviour into the behaviour of automobiles when positioned at a hub, and into the behaviour automobiles when positioned at on a link. In both cases the behaviours include the “idling” of the automobile, i.e., its “not moving”, standing still.

- 90 We abstract automobile behaviour at a Hub (hui).
- 91 The vehicle remains at that hub, “idling”,
- 92 informing the hub behaviour,
- 93 or, internally non-deterministically,

```

      a moves onto a link,  $tl_i$ , whose “next” hub, identified by  $th_{ui}$ , is obtained from the mereology of the link
      identified by  $tl_{ui}$ ;
      b informs the hub it is leaving and the link it is entering of its initial link position,
      c whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (0) of that link,
94 or, again internally non-deterministically,
95 the vehicle “disappears — off the radar” !

90 automobile $_{a_{ui}}$ ( $a_{ui}$ ,( $\{\},\{ruis,vuis\}\},rn$ )
90   (apos:atH(fl $_{ui}$ ,h $_{ui}$ ,tl $_{ui}$ ))  $\equiv$ 
91   (ba $_r$ _ch[ $a_{ui}$ ,h $_{ui}$ ] ! (record $_$ TIME(),atH(fl $_{ui}$ ,h $_{ui}$ ,tl $_{ui}$ )));
92   automobile $_{a_{ui}}$ ( $a_{ui}$ ,( $\{\},\{ruis,vuis\}\},rn$ )(apos))
93    $\sqcap$ 
93a (let ( $\{fh_{ui},th_{ui}\},ruis'$ )= $\text{mereo}_L(\wp(tl_{ui}))$  in
93a   assert:  $fh_{ui}=h_{ui} \wedge ruis=ruis'$ 
90   let onl = (tl $_{ui}$ ,h $_{ui}$ ,0,th $_{ui}$ ) in
93b (ba $_r$ _ch[ $a_{ui}$ ,h $_{ui}$ ] ! (record $_$ TIME(),onL(onl)) ||
93b   ba $_r$ _ch[ $a_{ui}$ ,tl $_{ui}$ ] ! (record $_$ TIME(),onL(onl))) ;
93c   automobile $_{a_{ui}}$ ( $a_{ui}$ ,( $\{\},\{ruis,vuis\}\},rn$ )
93c   (onL(onl)) end end)
94    $\sqcap$ 
95   stop

```

Section 1.9.3 presents the definition of the remaining automobile, hub, link, bus company and bus behaviours.

1.8.5 Running Systems

It is one thing to define the behaviours corresponding to all parts, whether composite or atomic. It is another thing to specify an initial configuration of behaviours, that is, those behaviours which “start” the overall system behaviour. The choice as to which parts, i.e., behaviours, are to represent an initial, i.e., a start system behaviour, cannot be “formalised”, it really depends on the “deeper purpose” of the system. In other words: requires careful analysis and is beyond the scope of the present chapter.

Example 36: Initial System

Initial States: We recall the *hub*, *link*, *bus company*, *bus* and the *automobile states* first mentioned in Sect. 1.3.8 Page 18.

value

```

61  $hs:H\text{-set} \equiv \equiv \text{obs\_sH}(\text{obs\_SH}(\text{obs\_RN}(rts)))$ 
62  $ls:L\text{-set} \equiv \equiv \text{obs\_sL}(\text{obs\_SL}(\text{obs\_RN}(rts)))$ 
64  $bcs:BC\text{-set} \equiv \equiv \text{obs\_BCs}(\text{obs\_SBC}(\text{obs\_FV}(\text{obs\_RN}(rts))))$ 
65  $bs:B\text{-set} \equiv \equiv \cup\{\text{obs\_Bs}(bc)|bc:BC \bullet bc \in bcs\}$ 
67  $as:A\text{-set} \equiv \equiv \text{obs\_BCs}(\text{obs\_SBC}(\text{obs\_FV}(\text{obs\_RN}(rts))))$ 

```

Starting Initial Behaviours: We are reaching the end of this domain modelling example. Behind us there are narratives and formalisations¹ Pg. 19 – 139 Pg. 63. Based on these we now express the signature and the body of the definition of a “system build and execute” function.

```

96 The system to be initialised is
    a the parallel composition ( $\parallel$ ) of
    b the distributed parallel composition ( $\parallel\{\dots\}\dots$ ) of
    c all the hub behaviours,
    d all the link behaviours,
    e all the bus company behaviours,
    f all the bus behaviours, and
    g all the automobile behaviours.

```

```

value
96  initial_system: Unit → Unit
96  initial_system() ≡
96c  || { hubhui(hui,me,hω)(htrf,hσ)
96c  | h:H•h ∈ hs,
96c  | hui:H_UI•hui=uid_H(h),
96c  | me:HMetL•me=mereo_H(h),
96c  | hω:HΩ•hω=attr_HΩ(h),
96c  | htrf:H_Traffic•htrf=attr_H_Traffic_H(h),
96c  | hσ:HΣ•hσ=attr_HΣ(h)∧hσ ∈ hω
96c  }

96a  ||
96d  || { linklui(lui,me,lω)(ltrf,lσ)
96d  | l:L•l ∈ ls,
96d  | lui:L_UI•lui=uid_L(l),
96d  | me:LMet•me=mereo_L(l),
96d  | lω:LΩ•lω=attr_LΩ(l),
96d  | ltrf:L_Traffic•ltrf=attr_L_Traffic_H(l),
96d  | lσ:LΣ•lσ=attr_LΣ(l)∧lσ ∈ lω
96d  }

96a  ||
96e  || { bus_companybcui(bcui,me)(btt)
96e  | bc:BC•bc ∈ bcs,
96e  | bcui:BC_UI•bcui=uid_BC(bc),
96e  | me:BCMet•me=mereo_BC(bc),
96e  | btt:BusTimTbl•btt=attr_BusTimTbl(bc)
96e  }

96a  ||
96f  || { busbui(bui,me)(ln,btt,bpos)
96f  | b:B•b ∈ bs,
96f  | bui:B_UI•bui=uid_B(b),
96f  | me:BMet•me=mereo_B(b),
96f  | ln:LN•ln=attr_LN(b),
96f  | btt:BusTimTbl•btt=attr_BusTimTbl(b),
96f  | bpos:BPos•bpos=attr_BPos(b)
96f  }

96a  ||
96g  || { automobileaui(aui,me,rn)(apos)
96g  | a:A•a ∈ as,
96g  | aui:A_UI•aui=uid_A(a),
96g  | me:AMet•me=mereo_A(a),
96g  | rn:RegNo•rn=attr_RegNo(a),
96g  | apos:APos•apos=attr_APos(a)
96g  } ■

```

1.8.6 Concurrency: Communication and Synchronisation

Process Schemas I, II, III and V (Pages 51, 53, 53 and 54), reveal that two or more parts, which temporally coexist (i.e., at the same time), imply a notion of *concurrency*. Process Schema IV, Page 54, through the RSL/CSP language expressions $ch!v$ and $ch?$, indicates the notions of *communication* and *synchronisation*. Other than this we shall not cover these crucial notion related to *parallelism*.

1.8.7 Summary and Discussion of Perdurants

The most significant contribution of Sect. 1.8 has been to show that for every domain description there exists a normal form behaviour — here expressed in terms of a CSP process expression.

Summary

We have proposed to analyse perdurant entities into actions, events and behaviours – all based on notions of state and time. We have suggested modelling and abstracting these notions in terms of functions with signatures and pre-/post-conditions. We have shown how to model behaviours in terms of CSP (communicating sequential processes). It is in modelling function signatures and behaviours that we justify the enduring entity notions of parts, unique identifiers, mereology and shared attributes.

Discussion

The analysis of perdurants into actions, events and behaviours represents a choice. We suggest skeptical readers to come forward with other choices.

1.9 The Example Concluded

Example 36:

1.9.1 Unique Identifier Concepts

We define a few concepts related to unique identification.

Extract Parts from Their Unique Identifiers:

97 From the unique identifier of a part we can retrieve, \emptyset , the part having that identifier.

type

97 $P = H \mid L \mid BC \mid B \mid A$

value

97 $\emptyset: H_UI \rightarrow H \mid L_UI \rightarrow L \mid BC_UI \rightarrow BC \mid B_UI \rightarrow B \mid A_UI \rightarrow A$

97 $\emptyset(ui) \equiv \text{let } p: (H \mid L \mid BC \mid B \mid A) \bullet p \in ps \wedge uid_P(p) = ui \text{ in } p \text{ end}$

Unique Identifier Constants

We can calculate:

98 the set, $h_{ui}s$, of unique hub identifiers;

99 the set, $l_{ui}s$, of unique link identifiers;

100 the map, $hl_{ui}m$, from unique hub identifiers to the set of unique link identifiers of the links connected to the zero, one or more identified hubs,

101 the map, $lh_{ui}m$, from unique link identifiers to the set of unique hub identifiers of the two hubs connected to the identified link;

102 the set, $r_{ui}s$, of all unique hub and link, i.e., road identifiers;

103 the set, bc_{uis} , of unique bus company identifiers;
 104 the set, b_{uis} , of unique bus identifiers;
 105 the set, a_{uis} , of unique private automobile identifiers;
 106 the set, v_{uis} , of unique bus and automobile, i.e., vehicle identifiers;
 107 the map, $bc_{b_{uis}m}$, from unique bus company identifiers to the set of its unique bus identifiers; and
 108 the (bijective) map, $bbc_{ui}bm$, from unique bus identifiers to their unique bus company identifiers.

98 $h_{uis}:H_UI\text{-set} \equiv \{uid_H(h)|h:H \cdot h \in hs\}$
 99 $l_{uis}:L_UI\text{-set} \equiv \{uid_L(l)|l:L \cdot l \in ls\}$
 102 $r_{uis}:R_UI\text{-set} \equiv h_{uis} \cup l_{uis}$
 100 $hl_{uis}m:(H_UI \rightarrow_m L_UI\text{-set}) \equiv$
 100 $[h_ui \mapsto luis|h_ui:H_UI, luis:L_UI\text{-set} \cdot h_ui \in h_{uis}$
 100 $\wedge (_, luis, _) = mereo_H(\eta(h_ui))] \text{ [cf. Item 24]}$
 101 $lh_{uis}m:(L_UI \rightarrow_m H_UI\text{-set}) \equiv$
 101 $[l_ui \mapsto huis \text{ [cf. Item 25]}$
 101 $| h_ui:L_UI, huis:H_UI\text{-set} \cdot l_ui \in l_{uis}$
 101 $\wedge (_, huis, _) = mereo_L(\eta(l_ui))]$
 103 $bc_{uis}:BC_UI\text{-set} \equiv \{uid_BC(bc)|bc:BC \cdot bc \in bcs\}$
 104 $b_{uis}:B_UI\text{-set} \equiv \cup \{uid_B(b)|b:B \cdot b \in bs\}$
 105 $a_{uis}:A_UI\text{-set} \equiv \{uid_A(a)|a:A \cdot a \in as\}$
 106 $v_{uis}:V_UI\text{-set} \equiv b_{uis} \cup a_{uis}$
 107 $bc_{b_{uis}m}:(BC_UI \rightarrow_m B_UI\text{-set}) \equiv$
 107 $[bc_ui \mapsto buis$
 107 $| bc_ui:BC_UI, bc:BC \cdot$
 107 $bc \in bcs \wedge bc_ui = uid_BC(bc)$
 107 $\wedge (_, _, buis) = mereo_BC(bc)]$
 108 $bbc_{ui}bm:(B_UI \rightarrow_m BC_UI) \equiv$
 108 $[b_ui \mapsto bc_ui$
 108 $| b_ui:B_UI, bc_ui:BC_ui \cdot$
 108 $bc_ui = \mathbf{dom}bc_{b_{uis}m} \wedge b_ui \in bc_{b_{uis}m}(bc_ui)]$

Uniqueness of Part Identifiers:

We refer to Sect. 1.5.4 Pg. 35. We must express the following axioms:

109 All hub identifiers are distinct.
 110 All link identifiers are distinct.
 111 All bus company identifiers are distinct.
 112 All bus identifiers are distinct.
 113 All private automobile identifiers are distinct.
 114 All part identifiers are distinct.

109 $\mathbf{card} hs = \mathbf{card} h_{uis}$
 110 $\mathbf{card} ls = \mathbf{card} l_{uis}$
 111 $\mathbf{card} bcs = \mathbf{card} bc_{uis}$
 112 $\mathbf{card} bs = \mathbf{card} b_{uis}$
 113 $\mathbf{card} as = \mathbf{card} a_{uis}$
 114 $\mathbf{card} \{h_{uis} \cup l_{uis} \cup bc_{uis} \cup b_{uis} \cup a_{uis}\}$
 114 $= \mathbf{card} h_{uis} + \mathbf{card} l_{uis} + \mathbf{card} bc_{uis} + \mathbf{card} b_{uis} + \mathbf{card} a_{uis}$

1.9.2 Further Transport System Attributes

Links: We show just a few attributes.

115 There is a link state. It is a set of pairs, (h_f, h_t) , of distinct hub identifiers, where these hub identifiers are in the mereology of the link. The meaning of a link state in which (h_f, h_t) is an element is that the link is open, "green", for traffic from hub h_f to hub h_t . Link states can have either 0, 1 or 2 elements.

116 There is a link state space. It is a set of link states. The meaning of the link state space is that its states are all those the which the link can attain. The current link state must be in its state space. If a link state space is empty then the link is (permanently) closed. If it has one element then it is a one-way link. If a one-way link, l , is imminent on a hub whose mereology designates that link, then the link is a “trap”, i.e., a “blind cul-de-sac”.

117 Since we can think rationally about it, it can be described, hence it can model, as an attribute of links a history of its traffic: the recording, per unique bus and automobile identifier, of the time ordered positions along the link (from one hub to the next) of these vehicles.

118 The hub identifiers of link states must be in the set, $h_{ui}s$, of the road net’s hub identifiers.

type

115 $L\Sigma = H_UI\text{-set}$ [programmable, Df.8 Pg.30]

axiom

115 $\forall l\sigma:L\Sigma\cdot\text{card } l\sigma=2$

115 $\forall l:L\cdot\text{obs_L}\Sigma(l) \in \text{obs_L}\Omega(l)$

type

116 $L\Omega = L\Sigma\text{-set}$ [static, Df.1 Pg.30]

117 $L_Traffic$ [programmable, Df.8 Pg.30]

117 $L_Traffic = (A_UI|B_UI) \rightarrow_{\#} (\mathbb{T}\times(H_UI\times\text{Frac}\times H_UI))^*$

117 $\text{Frac} = \text{Real}$, axiom $\text{frac}:\text{Frac} \cdot 0 < \text{frac} < 1$

value

115 $\text{attr_L}\Sigma: L \rightarrow L\Sigma$

116 $\text{attr_L}\Omega: L \rightarrow L\Omega$

117 $\text{attr_L_Traffic}: : \rightarrow L_Traffic$

axiom

117 $\forall lt:L_Traffic,ui:(A_UI|B_UI)\cdot ui \in \text{dom } ht$

117 $\Rightarrow \text{time_ordered}(ht(ui))$

118 $\forall l:L\cdot l \in ls \Rightarrow$

118 **let** $l\sigma = \text{attr_L}\Sigma(l)$ **in**

118 $\forall (h_{ui},h_{ui'}):(H_UI\times K_UI) \cdot$

118 $(h_{ui},h_{ui'}) \in l\sigma \Rightarrow \{h_{ui},h_{ui'}\} \subseteq h_{ui}s$ **end**

Bus Companies:

Bus companies operate a number of lines that service passenger transport along routes of the road net. Each line being serviced by a number of buses.

119 Bus companies create, maintain, revise and distribute [to the public (not modeled here), and to buses] bus time tables, not further defined.

type

119 BusTimTbl [programmable, Df.8 Pg.30]

value

119 $\text{attr_BusTimTbl}: BC \rightarrow \text{BusTimTbl}$

There are two notions of time at play here: the indefinite “real” or “actual” time; and the definite calendar, hour, minute and second time designation occurring in some textual form in, e.g., time tables.

Buses: We show just a few attributes:

120 Buses run routes, according to their line number, $ln:LN$, in the

121 bus time table, $btt:\text{BusTimTbl}$ obtained from their bus company, and and keep, as inert attributes, their segment of that time table.

122 Buses occupy positions on the road net:

a either at a hub identified by some h_{ui} ,

b or on a link, some fraction, $f:\text{Frac}$, down an identified link, L_{ui} , from one of its identified connecting hubs, fh_{ui} , in the direction of the other identified hub, th_{ui} .

123 Et cetera.

```

type
120 LN [programmable, Df.8 Pg.30]
121 BusTimTbl [inert, Df.3 Pg.30]
122 BPos == atHub | onLink [programmable, Df.8 Pg.30]
122a atHub :: h_ui:H_UI
122b onLink :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
122b Fract = Real, axiom frac:Fract • 0 < frac < 1
123 ...
value
121 attr_BusTimTbl: B → BusTimTbl
122 attr_BPos: B → BPos

```

Private Automobiles: We show just a few attributes:
 We illustrate but a few attributes:

- 124 Automobiles have static number plate registration numbers.
 125 Automobiles have dynamic positions on the road net:
 [122a] either at a hub identified by some h_{ui} ,
 [122b] or on a link, some fraction, $frac:Fract$ down an identified link, L_{ui} , from one of its identified connecting hubs, fh_{ui} , in the direction of the other identified hub, th_{ui} .

```

type
124 RegNo [static, Df.1 Pg.30]
125 APos == atHub | onLink [programmable, Df.8 Pg.30]
122a atHub :: h_ui:H_UI
122b onLink :: fh_ui:H_UI × l_ui:L_UI × frac:Fract × th_ui:H_UI
122b Fract = Real, axiom frac:Fract • 0 < frac < 1
value
124 attr_RegNo: A → RegNo
125 attr_APos: A → APos

```

Obvious attributes that are not illustrated are those of velocity and acceleration, forward or backward movement, turning right, left or going straight, etc. The acceleration, deceleration, even velocity, or turning right, turning left, moving straight, or forward or backward are seen as command actions. As such they denote actions by the automobile — such as *pressing the accelerator*, or *lifting accelerator pressure* or *braking*, or *turning the wheel in one direction or another*, etc. As actions they have a kind of counterpart in the velocity, the acceleration, etc. attributes.

Discussion:

Observe that bus companies each have their own distinct *bus time table*, and that these are modeled as *programmable*, Item 119 on the previous page, Page 59. Observe then that buses each have their own distinct *bus time table*, and that these are model-led as *inert*, Item 121 on the preceding page, Page 59. In Items 135–136 Pg. 62 we shall see how the buses communicate with their respective bus companies in order for the buses to obtain the *programmed* bus time tables “in lieu” of their *inert* one! In Items 33 Pg. 31 and 117 Pg. 58, we illustrated an aspect of domain analysis & description that may seem, and at least some decades ago would have seemed, strange: namely that if we can think, hence speak, about it, then we can model it “as a fact” in the domain. The case in point is that we include among hub and link attributes their histories of the timed whereabouts of buses and automobiles.⁶³

1.9.3 Behaviours

Automobile Behaviour (on a link)

126 We abstract automobile behaviour on a Link.

⁶³ In this day and age of road cameras and satellite surveillance these traffic recordings may not appear so strange: We now know, at least in principle, of technologies that can record approximations to the hub and link traffic attributes.


```

a Internally non-deterministically, either
  i the automobile remains, “idling”, i.e., not moving, on the link,
  ii however, first informing the link of its position,
b or
  i if if the automobile’s position on the link has not yet reached the hub, then
    1 then the automobile moves an arbitrary small, positive Real-valued increment along the link
    2 informing the hub of this,
    3 while resuming being an automobile at the new position, or
  ii else,
    1 while obtaining a “next link” from the mereology of the hub (where that next link could very well
      be the same as the link the vehicle is about to leave),
    2 the vehicle informs both the link and the imminent hub that it is now at that hub, identified by th_ui,
    3 whereupon the vehicle resumes the vehicle behaviour positioned at that hub;
c or
d the vehicle “disappears — off the radar” !

126 automobileau(a_ui,({},ruis,{}),rno)
126      (vp:onL(fh_ui,l_ui,f,th_ui)) ≡
126(a)ii (ba_r_ch[thui,au]!atH(lui,thui,nxt_lui) ;
126(a)i  automobileau(a_ui,({},ruis,{}),rno)(vp))
126b   []
126(b)i (if not_yet_at_hub(f)
126(b)i  then
126(b)i1  (let incr = increment(f) in
90      let onl = (tl_ui,h_ui,incr,th_ui) in
126(b)i2  ba_r_ch[l_ui,a_ui] ! onL(onl) ;
126(b)i3  automobileau(a_ui,({},ruis,{}),rno)
126(b)i3  (onL(onl))
126(b)i  end end)
126(b)ii else
126(b)ii1 (let nxt_lui:L_UI•nxt_lui ∈ mereo_H(∅(th_ui)) in
126(b)ii2 ba_r_ch[thui,au]!atH(l_ui,th_ui,nxt_lui) ;
126(b)ii3 automobileau(a_ui,({},ruis,{}),rno)
126(b)ii3 (atH(l_ui,th_ui,nxt_lui)) end)
126(b)i  end)
126c   []
126d   stop
126(b)i1 increment: Fract → Fract

```

Hub Behaviour We model the hub behaviour vis-a-vis vehicles: buses and automobiles.

```

127 The hub behaviour
  a non-deterministically, externally offers
  b to accept timed vehicle positions —
  c which will be at the hub, from some vehicle, v_ui.
  d The timed vehicle hub position is appended to the front of that vehicle’s entry in the hub’s traffic table;
  e whereupon the hub proceeds as a hub behaviour with the updated hub traffic table.
  f The hub behaviour offers to accept from any vehicle.
  g A post condition expresses what is really a proof obligation: that the hub traffic, ht’ satisfies the axiom of
    the enduring hub traffic attribute Item 33 Pg. 31.

```

value

```

127 hubh(h_ui,(luis,vuis),hω)(hσ,ht) ≡
127a   []
127b   { let m = ba_r_ch[h_ui,v_ui] ? in
127c     assert: m=(_,atHub(_,h_ui,_))
127d     let ht’ = ht † [h_ui ↦ ⟨m⟩ht(h_ui)] in

```

127e $\text{hub}_{h_{ui}}(h_{ui}, (l_{ui}, v_{ui}), (h\omega))(h\sigma, ht')$
 127f $| v_{ui}:V_UI \cdot v_{ui} \in v_{uis} \text{ end end } \}$
 127g **post:** $\forall v_{ui}:V_UI \cdot v_{ui} \in \text{dom } ht' \Rightarrow \text{time_ordered}(ht'(v_{ui}))$

Link Behaviour

128 The link behaviour non-deterministically, externally offers
 129 to accept timed vehicle positions —
 130 which will be on the link, from some vehicle, v_{ui} .
 131 The timed vehicle link position is appended to the front of that vehicle's entry in the link's traffic table;
 132 whereupon the link proceeds as a link behaviour with the updated link traffic table.
 133 The link behaviour offers to accept from any vehicle.
 134 A **post** condition expresses what is really a **proof obligation**: that the link traffic, lt' satisfies the **axiom** of the
 enduring link traffic attribute Item 117 Pg. 58.

128 $\text{link}_{l_{ui}}(l_{ui}, (_ , (huis, vuis), _), l\omega)(l\sigma, lt) \equiv$
 128 \square
 129 $\{ \text{let } m = \text{ba_r_ch}[l_{ui}, v_{ui}] ? \text{ in}$
 130 $\quad \text{assert: } m = (_ , \text{onLink}(_ , l_{ui}, _ , _))$
 131 $\quad \text{let } lt' = lt \dagger [l_{ui} \mapsto \langle m \rangle \hat{\sim} lt(l_{ui})] \text{ in}$
 132 $\quad \text{link}_{l_{ui}}(l_{ui}, (huis, vuis), h\omega)(h\sigma, lt')$
 133 $\quad | v_{ui}:V_UI \cdot v_{ui} \in v_{uis} \text{ end end } \}$
 134 **post:** $\forall v_{ui}:V_UI \cdot v_{ui} \in \text{dom } lt' \Rightarrow \text{time_ordered}(lt'(v_{ui}))$

Bus Company Behaviour

We model bus companies very rudimentary. Bus companies keep a fleet of buses. Bus companies create, maintain, distribute bus time tables. Bus companies deploy their buses to honor obligations of their bus time tables. We shall basically only model the distribution of bus time tables to buses. We shall not cover other aspects of bus company management, etc.

135 Bus companies non-deterministically, internally, chooses among
 a updating their bus time tables
 b whereupon they resume being bus companies, albeit with a new bus time table;
 136 “interleaved” with
 a offering the current time-stamped bus time table to buses which offer willingness to received them
 b whereupon they resume being bus companies with unchanged bus time table.

87 $\text{bus_company}_{bc_{ui}}(bc_{ui}, (_ , \text{buis}, _))(btt) \equiv$
 135a $(\text{let } btt' = \text{update}(btt, \dots) \text{ in}$
 135b $\quad \text{bus_company}_{bc_{ui}}(bc_{ui}, (_ , \text{buis}, _))(btt') \text{ end })$
 136 \square
 136a $(\square \{ bc_b_ch[bc_{ui}, b_{ui}] ! btt \mid b_{ui}:B_UI \cdot b_{ui} \in \text{buis}$
 136b $\quad \text{bus_company}_{bc_{ui}}(bc_{ui}, (_ , \text{buis}, _))(\text{record_TIME}(), btt) \})$

We model the interface between buses and their owning companies — as well as the interface between buses and the road net, the latter by almost “carbon-copying” all elements of the automobile behaviour(s).

137 The bus behaviour chooses to either
 a accept a (latest) time-stamped buss time table from its bus company –
 b where after it resumes being the bus behaviour now with the updated bus time table.
 138 or, non-deterministically, internally,
 a based on the bus position
 i if it is at a hub then it behaves as prescribed in the case of automobiles at a hub,
 ii else, it is on a link, and then it behaves as prescribed in the case of automobiles on a link.

```

137 busbui(bui,(,(bcui,ruis),_))(ln,btt,bpos) ≡
137a (let btt' = bbcch[bui,bcui] ? in
137b busbui(bui,({,(bcui,ruis),{}}))(ln,btt',bpos) end)
138 []
138a (case bpos of
138(a)i atH(flui,hui,tlui) →
138(a)i atHbusbui(bui,(,(bcui,ruis),_))(ln,btt,bpos),
138(a)ii aonL(fhui,lui,f,thui) →
138(a)ii onLbusbui(bui,(,(bcui,ruis),_))(ln,btt,bpos)
138a end)

```

Bus Behaviour at a Hub The $\text{atH}_{\text{bus}_{b_{ui}}}$ behaviour definition is a simple transcription of the $\text{automobile}_{a_{ui}}$ (atH) behaviour definition: mereology expressions being changed from to , , programmed attributes being changed from $\text{atH}(fl_{ui},h_{ui},tl_{ui})$ to $(ln,btt,\text{atH}(fl_{ui},h_{ui},tl_{ui}))$, channel references a_{ui} being replaced by b_{ui} , and behaviour invocations renamed from $\text{automobile}_{a_{ui}}$ to $\text{bus}_{b_{ui}}$. So formula lines 91–126d below presents “nothing new”!

```

138(a)i atHbusbui(bui,(,(bcui,ruis),_))
138(a)i (ln,btt,\text{atH}(flui,hui,tlui)) ≡
91 (barch[bui,hui] ! (recordTIME() ,\text{atH}(flui,hui,tlui)));
92 busbui(bui,({,(bcui,ruis),{}}))(ln,btt,bpos)
137a []
93a (let ({fhui,thui},ruis')=mereoL(∅(tlui)) in
93a assert: fhui=hui ∧ ruis=ruis'
90 let onl = (tlui,hui,0,thui) in
93b (barch[bui,hui] ! (recordTIME() ,onl)) ||
93b (barch[bui,tlui] ! (recordTIME() ,onl));
93c busbui(bui,({,(bcui,ruis),{}}))
93c (ln,btt,onl(onl)) end end )
126c []
126d stop

```

Bus Behaviour on a Link

The $\text{onL}_{\text{bus}_{b_{ui}}}$ behaviour definition is a similar simple transcription of the $\text{automobile}_{a_{ui}}$ (onL) behaviour definition. So formula lines 91–126d below presents “nothing new”!

139 – this is the “almost last formula line”!

```

138(a)ii onLbusbui(bui,(,(bcui,ruis),_))
138(a)ii (ln,btt,bpos:onL(fhui,lui,f,thui)) ≡
91 (barch[bui,hui] ! (recordTIME() ,bpos);
92 busbui(bui,({,(bcui,ruis),{}}))(ln,btt,bpos)
137a []
126(b)i (if notyetathub(f)
126(b)i then
126(b)i1 (let incr = increment(f) in
90 let onl = (tlui,hui,incr,thui) in
126(b)i2 barch[lui,bui] ! onl(onl) ;
126(b)i3 busbui(bui,({,(bcui,ruis),{}}))
126(b)i3 (ln,btt,onl(onl))
126(b)i end end)
126(b)ii else
126(b)ii1 (let nlui:LUI•nxtlui∈mereoH(∅(thui)) in
126(b)ii2 barch[thui,bui]!\text{atH}(lui,thui,nxtlui) ;
126(b)ii3 busbui(bui,({,(bcui,ruis),{}}))
126(b)ii3 (ln,btt,\text{atH}(lui,hui,nxtlui))
126(b)ii1 end)end)

```

126c □
 126d stop

1.10 Closing

Domain models abstract some reality. They do not pretend to capture all of it.

1.10.1 What Have We Achieved ?

A step-wise *method*, its *principles*, *techniques*, and a series of *languages* for the rigorous development of domain models has been presented. A seemingly large number of domain concepts has been established: *entities*, *endurants* and *perdurants*, *discrete* and *continuous* endurants, *structure*, *part*, *component* and *material* endurants, *living species*, *plants*, *animals*, *humans* and *artefacts*, *unique identifiers*, *mereology* and *attributes*.

It is shown how CSP *channels* can be calculated from endurant mereologies, and how the form of *behaviour arguments* can be calculated from respective attribute categorisations.

The domain concepts outlined above form a *domain ontology* that applies to a wide variety of domains.

The Transcendental Deduction: A concept of *transcendental deduction* has been introduced. It is used to justify the interpretation of *endurant parts* as *perdurant behaviours* – à la CSP. The interpretation of *endurant parts* as *perdurant behaviours* represents a *transcendental deduction* – and must, somehow, be rationally justified. the justification is here seen as exactly that: a *transcendental deduction*. We claim that when, as an example, programmers, in thinking about or in explaining their code, anthropomorphically⁶⁴, say that “*the program does so and so*” they ‘perform’ and transcendental deduction. We refer to the forthcoming [73, Philosophical Issues in Domain Modeling].

- This concept should be studied further: *Transcendental Deduction in Computing Science*.

Living Species: The concept of *living species* has been introduced, but it has not been “sufficiently” studied, that is, we have, in Sect. 1.5.3 on Page 32, hinted at a number of ‘living species’ notions: *causality of purpose* et cetera, but no hints has been given as to the kind of attributes that *living species*, especially *humans* give rise to.

- This concept should be studied further: *Attributes of Living Species in Computing Science*.

Intentional “Pull”: A new concept of *intentional “pull”* has been introduced. It applies, in the form of attributes, to humans and artifacts. It “corresponds”, in a way, to *gravitational pull*; that concept invites further study. The pair of gravitational pull and intentional “pull” appears to lie behind the determination of the mereologies of parts; that possibility invites further study.

- This concept should be studied further: *Intentional “Pull” in Computing Science*.

What Can Be Described ? When you read the texts that explain when phenomena can be considered entities, entities can be considered endurants or perdurants, endurants can be considered discrete or continuous, discrete endurants can be considered structures, parts or components, et cetera, then you probably, expecting to read a technical/scientific paper, realise that those explanations are not precise in the sense of such papers.

Many of our definitions are taken from [170, The Oxford Shorter English Dictionary] and from the Internet based [263, The Stanford Encyclopedia of Philosophy].

In technical/scientific papers definitions are expected to be precise, but can be that only if the definer has set up, beforehand, or the reported work is based on a precise, in our case mathematical framework. That can not be done here. There is no, a priori given, model of the domains we are interested in. This raises the more general question, such as we see it: “*which are the absolutely necessary and unavoidable bases for describing the world ?*” This is a question of philosophy. We shall not develop the reasoning here.

⁶⁴ Anthropomorphism is the attribution of human traits, emotions, or intentions to non-human entities.

Some other issues are to be further studied. (i) When to use *physical mereologies* and when to apply *conceptual mereologies*, cf. final paragraph of Sect. 1.5.2 on Page 28. (ii) How do we know that the categorisation into unique identification, mereology and attributes embodies all internal qualities; could there be a fourth, etc. ? (iii) Is *intent* an attribute, or does it “belong” to a fourth internal quality category, or a fifth ? (iv) It seems that most of what we first thought off as natural parts really are materials: geographic land masses, etc. – subject, still, to the laws of physics: geo-physics.

- We refer to the forthcoming study [73, Philosophical Issues in Domain Modeling] based on [238, 239, 241, 245].

A Conjecture: It could be interesting to study *under what circumstances, including for which kind of behaviours, we can postulate the following:*

Conjecture: Parts \cong Behaviours

To every part there is a behaviour, and to every suitably expressed behaviour there is a part.

We shall leave this study to the reader !

The Contribution: In summary we have shown that the domain analysis & description calculi form a sound, consistent and complete approach to domain modelling, and that this approach takes its “resting point” in Kai Sørlander’s Philosophy.

1.10.2 The Four Languages of Domain Analysis & Description

Usually mathematics, in many of its shades and forms are deployed in *describing* properties of nature, as when pursuing physics, Usually the formal specification languages of *computer & computing science* have a precise semantics and a consistent proof system. To have these properties those languages must deal with *computable objects*. *Domains are not computable*.

So we revert, in a sense, to mathematics as our specification language. Instead of the usual, i.e., the classical style of mathematics, we “couch” the mathematics in a style close to RSL [131, 27]. We shall refer to this language as RSL⁺. Main features of RSL⁺ evolves in this chapter, mainly in Sect. 1.8.3.

Here we shall make it clear that we need three languages: (i) an **analysis language**, (ii) a **description language**, i.e., RSL⁺, and (iii) the language of explaining domain analysis & description, (iv) in modelling “the fourth” language, the domain, its syntax and some abstract semantics.

The Analysis Language:

Use of the *analysis language* is not written down. It consists of a number of single, usually *is_* or *has_*, prefixed *domain analysis prompt* and *domain description prompt* names. The **domain analysis prompts** are:

The Analysis Prompts

- | | |
|------------------------------------|---|
| a. <i>is_ entity</i> , 10 | l. <i>is_ living_ species</i> , 15 |
| b. <i>is_ enduring</i> , 11 | m. <i>is_ plant</i> , 15 |
| c. <i>is_ perdurant</i> , 11 | n. <i>is_ animal</i> , 16 |
| d. <i>is_ discrete</i> , 11 | o. <i>is_ human</i> , 16 |
| e. <i>is_ continuous</i> , 11 | p. <i>has_ components</i> , 17 |
| f. <i>is_ physical_ part</i> , 12 | q. <i>has_ materials</i> , 17 |
| g. <i>is_ living_ species</i> , 12 | r. <i>is_ artefact</i> , 17 |
| h. <i>is_ structure</i> , 13 | s. <i>observe_ enduring_ sorts</i> , 18 |
| i. <i>is_ part</i> , 14 | t. <i>has_ concrete_ type</i> , 21 |
| j. <i>is_ atomic</i> , 15 | u. <i>has_ mereology</i> , 26 |
| k. <i>is_ composite</i> , 15 | v. <i>attribute_ types</i> , 29 |

They apply to phenomena in the domain, that is, to “the world out there”! Except for `observe_endurants` and attribute types these queries result in truth values; `observe_endurants` results in the *domain scientist cum engineer* noting down, in memory or in typed form, suggestive names [of endurant sorts]; and `attribute_types` results in suggestive names [of attribute types]. The truth-valued queries directs, as we shall see, the *domain scientist cum engineer* to either further analysis or to “issue” some *domain description prompts*. The ‘name’-valued queries help the human analyser to formulate the result of **domain description prompts**:

The Description Prompts

[1] <code>observe_endurant_sorts</code> , 19	[5] <code>observe_unique_identifier</code> , 25
[2] <code>observe_part_type</code> , 21	[6] <code>observe_mereology</code> , 27
[3] <code>observe_component_sorts</code> , 22	[7] <code>observe_attributes</code> , 29
[4] <code>observe_material_sorts</code> , 24	

Again they apply to phenomena in the domain, that is, to “the world out there”! In this case they result in $RSL^+ \text{Text}$!

The Description Language:

The **description language** is RSL^+ . It is a basically applicative subset of RSL [131, 27], that is: no assignable variables. Also we omit RSL’s elaborate *scheme*, *class*, *object* notions.

The Description Language Primitives

• <i>Structures, Parts, Components and Materials:</i>	
∞ obs_E ,	dfn. 1, [o] pg. 19
∞ obs_T : P,	dfn. 2, [t ₂] pg. 21
• <i>Part and Component Unique Identifiers:</i>	
∞ uid_P ,	dfn. 5, [u] pg. 25
• <i>Part Mereologies:</i>	
∞ obs_mereo_P ,	dfn. 6, [m] pg. 27
• <i>Part and Material Attributes:</i>	
∞ attr_A_i ,	dfn. 7, [a] pg. 30

We refer, generally, to all these functions as observer functions. They are defined by the analyser cum describer when “applying” description prompts. That is, they should be considered user-defined. In our examples we use the non-bold-faced observer function names.

The Language of Explaining Domain Analysis & Description:

In explaining the *analysis & description prompts* we use a natural language which contains terms and phrases typical of the technical language of *computer & computing science*, and the language of *philosophy*, more specifically *epistemology* and *ontology*. The reason for the former should be obvious. The reason for the latter is given as follows: We are, on one hand, dealing with real, actual segments of domains characterised by their basis in nature, in economics, in technologies, etc., that is, in informal “worlds”, and, on the other hand, we aim at a formal understanding of those “worlds”. There is, in other words, the task of explaining how we observe those “worlds”, and that is what brings us close to some issues well-discussed in *philosophy*.

The Language of Domains:

We consider a domain through the *semiotic looking glass* of its *syntax* and its *semantics*; we shall not consider here its possible *pragmatics*. By “*its syntax*” we shall mean the form and “*contents*”, i.e., the *external* and *internal qualities* of the *endurants* of the domain, i.e., those *entities* that endure. By “*its semantics*” we shall, by a *transcendental deduction*, mean the *perdurants*: the *actions*, the *events*, and the *behaviours* that center on the the endurants and that otherwise characterise the domain.

An Analysis & Description Process:

It will transpire that the domain analysis & description process can be informally modeled as follows:

Program Schema: A Domain Analysis & Description Process

```

type
  V = Part_VAL | Komp_VAL | Mat_VAL
variable
  new:V-set := {uod:UoD} ,
  gen:V-set := {} ,
  txt:Text := {}
value
  discover_sorts: Unit → Unit
  discover_sorts() ≡
    while new ≠ {} do
      let v:V • v ∈ new in
        new := new \ {v} || gen := gen ∪ {v} ;
        is_part(v) →
          ( is_atomic(v) → skip ,
            is_composite(v) →
              let {e1:E1,e:E2,...,en:En} = observe_endurants(v) in
                new := new ∪ {e1,e,...,en} ; txt := txt ∪ observe_endurant_sorts(e) end ,
              has_concrete_type(v) →
                let {s1,s2,...,sm} = new_sort_values(v) in
                  new := new ∪ {s1,s2,...,sm} ; txt := txt ∪ observe_part_type(v) end ) ,
              has_components(v) → let {k1:K1,k2:K2,...,kn:Kn} = observe_components(v) in
                new := new ∪ {k1,k2,...,kn} ; txt := txt ∪ observe_component_sorts(v) end ,
              has_materials(v) → txt := txt ∪ observe_material_sorts(v) ,
              is_structure(v) → ... EXERCISE FOR THE READER!
            end
          end
    end

  discover_uids: Unit → Unit
  discover_uids() ≡
    for ∀ v:(PVAL|KVAL) • v ∈ gen
      do txt := txt ∪ observe_unique_identifier(v) end
  discover_mereologies: Unit → Unit
  discover_mereologies() ≡
    for ∀ v:PVAL • v ∈ gen
      do txt := txt ∪ observe_mereology(v) end
  discover_attributes: Unit → Unit
  discover_attributes() ≡
    for ∀ v:(PVAL|MVAL) • v ∈ gen
      do txt := txt ∪ observe_attributes(v) end

  analysis+description: Unit → Unit

```

```
analysis+description() ≡
  discover_sorts(); discover_uids(); discover_mereologies(); discover_attributes()
```

Possibly duplicate texts “disappear” in txt – the output text.

1.10.3 Relation to Other Formal Specification Languages

In this contribution we have based the analysis and description calculi and the specification texts emanating as domain descriptions on RSL [131]. There are other formal specification languages:

- **Alloy** [156],
- **B** (etc.) [1],
- **CafeObj** [129],
- **CASL** [113],
- **VDM** [90, 91, 126],
- **Z** [260],

to mention a few. Two conditions appear to apply for any of these other formal specification languages to become a basis for analysis and description calculi similar to the ones put forward in the current chapter: (i) it must be possible, as in RSL, to define and express sorts, i.e., *further undefined types*, and (ii) it must be possible, as with RSL’s “built-in” **CSP** [148] in some form or another, to define and express concurrency. Insofar as these and other formal languages can satisfy these two conditions, they can certainly also be the basis for domain analysis & description.

We do not consider **Coq** [118, 153, 197]⁶⁵, **CSP** [148] **The Duration Calculus** [264] nor **TLA+** [167] as candidates for expressing full-fledged domain descriptions. Some of these formal specification languages, like **Coq**, are very specifically oriented towards proofs (of properties of specifications). Some, like **The Duration Calculus** and **CSP**, go very well in hand with other formal specification languages like **VDM**, **RAISE**⁶⁶ and **Z**. It seems, common to these languages, that, taken in isolation, they can be successfully used for the development and proofs of properties of algorithms and code for, for example safety-critical and embedded systems. But our choice (of not considering) is not a “hard nailed” one! Also less formal, usually computable, languages, like **Scala** [<https://www.scala-lang.org/>] or **Python** [<https://www.python.org/>], can, if they satisfy criteria (i-ii), serve similarly. We refer, for a more general discussion – of issues related to the choice of other formal language being the basis for domain analysis & description – to [89, 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities] for a general discussion that touches upon the issue of formal, or near-formal, specification languages.

1.10.4 Two Frequently Asked Questions

How much of a DOMAIN must or should we ANALYSE & DESCRIBE ? When this question is raised, after a talk of mine over the subject, and by a colleague researcher & scientist I usually reply: *As large a domain as possible!* This reply is often met by this *comment* (from the audience) *Oh ! No, that is not reasonable!* To me that comment shows either or both of: the questioner was not asking as a researcher/scientist, but as an engineer. Yes, an engineer needs only analyse & describe up to and slightly beyond the “border” of the domain-of-interest for a current software development – but a researcher cum scientist is, of course, interested not only in a possible requirements engineering phase beyond domain engineering, but is also curious about the larger context of the domain, in possibly establishing a proper domain theory, etc.

How, then, should a domain engineer pursue DOMAIN MODELLING ? My answer assumes a “state-of-affairs” of domain science & engineering in which domain modelling is an established subject, i.e., where the domain analysis & description topic, i.e., its methodology, is taught, where there are “text-book”

⁶⁵ <http://doi.org/10.5281/zenodo.1028037>

⁶⁶ A variant of **CSP** is thus “embedded” in **RSL**

examples from relevant fields – that the domain engineers can rely on, and in whose terminology they can communicate with one another; that is, there is an acknowledged *body of knowledge*. My answer is therefore: the domain engineer, referring to the relevant *body of knowledge*, develops a domain model that covers the domain and the context on which the software is to function, just, perhaps covering a little bit more of the context, than possibly necessary — just to be sure. Until such a “state-of-affairs” is reached the domain model developer has to act both as a domain scientist and as a domain engineer, researching and developing models for rather larger domains than perhaps necessary while contributing also to the **domain science & engineering body of knowledge**.

1.10.5 On How to Pursue Domain Science & Engineering

We set up a dogma and discuss a ramification. One thing is the doctrine, the method for domain analysis & description outlined in this chapter. Another thing is its practice. I find myself, when experimentally pursuing the modelling of domains, as, for example, reported in [23, 88, 26, 215, 249, 57, 56, 33, 21, 69, 67, 95, 72, 74], **that I am often not following the doctrine!** That is: (i) in not first, carefully, exploring parts, components and materials, the external properties, (ii) in not then, again carefully settling issues of unique identifiers, (iii) then, carefully, the issues of mereology, (iv) followed by careful consideration of attributes, then the transcendental deduction of behaviours from parts; (v) carefully establishing channels: (v.i) their message types, and (v.ii) declarations, (vi) followed by the careful consideration of behaviour signatures, systematically, one for each transcendently deduced part, (vii) then the careful definition of each of all the deduced behaviours, and, finally, (viii) the definition of the overall system initialisation. No, instead I falter, get diverted into exploring “*this & that*” in the domain exploration. And I get stuck. When despairing I realise that I must “*slavically*” follow the doctrine. When reverting to the strict adherence of the doctrine, I find that I, very quickly, find my way, and the domain modelling get’s *unstuck!* I remarked this situation to a dear friend and colleague. His remark stressed what was going on: the **creative engineer took possession**, the **exploring**, sometimes **sceptic** scientist **entered the picture**, the well-trained engineer **lost ground in the realm of imagination**. But perhaps, in the interest of **innovation etc.** it is necessary to be **creative** and **sceptic** and **loose ground** – for a while! I knew that, but had sort-of-forgotten it! I thank Ole N. Oest for this observation.

The lesson is: *waver between adhering to the method and being innovative, curious – a dreamer!*

1.10.6 Domain Science & Engineering

The present chapter is but one in a series on the topic of *domain science & engineering*. With this chapter the author expects to have laid a foundation. With the many experimental case studies, referenced in Example *Universes of Discourse* Page 9, the author seriously think that reasonably convincing arguments are given for this *domain science & engineering*. We comment on some previous publications: [45, 76] explores additional views on analysing & describing domains, in terms of *domain facets: intrinsics, support technologies, rules & regulations, scripts, management & organisation, and human behaviour*. [41, 78] explores relations between Stanisław Leśniewski’s mereology and ours. [35, 66] shows how to rigorously transform domain descriptions into software system requirements prescriptions. [62] explores relations between the present domain analysis & description approach and issues of *safety critical software design*. [65] discusses various interpretations of domain models: as bases for demos, simulators, real system monitors and real system monitor & controllers. [84] is a compendium of reports around the management and engineering of software development based in domain analysis & description. These reports were the result of a year at JAIST: Japan Institute of Science & Technology, Ishikawa, Japan.

1.10.7 Comparison to Related Work⁶⁷

⁶⁷ This section was not in [80]. It is a slightly edited version of [70, Sect. 5.3].

We shall now compare the approach of this chapter to a number of techniques and tools that are somehow related — if only by the term ‘domain’ ! Common to all the “other” approaches is that none of them presents a prompt calculus that help the domain analyser elicit a, or the, domain description. Figure 1.4 on Page 8 shows the tree-like structuring of what modern day AI researchers cum ontologists would call *an upper ontology*.

General

Two related approaches to structuring domain understanding will be reviewed.

0: Ontology Science & Engineering:

Ontologies are “*formal representations of a set of concepts within a domain and the relationships between those concepts*” — expressed usually in some logic. Ontology engineering [14] construct ontologies. Ontology science appears to mainly study structures of ontologies, especially so-called *upper ontology* structures, and these studies “waver” between *philosophy* and *information science*. Internet published ontologies usually consists of thousands of logical expressions. These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology research groups and processed by various tools. There does not seem to be a concern for “deriving” such ontologies into requirements for software. Usually ontology presentations either start with the presentation of, or makes reference to its reliance on, an *upper ontology*. The term ‘ontology’ has been much used in connection with automating the design of various aspects WWW applications [256]. Description Logic [8] has been proposed as a language for the Semantic Web [9].

The interplay between endurants and perdurants is studied in [18]. That study investigates axiom systems for two ontologies. One for endurants (SPAN), another for perdurants (SNAP). No examples of descriptions of specific domains are, however, given, and thus no specific techniques nor tools are given, method components which could help the engineer in constructing specific domain descriptions. [18] is therefore only relevant to the current chapter insofar as it justifies our emphasis on endurant versus perdurant entities. The interplay between endurant and perdurant entities and their qualities is studied in [161]. In our study the term *quality* is made specific and covers the ideas of external and internal qualities. External qualities focus on whether endurant or perdurant, whether part, component or material, whether action, event or behaviour, whether atomic or composite part, etcetera. Internal qualities focus on unique identifiers (of parts), the mereology (of parts), and the attributes (of parts, components and materials), that is, of endurants. In [161] the relationship between universals (types), particulars (values of types) and qualities is not “restricted” as in the TripTych domain analysis, but is axiomatically interwoven in an almost “recursive” manner. Values [of types (‘quantities’ [of ‘qualities’])] are, for example, seen as sub-ordinated types; this is an ontological distinction that we do not make. The concern of [161] is also the relations between qualities and both endurant and perdurant entities, where we have yet to focus on “qualities”, other than signatures, of perdurants. [161] investigates the quality/quantity issue wrt. endurance/perdurance and poses the questions: [b] are non-persisting quality instances enduring, perduring or neither? and [c] are persisting quality instances enduring, perduring or neither? and arrives, after some analysis of the endurance/perdurance concepts, at the answers: [b'] non-persisting quality instances are neither enduring nor perduring particulars (i.e., entities), and [c'] persisting quality instances are enduring particulars. Answer [b'] justifies our separating enduring and perduring entities into two disjoint, but jointly “exhaustive” ontologies. The more general study of [161] is therefore really not relevant to our prompt calculi, in which we do not speculate on more abstract, conceptual qualities, but settle on external endurant qualities, on the *unique identifier*, *mereology* and *attribute* qualities of endurants, and the simple relations between endurants and perdurants, specifically in the relations between *signatures* of actions, events and behaviours and the endurant sorts, and especially the relation between parts and behaviours.. That is, the TripTych approach to ontology, i.e., its domain concept, is not only model-theoretic, but, we risk to say, radically different. The concerns of TripTych domain science & engineering is based on that of algorithmic engineering. The domains to

which we are applying our analysis & description tools and techniques are spatio-temporal, that is, can be observed, physically; this is in contrast to such conceptual domains as various branches of mathematics, physics, biology, etcetera. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of, but not “in” the domain. The TripTych form of domain science & engineering differs from conventional *ontological engineering* in the following, essential ways: The TripTych domain descriptions rely essentially on a “built-in” *upper ontology*: types, abstract as well as model-oriented (i.e., concrete) and actions, events and behaviours. Domain science & engineering is not, to a first degree, concerned with modalities, and hence do not focus on the modeling of knowledge and belief, necessity and possibility, i.e., alethic modalities, epistemic modality (certainty), promise and obligation (deontic modalities), etcetera.

The TripTych emphasis is on the method for constructing descriptions. It seems that publications on ontological engineering, in contrast, emphasise the resulting ontologies. The papers on ontologies are almost exclusively *computer science* (i.e., *information science*) than *computing science* papers.

The next section overlaps with the present section.

1: Knowledge Engineering:

The concept of *knowledge* has occupied philosophers since Plato. No common agreement on what ‘knowledge’ is has been reached. From [170, 6, 184, 248] we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works*. The seminal reference here is [122]. The aim of *knowledge engineering* was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [125] *knowledge engineering* is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. *Knowledge engineering* focus on continually building up (acquire) large, shared data bases (i.e., *knowledge bases*), their continued maintenance, testing the validity of the stored ‘knowledge’, continued experiments with respect to *knowledge representation*, etcetera. *Knowledge engineering* can, perhaps, best be understood in contrast to *algorithmic engineering*: In the latter we seek more-or-less conventional, usually *imperative programming language* expressions of algorithms *whose algorithmic structure embodies the knowledge required to solve the problem being solved by the algorithm*. The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts: *a collection that “mimics” the semantics of, say, the imperative programming language, a collection that formulates the problem, and a collection that constitutes the knowledge particular to the problem*. We refer to [92]. Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of the domain.

Finally, the domains to which we are applying ‘our form of’ domain analysis are domains which focus on spatio-temporal phenomena. That is, domains which have concrete renditions: air traffic, banks, container lines, manufacturing, pipelines, railways, road transport, stock exchanges, etcetera. In contrast one may claim that the domains described in classical ontologies and knowledge representations are mostly conceptual: mathematics, physics, biology, etcetera.

Specific

2: Database Analysis:

There are different, however related “schools of database analysis”. DSD: the Bachman (or data structure) diagram model [10]; RDM: the relational data model [112]; and ER: entity set relationship model [106] “schools”. DSD and ER aim at graphically specifying database structures. Codd’s RDM simplifies the data models of DSD and ER while offering two kinds of languages with which to operate on RDM databases: SQL

and Relational Algebra. All three “schools” are focused more on data modeling for databases than on domain modeling both enduring and perdurant entities.

3: Domain Analysis:

Domain analysis, or *product line analysis* (see below), as it was then conceived in the early 1980s by James Neighbors [190], is the analysis of related software systems in a domain to find their common and variable parts. This form of domain analysis turns matters “upside-down”: it is the set of software “systems” (or packages) that is subject to some form of inquiry, albeit having some domain in mind, in order to find common features of the software that can be said to represent a named domain.

In this section we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Díaz’s approach [204, 205, 206]. Firstly, our understanding of *domain analysis* basically coincides with Prieto-Díaz’s. Secondly, in, for example, [204], Prieto-Díaz’s domain analysis is focused on the very important stages that precede the kind of *domain modeling* that we have described: major concerns are *selection of what appears to be similar, but specific entities, identification of common features, abstraction of entities and classification*. *Selection* and *identification* is assumed in our approach, but we suggest to follow the ideas of Prieto-Díaz. *Abstraction* (from values to types and signatures) and *classification* into parts, materials, actions, events and behaviours is what we have focused on. All-in-all we find Prieto-Díaz’s work very relevant to our work: relating to it by providing guidance to pre-modeling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Díaz does mention a need for *domain specific languages*, he does not show examples of *domain descriptions* in such DSLs. We, of course, basically use mathematics as the DSL. In our approach we do not consider requirements, let alone software components, as do Prieto-Díaz, but we find that that is not an important issue.

4: Domain Specific Languages:

Martin Fowler⁶⁸ defines a *Domain-specific language (DSL)* as a *computer programming language of limited expressiveness focused on a particular domain* [127]. Other references are [183, 246]. Common to [246, 183, 127] is that they define a domain in terms of classes of software packages; that they never really “derive” the DSL from a description of the domain; and that they certainly do not describe the domain in terms of that DSL, for example, by formalising the DSL. In [144] a domain specific language for railway tracks is the basis for verification of the monitoring and control of train traffic on these tracks. Specifications in that domain specific language, DSL, manifested by track layout drawings and signal interlocking tables, are translated into SystemC [135]. [144] thus takes one very specific DSL and shows how to (informally) translate their “programs”, which are not “directly executable”, and hence does not satisfy Fowler’s definition of DSLs, into executable programs. [144] is a great paper, but it is not solving our problem, that of systematically describing any manifest domain. [144] does, however, point a way to search for — say graphical — DSLs and the possible translation of their programs into executable ones. [144] rely on the DSL of that paper. But it does not give an analysis and a description, i.e., a semantics, of the railway system domain. Such a description, in fact any domain analysis & description, such as we advocate, can then be a basis for one or more specific railway domain DSLs.

DSL Dogma: Domain Specific Languages

Our dogma with respect to DSL’s is: The basis for the design of any DSL, $\mathbb{D}\mathbb{S}\mathbb{L}$, must be a domain analysis & description of that domain, for example, as per the method of the present chapter. Based on such a domain description, \mathcal{D} , we can give semantics to $\mathbb{D}\mathbb{S}\mathbb{L}$ and somehow show that that semantics relates to \mathcal{D} .

⁶⁸ <http://martinfowler.com/dsl.html>

5: Feature-oriented Domain Analysis (FODA):

Feature oriented domain analysis (FODA) is a domain analysis method which introduced feature modeling to domain engineering. FODA was developed in 1990 following several U.S. Government research projects. Its concepts have been regarded as “critically advancing software engineering and software reuse.” The US Government–supported report [163] states: “*FODA is a necessary first step*” for software reuse. To the extent that TripTych *domain engineering* with its subsequent *requirements engineering* indeed encourages reuse at all levels: *domain descriptions* and *requirements prescription*, we can only agree. Another source on FODA is [115]. Since FODA “leans” quite heavily on ‘Software Product Line Engineering’ our remarks in that section, next, apply equally well here.

6: Software Product Line Engineering:

Software product line engineering, earlier known as domain engineering, is the entire process of *reusing domain knowledge* in the production of new software systems. Key concerns of software product line engineering are *reuse*, the building of repositories of *reusable software components*, and *domain specific languages* with which to more-or-less automatically build software based on *reusable software components*. These are not the primary concerns of TripTych *domain science & engineering*. But they do become concerns as we move from *domain descriptions* to *requirements prescriptions*. But it strongly seems that *software product line engineering* is not really focused on the concerns of *domain description* — such as is TripTych *domain engineering*. It seems that *software product line engineering* is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [53] puts the ideas of *software product lines* and *model-oriented software development* in the context of the TripTych approach.

7: Problem Frames:

The concept of *problem frames* is covered in [158] Jackson’s prescription for software development focus on the “triple development” of descriptions of the *problem world*, the *requirements* and the *machine* (i.e., the *hardware* and *software*) to be built. Here *domain analysis* means the same as for us: the *problem world analysis*. In the *problem frame* approach the software developer plays three, that is, all the TripTych rôles: *domain engineer*, *requirements engineer* and *software engineer*, “all at the same time”, iterating between these rôles repeatedly. So, perhaps belabouring the point, *domain engineering* is done only to the extent needed by the prescription of *requirements* and the *design* of *software*. These, really are minor points. But in “restricting” oneself to consider only those aspects of the domain which are mandated by the *requirements prescription* and *software design* one is considering a potentially smaller fragment [159] of the domain than is suggested by the TripTych approach. At the same time one is, however, sure to consider aspects of the domain that might have been overlooked when pursuing *domain description development* in the “more general” TripTych approach.

8: Domain Specific Software Architectures (DSSA):

It seems that the concept of DSSA was formulated by a group of ARPA⁶⁹ project “seekers” who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [251]. The [251] definition of *domain engineering* is “*the process of creating a DSSA: domain analysis and domain modeling followed by creating a software architecture and populating it with software components.*” This definition is basically followed also by [185, 235, 180]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the analysis of software components, “per domain”, to identify commonalities within application software, and to then base the idea of *software architecture* on these findings. Thus DSSA turns matter

⁶⁹ ARPA: The US DoD Advanced Research Projects Agency

“upside-down” with respect to TripTych *requirements development* by starting with *software components*, assuming that these satisfy some *requirements*, and then suggesting *domain specific software* built using these components. This is not what we are doing: we suggest, **Chapter 4, From Domain Descriptions to Requirements Prescriptions**, [66], that *requirements* can be “derived” systematically from, and formally related back to *domain descriptions* without, in principle, considering *software components*, whether already existing, or being subsequently developed. Of course, given a *domain description* it is obvious that one can develop, from it, any number of *requirements prescriptions* and that these may strongly hint at shared, (to be) implemented *software components*; but it may also, as well, be the case that two or more *requirements prescriptions* “derived” from the same *domain description* may share no *software components* whatsoever! It seems to this author that had the DSSA promoters based their studies and practice on also using formal specifications, at all levels of their study and practice, then some very interesting insights might have arisen.

9: Domain Driven Design (DDD):

Domain-driven design (DDD)⁷⁰ “is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts; the premise of domain-driven design is the following: placing the project’s primary focus on the core domain and domain logic; basing complex designs on a model; initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.”⁷¹ We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [145], and find that it really does not contribute to new insight into *domains* such as we see them: it is just “plain, good old software engineering cooked up with a new jargon.

10: Unified Modeling Language (UML):

Three books representative of UML are [98, 229, 160]. jacobson@Ivar Jacobson The term *domain analysis* appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the *domain*, stands for entities in the domain such as we understand it, or whether it is wrought up, as in several of the ‘approaches’ treated in this section, to wit, in items [3–5, 7–9] with either *software design* (as it most often is), or *requirements prescription*. Certainly, in UML, in [98, 229, 160] jacobson@Ivar Jacobsons well as in most published papers claiming “adherence” to UML, that domain analysis usually is manifested in some UML text which “models” some *requirements* facet. Nothing is necessarily wrong with that, but it is therefore not really the TripTych form of *domain analysis* with its concepts of abstract representations of enduring and perdurants, with its distinctions between *domain* and *requirements*, and with its possibility of “deriving” *requirements prescriptions* from *domain descriptions*. The UML notion of *class diagrams* is worth relating to our structuring of the domain. Class diagrams appear to be inspired by [10, Bachman, 1969] and [106, Chen, 1976]. It seems that (i) each part sort — as well as other than part sorts — deserves a class diagram (box); and (ii) that (assignable) attributes — as well as other non-part types — are written into the diagram box. Class diagram boxes are line-connected with annotations where some annotations are as per the mereology of the part type and the connected part types and others are not part related. The class diagrams are said to be object-oriented but it is not clear how objects relate to parts as many are rather implementation-oriented quantities. All this needs looking into a bit more, for those who care.

11: Requirements Engineering:

There are in-numerous books and published papers on *requirements engineering*. A seminal one is [254]. I, myself, find [168] full of very useful, non-trivial insight. [119] is seminal in that it brings a number of early contributions and views on *requirements engineering*. Conventional text books, notably [199, 203, 237] all have their “mandatory”, yet conventional coverage of *requirements engineering*. None of them “derive”

⁷⁰ Eric Evans: <http://www.domaindrivendesign.org/>

⁷¹ http://en.wikipedia.org/wiki/Domain-driven_design

requirements from domain descriptions, yes, OK, from domains, but since their description is not mandated it is unclear what “the domain” is. Most of them repeatedly refer to *domain analysis* but since a written record of that *domain analysis* is not mandated it is unclear what “domain analysis” really amounts to. Axel van Laamsweerde’s book [254] is remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between *goals* and *requirements*, also formally. Most of the advices given in [168] can beneficially be followed also in TripTych *requirements development*. Neither [254] or [168] preempts TripTych *requirements development*.

Summary of Comparisons

We find that there are two kinds of relevant comparisons: the concept of ontology, its science more than its engineering, and the *Problem Frame* work of Michael A. Jackson. The ontology work, as commented upon in Item [1] (Pages 70–71), is partly relevant to our work: There are at least two issues: Different classes of domains may need distinct upper ontologies. Our approach admits that there may be different upper ontologies for non-manifest domains such as *financial systems*, etcetera. This seems to warrant at least a comparative study. We have assumed, cf. Sect. 1.5.3, that attributes cannot be separated from parts. [161, Johansson 2005] develops the notion that *persisting quality instances are enduring particulars*. The issue needs further clarification.

Of all the other “comparison” items ([2]–[12]) basically only Jackson’s *problem frames* (Item [8]) and [144] (Item [5]) really take the same view of *domains* and, in essence, basically maintain similar relations between *requirements prescription* and *domain description*. So potential sources of, we should claim, mutual inspiration ought to be found in one-another’s work — with, for example, [136, 159, 144], and the present document, being a good starting point.

But none of the referenced works make the distinction between discrete endurants (parts) and their qualities, with their further distinctions between *unique identifiers*, *mereology* and *attributes*.

And none of them makes the distinction between *parts*, *components* and *materials*. Therefore our contribution can include the mapping of parts into behaviours interacting as per the part mereologies.

1.10.8 Tony Hoare’s Summary on ‘Domain Modelling’

In a 2006 e-mail, in response, undoubtedly to my steadfast – perhaps conceived as stubborn – insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote⁷²:

“There are many unique contributions that can be made by domain modelling.

- 1 The models describe all aspects of the real world that are relevant for any good software design in the area.
They describe possible places to define the system boundary for any particular project.
- 2 They make explicit the preconditions about the real world that have to be made in any embedded software design,
especially one that is going to be formally proved.
- 3 They describe the whole range of possible designs for the software,
and the whole range of technologies available for its realisation.
- 4 They provide a framework for a full analysis of requirements,
which is wholly independent of the technology of implementation.
- 5 They enumerate and analyse the decisions that must be taken earlier or later in any design project,
and identify those that are independent and those that conflict.
Late discovery of feature interactions can be avoided.”

All of these issues were covered in [32, Part IV].

⁷² E-Mail to Dines Bjørner, July 19, 2006

Domain Facets: Analysis & Description

We¹ investigate some principles and techniques for analysing & describing domain facets.

2.1 Introduction

In Chapter 1 we outlined a *method* for analysing &² and describing domains. In this chapter we cover domain analysis & description principles and techniques not covered in Chapter 1. That chapter focused on *manifest domains*. Here we, on one side, go “outside” the realm of *manifest domains*, and, on the other side, cover, what we shall refer to as, *facets*, not covered in Chapter 1.

2.1.1 Facets of Domains

By a **domain facet** we shall understand *one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain* ■ Now, the definition of what a *domain facet* is can seem vague. It cannot be otherwise. The definition is sharpened by the definitions of the specific facets. You can say, that the definition of *domain facet* is the “sum” of the definitions of these specific facets. The specific facets – so far³ – are:

- *intrinsic*s (Sect. 2.2),
- *support technology* (Sect. 2.3),
- *rules & regulations* (Sect. 2.4),
- *scripts* (Sect. 2.5),
- *license languages* (Sect. 2.6),
- *management & organisation* (Sect. 2.7) and
- *human behaviour* (Sect. 2.8).

Of these, the *rules & regulations*, *scripts* and *license languages* are closely related. Vagueness may “pop up”, here and there, in the delineation of facets. It is necessarily so. We are not in a domain of computer

¹ Chapter 2 is primarily based on [76]. which itself was based on publication [45]. Introductory sections are different, but of no real consequence to this thesis. The present chapter represents the with respect to [45]: Unnumbered initial paragraphs of [45] are not present in chapter 2. Sections 1–3 of [45] are basically omitted here. Their contents already, in another form, present in Chapter 1 of this thesis. Section 4.1 of [45] corresponds, roughly, to Sects. 2.2. Example 7, Traffic Signals, of Section 2.3 of the present chapter is new. Section 2.6 of this chapter is new wrt. [45]. Present Sect. 2.7 moved wrt. Sects. 4.4–4.5 of [45]. Example 20 of Sect. 2.7 is new.

² We use the ampersand (logogram), &, in the following sense: Let *A* and *B* be two concepts. By *A and B* we mean to refer to these two concepts. With *A&B* we mean to refer to a composite concept “containing” elements of both *A* and *B*.

³ We write: ‘so far’ in order to “announce”, or hint that there may be other specific facets. The one listed are the ones we have been able to “isolate”, to identify, in the most recent 10-12 years.

science, let alone mathematics, where we can just define ourselves precisely out of any vagueness problems. We are in the domain of (usually) really world facts. And these are often hard to encircle.

2.1.2 Relation to Previous Work

The present chapter is a rather complete rewrite of [45]. The reason for the rewriting is the expected publication of [81]. [45] was finalised already in 2006, 10 years ago, before the analysis & description calculus of [81] had emerged. It was time to revise [45] rather substantially.

2.1.3 Structure of Chapter

The structure of this chapter follows the seven specific facets, as listed above. Each section, 2.2.–2.8., starts by a definition of the *specific facet*, Then follows an analysis of the abstract concepts involved usually with one or more examples – with these examples making up most of the section. We then “speculate” on derivable requirements thus relating the present chapter to [66]. We close each of the sections, 2.2.–2.8., with some comments on how to model the specific facet of that section.

•••

Examples 1–22 of sections 2.2.–2.8. present quite a variety. In that, they reflect the wide spectrum of facets.

•••

More generally, domains can be characterised by intrinsically being *endurant*, or *function*, or *event*, or *behaviour intensive*. Software support for activities in such domains then typically amount to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Other than this brief discourse we shall not cover the “intensity”-aspect of domains in this chapter.

2.2 Intrinsic

- By domain **intrinsic** we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsic initially covering at least one specific, hence named, stakeholder view ■

2.2.1 Conceptual Analysis

The principles and techniques of domain analysis & description, as unfolded in Chapter 1, focused on and resulted in descriptions of the intrinsic of domains. They did so in focusing the analysis (and hence the description) on the basic *endurants* and their related *perdurants*, that is, on those parts that most readily present themselves for observation, analysis & description.

Example 1 Railway Net Intrinsic: We narrate and formalise three railway net intrinsic.

From the view of potential train passengers a railway net consists of lines, $l:L$, with names, $ln:Ln$, stations, $s:S$, with names $sn:Sn$, and trains, $tn:TN$, with names $tnm:Tnm$. A line connects exactly two distinct stations.

```

scheme N0 =
  class
    type
      N, L, S, Sn, Ln, TN, Tnm
    value

```

```

obs_Ls: N → L-set, obs_Ss: N → S-set
obs_Ln: L → Ln, obs_Sn: S → Sn
obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
axiom
...
end

```

N, L, S, Sn and Ln designate nets, lines, stations, station names and line names. One can observe lines and stations from nets, line and station names from lines and stations, pair sets of station names from lines, and lines names (of lines) into and out from a station from stations. Axioms ensure proper graph properties of these concepts.

From the view of *actual train passengers* a railway net — in addition to the above — allows for several lines between any pair of stations and, within stations, provides for one or more platform tracks, tr:Tr, with names, trn:Trn, from which to embark on or alight from a train.

```

scheme N1 = extend N0 with
class
  type
    Tr, Trn
  value
    obs_Tr: S → Tr-set, obs_Trn: Tr → Trn
  axiom
  ...
end

```

The only additions are that of track and track name types, related observer functions and axioms.

From the view of *train operating staff* a railway net — in addition to the above — has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path, p:P, (through a unit) is a pair of connectors of that unit. A state, $\sigma : \Sigma$, of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in any one of a number of states of its state space, $\omega : \Omega$.

```

scheme N2 = extend N1 with
class
  type
    U, C
    P' = U × (C × C)
    P = { | p:P' • let (u,(c,c'))=p in (c,c') ∈ U obs_Ω(u) end | }
    Σ = P-set
    Ω = Σ-set
  value
    obs_Us: (N|L|S) → U-set
    obs-Cs: U → C-set
    obs_Σ: U → Σ
    obs_Ω: U → Ω
  axiom
  ...
end

```

Unit and connector types have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers. ■

Different stakeholder perspectives, not only of intrinsics, as here, but of any facet, lead to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide with the name of a “similar” phenomenon of another perspective, that is, of another model, and so on. If the intention is that the “same” names cover comparable phenomena, then the developer must state the comparison relation.

Example 2 Intrinsic of Switches: *The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch (${}^c_1 Y_c^{c/}$) has three connectors: $\{c, c_1, c/\}$. c is the connector of the common rail from which one can either “go straight” c_1 , or “fork” $c/$ (Fig. 2.1). So we have that a possible state space of such a switch could be ω_{gs} :*

$$\begin{aligned} & \{\{\}, \\ & \{(c, c_1)\}, \{(c_1, c)\}, \{(c, c_1), (c_1, c)\}, \\ & \{(c, c/)\}, \{(c/, c)\}, \{(c, c/), (c/, c)\}, \{(c/, c), (c_1, c)\}, \\ & \{(c, c_1), (c_1, c), (c/, c)\}, \{(c, c/), (c/, c), (c_1, c)\}, \{(c/, c), (c, c_1)\}, \{(c, c/), (c_1, c)\} \} \end{aligned}$$

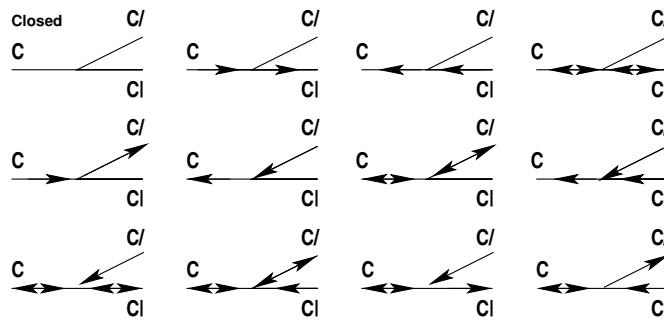


Fig. 2.1. Possible states of a rail switch

The above models a general switch ideally. Any particular switch ω_{ps} may have $\omega_{ps} \subset \omega_{gs}$. Nothing is said about how a state is determined: who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (i.e., invisible, intangible) signals up or down the rail, away from the switch. ■

Example 3 An Intrinsic of Documents: *Think of documents, written, by hand, or typed “onto” a computer text processing system. One way of considering such documents is as follows. First we abstract from the syntax that such a document, or set of more-or-less related documents, or just documents, may have: whether they are letters, with sender and receive addressees, dates written, sent and/or received, opening and closing paragraphs, etc., etc.; or they are books, technical, scientific, novels, or otherwise, or they are application forms, tax returns, patient medical records, or otherwise. Then we focus on the operations that one may perform on documents: their creation, editing, reading, copying, authorisation, “transfer”⁴, “freezing”⁵, and shredding. Finally we consider documents as manifest parts, cf. Chapter 1, Parts, so documents have unique identifications, in this case, changeable mereology, and a number of attributes. The mereology of a document, d , reflects those other documents upon which a document is based, i.e., refers to, and/or refers to d . Among the attributes of a document we can think of (i) a trace of what has happened to a document, i.e., a trace of all the operations performed on “that” document, since and including creation — with that trace, for example, consisting of time-stamped triples of the essence of the operations, the “actor” of the operation (i.e., the operator), and possibly some abstraction of the locale*

⁴ to other editors, readers, etc.

⁵ i.e., prevention of future operations

of the document when operated upon; (ii) a synopsis of what the document text “is all about”, (iii) and some “rendition” of the document text. We refer to experimental technical research report [72]. ■

This view of documents, whether “implementable” or “implemented” or not, is at the basis of our view of license languages (for *digital media*, *health-care* (patient medical record), *documents*, and *transport* (contracts) as that facet is covered in Sect. 2.6.

2.2.2 Requirements

Chapter 5 illustrates requirements “derived” from the intrinsics of a road transport system – as outlined in Chapter 1. So the present chapter has little to add to the subject of requirements “derived” from intrinsics.

2.2.3 On Modeling Intrinsics

Chapter 1 outlines basic principles, techniques and tools for modeling the intrinsics of manifest domains. Modeling the domain intrinsics can often be expressed in property-oriented specification languages (like CafeOBJ [129]), model-oriented specification languages (like Alloy [156], B [1], VDM-SL [90, 91, 126], RSL [131], or Z [260]), event-based languages (like Petri nets or [217] or CSP [148], respectively in process-based specification languages (like MSCs [155], LSCs [142], Statecharts [141], or CSP [148]). An area not well-developed is that of modeling continuous domain phenomena like the dynamics of automobile, train and aircraft movements, flow in pipelines, etc. We refer to [193].

2.3 Support Technologies

- By a domain **support technology** we shall understand ways and means of implementing certain observed phenomena or certain conceived concepts ■

The “ways and means” may be in the form of “soft technologies”: human manpower, see, however, Sect. 2.8, or in the form of “hard” technologies: electro-mechanics, etc. The term ‘implementing’ is crucial. It is here used in the sense that, $\psi\tau$, which is an ‘implementation’ of a *endurant* or *perdurant*, ϕ , is an *extension* of ϕ , with ϕ being an *abstraction* of $\psi\tau$. We strive for the extensions to be *proof theoretic conservative extensions* [174].

2.3.1 Conceptual Analysis

There are [always] basically two approaches the task of analysing & describing the support technology facets of a domain. One either stumbles over it, or one tries to tackle the issue systematically. The “stumbling” approach occurs when one, in the midst of analysing & describing a domain realises that one is tackling something that satisfies the definition of a support technology facet. In the systematic approach to the analysis & description of the support technology facets of a domain one usually starts with a basically intrinsics facet-oriented domain description. We then suggest that the domain engineer “inquires” of every *endurant* and *perdurant* whether it is an *intrinsic entity* or, perhaps a support technology.

Example 4 Railway Support Technology: We give a rough sketch description of possible rail unit switch technologies.

(i) In “ye olde” days, rail switches were “thrown” by manual labour, i.e., by railway staff assigned to and positioned at switches.

(ii) With the advent of reasonably reliable mechanics, pulleys and levers⁶ and steel wires, switches were made to change state by means of “throwing” levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

(iii) This partial mechanical technology then emerged into electro-mechanics, and cabin tower staff was “reduced” to pushing buttons.

(iv) Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another). ■

It must be stressed that Example 4 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electro-mechanics and the human operator interface (buttons, lights, sounds, etc.). An aspect of supporting technology includes recording the state-behaviour in response to external stimuli. We give an example.

Example 5 Probabilistic Rail Switch Unit State Transitions: Figure 2.2 indicates a way of formalising this aspect of a supporting technology. Figure 2.2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd . ■

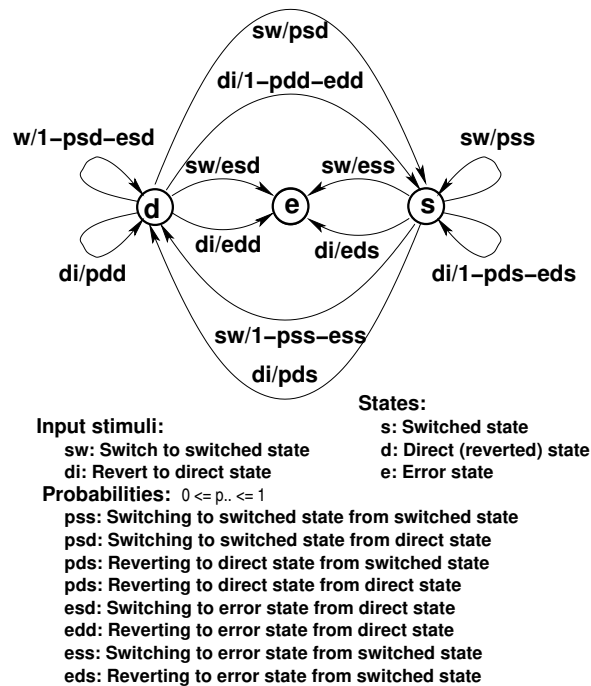


Fig. 2.2. Probabilistic state switching

Example 6 Traffic Signals: A traffic signal represents a technology in support of visualising hub states (transport net road intersection signaling states) and in effecting state changes.

140 A traffic signal, $ts:TS$, is here⁷ considered a part with observable hub states and hub state spaces. Hub states and hub state spaces are programmable, respectively static attributes of traffic signals.

⁶ <https://en.wikipedia.org/wiki/Pulley> and <http://en.wikipedia.org/wiki/Lever>

⁷ In Chapter 1 a traffic signal was an attribute of a hub.

- 141 A hub state space, $h\omega$, is a set of hub states such that each current hub state is in that hubs' hub state space.
- 142 A hub state, $h\sigma$, is now modeled as a set of hub triples.
- 143 Each hub triple has a link identifier l_i ("coming from"), a colour (**red**, **yellow** or **green**), and another link identifier l_j ("going to").
- 144 Signaling is now a sequence of one or more pairs of next hub states and time intervals, $ti:TI$, for example: $\langle (h\sigma_1, ti_1), (h\sigma_2, ti_2), \dots, (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) \rangle, n > 0$. The idea of a signaling is to first change the designated hub to state $h\sigma_1$, then wait ti_1 time units, then set the designated hub to state $h\sigma_2$, then wait ti_2 time units, etcetera, ending with final state σ_n and a (supposedly) long time interval ti_n before any decisions are to be made as to another signaling. The set of hub states $\{h\sigma_1, h\sigma_2, \dots, h\sigma_{n-1}\}$ of $\langle (h\sigma_1, ti_1), (h\sigma_2, ti_2), \dots, (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) \rangle, n > 0$, is called the set of intermediate states. Their purpose is to secure an orderly phase out of green via yellow to red and phase in of red via yellow to green in some order for the various directions. We leave it to the reader to devise proper well-formedness conditions for signaling sequences as they depend on the hub topology.
- 145 A street signal (a semaphore) is now abstracted as a map from pairs of hub states to signaling sequences. The idea is that given a hub one can observe its semaphore, and given the state, $h\sigma$ (not in the above set), of the hub "to be signaled" and the state $h\sigma_n$ into which that hub is to be signal-led "one looks up" under that pair in the semaphore and obtains the desired signaling.

type

140 $TS \equiv H, H\Sigma, H\Omega$

value

141 $attr_H\Sigma: H, TS \rightarrow H\Sigma$

141 $attr_H\Omega: H, TS \rightarrow H\Omega$

type

142 $H\Sigma = \text{Htriple-set}$

142 $H\Omega = \text{H}\Sigma\text{-set}$

143 $\text{Htriple} = \text{LI} \times \text{Colour} \times \text{LI}$

axiom

141 $\forall ts:TS \cdot attr_H\Sigma(ts) \in attr_H\Omega(ts)$

type

143 $\text{Colour} == \text{red} \mid \text{yellow} \mid \text{green}$

144 $\text{Signaling} = (\text{H}\Sigma \times \text{TI})^*$

144 TI

145 $\text{Semaphore} = (\text{H}\Sigma \times \text{H}\Sigma) \rightarrow_{\#} \text{Signalling}$

value

145 $attr_Semaphore: TS \rightarrow \text{Semaphore}$

- 146 We treat hubs as processes with hub state spaces and semaphores as static attributes and hub states as programmable attributes. We ignore other attributes and input/outputs.
- 147 We can think of the change of hub states as taking place based the result of some internal, non-deterministic choice.

value

146. $\text{hub}: \text{HI} \times \text{LI-set} \times (\text{H}\Omega \times \text{Semaphore}) \rightarrow \text{H}\Sigma \text{ in } \dots \text{ out } \dots \text{ Unit}$

146. $\text{hub}(hi, lis, (h\omega, sema))(h\sigma) \equiv$

146. \dots

147. $\square \text{ let } h\sigma': \text{HI} \cdot \dots \text{ in } \text{hub}(hi, lis, (h\omega, sema))(\text{signaling}(h\sigma, h\sigma')) \text{ end}$

146. \dots

146. **pre:** $\{h\sigma, h\sigma'\} \subseteq h\omega$

where we do not bother about the selection of $h\sigma'$.

148 Given two traffic signal, i.e., hub states, $h\sigma_{\text{init}}$ and $h\sigma_{\text{end}}$, where $h\sigma_{\text{init}}$ designates a present hub state and $h\sigma_{\text{end}}$ designates a desired next hub state after signaling.

149 Now *signaling* is a sequence of one or more successful hub state changes.

value

148 `signaling: (HΣ × HΣ) × Semaphore → HΣ → HΣ`

149 `signaling($h\sigma_{\text{init}}, h\sigma_{\text{end}}, \text{sema}$)($h\sigma$) ≡ let sg = sema($h\sigma_{\text{init}}, h\sigma_{\text{end}}$) in signal_sequence(sg)($h\sigma$) end`

149 `pre $h\sigma_{\text{init}} = h\sigma \wedge (h\sigma_{\text{init}}, h\sigma_{\text{end}}) \in \text{dom sema}$`

If a desired hub state change fails (i.e., does not meet the **pre**-condition, or for other reasons (e.g., failure of technology)), then we do not define the outcome of signaling.

149 `signal_sequence($\langle \rangle$)($h\sigma$) ≡ $h\sigma$`

149 `signal_sequence($\langle (h\sigma', ti) \rangle \wedge \text{sg}$)($h\sigma$) ≡ wait(ti); signal_sequence(sg)($h\sigma'$)`

We omit expression of a number of well-formedness conditions, e.g., that the *htriple* link identifiers are those of the corresponding mereology (*lis*), etcetera. The design of the semaphore, for a single hub or for a net of connected hubs has many similarities with the design of interlocking tables for railway tracks [144].

Another example shows another aspect of support technology: Namely that the technology must guarantee certain of its own behaviours, so that software designed to interface with this technology, together with the technology, meets dependability requirements.

Example 7 Railway Optical Gates: *Train traffic* (*itf*:iTF), *intrinsically*, is a total function over some time interval, from time (*t*:T) to continuously positioned (*p*:P) trains (*tn*:TN). Conventional optical gates sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic (*stf*:sTF). Hence the collection of all optical gates, for any given railway, is a partial function from intrinsic to sampled train traffics (*stf*). We need to express quality criteria that any optical gate technology should satisfy — relative to a necessary and sufficient description of a closeness predicate. The following axiom does that:

- For all intrinsic traffics, *itf*, and for all optical gate technologies, *og*, the following must hold: Let *stf* be the traffic sampled by the optical gates. For all time points, *t*, in the sampled traffic, those time points must also be in the intrinsic traffic, and, for all trains, *tn*, in the intrinsic traffic at that time, the train must be observed by the optical gates, and the actual position of the train and the sampled position must somehow be check-able to be close, or identical to one another.

Since units change state with time, $n:\mathbb{N}$, the railway net, needs to be part of any model of traffic.

type

T, TN

P = U*

NetTraffic == net:N trf:(TN \xrightarrow{m} P)

iTF = T → NetTraffic

sTF = T \xrightarrow{m} NetTraffic

oG = iTF $\xrightarrow{\sim}$ sTF

value

close: NetTraffic × TN × NetTraffic $\xrightarrow{\sim}$ Bool

axiom

$\forall \text{itt:iTF, og:OG} \cdot \text{let stt} = \text{og(itt)} \text{ in}$

$\forall t:T \cdot t \in \text{dom stt} \Rightarrow$

$\forall Tn:TN \cdot tn \in \text{dom trf(itt(t))}$

$\Rightarrow tn \in \text{dom trf(stt(t))} \wedge \text{close(itt(t), tn, stt(t))} \text{ end}$

Check-ability is an issue of testing the optical gates when delivered for conformance to the closeness predicate, i.e., to the axiom.

2.3.2 Requirements

Section 4.4 [Extension] of [66] illustrates a possible toll-gate, whose behaviour exemplifies a support technology. So do pumps of a pipe-line system such as illustrated in Examples 24, 29 and 42–44 in [81]. A pump of a pipe-line system gives rise to several forms of support technologies: from the Egyptian Shadoof [irrigation] pumps, and the Hellenic Archimedian screw pumps, via the 11th century Su Song pumps of China⁸, and the hydraulic “technologies” of Moorish Spain⁹ to the centrifugal and gear pumps of the early industrial age, etcetera, The techniques – to mention those that have influenced this author – of [264, 162, 192, 144] appears to apply well to the modeling of support technology requirements.

2.3.3 On Modeling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such the techniques and languages for modeling support technologies resemble those for modeling event and process intensity, while temporal notions are brought into focus. Hence typical modeling notations include event-based languages (like Petri nets [217] or CSP) [148], respectively process-based specification languages (like MSCs, [155], LSCs [142], Statecharts [141], or CSP) [148], as well as temporal languages (like the Duration Calculus and [264] and Temporal Logic of Actions, TLA+) [167]).

2.4 Rules & Regulations

- By a **domain rule** we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duties, respectively when performing their functions ■
- By a **domain regulation** we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention ■

The domain rules & regulations need or may not be explicitly present, i.e., written down. They may be part of the “folklore”, i.e., tacitly assumed and understood.

2.4.1 Conceptual Analysis

Example 8 Trains at Stations:

- Rule: *In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:*
In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.*

Example 9 Trains Along Lines:

- Rule: *In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:*

⁸ https://en.wikipedia.org/wiki/Su_Song

⁹ <http://www.islamicspain.tv/Arts-and-Science/The-Culture-of-Al-Andalus/Hydraulic-Technology.htm>

There must be at least one free sector (i.e., without a train) between any two trains along a line.

- Regulation: *If it is discovered that the above rule is not obeyed, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.*

■

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following: There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules, respectively regulations; and one, Stimulus, exists for describing the form of the [always current] domain action stimuli. A syntactic stimulus, sy_sti , denotes a function, $se_sti:STI: \Theta \rightarrow \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, $sy_rul:Rule$, stands for, i.e., has as its semantics, its meaning, $rul:RUL$, a predicate over current and next configurations, $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express: If the predicate holds then the stimulus will result in a valid next configuration.

type

Stimulus, Rule, Θ
 $STI = \Theta \rightarrow \Theta$
 $RUL = (\Theta \times \Theta) \rightarrow \mathbf{Bool}$

value

meaning: Stimulus \rightarrow STI
 meaning: Rule \rightarrow RUL
 valid: Stimulus \times Rule $\rightarrow \Theta \rightarrow \mathbf{Bool}$
 $valid(sy_sti, sy_rul)(\theta) \equiv meaning(sy_rul)(\theta, (meaning(sy_sti))(\theta))$

A syntactic regulation, $sy_reg:Reg$ (related to a specific rule), stands for, i.e., has as its semantics, its meaning, a semantic regulation, $se_reg:REG$, which is a pair. This pair consists of a predicate, $pre_reg:Pre_REG$, where $Pre_REG = (\Theta \times \Theta) \rightarrow \mathbf{Bool}$, and a domain configuration-changing function, $act_reg:Act_REG$, where $Act_REG = \Theta \rightarrow \Theta$, that is, both involving current and next domain configurations. The two kinds of functions express: If the predicate holds, then the action can be applied. The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function.

type

Reg
 $Rul_and_Reg = Rule \times Reg$
 $REG = Pre_REG \times Act_REG$
 $Pre_REG = \Theta \times \Theta \rightarrow \mathbf{Bool}$
 $Act_REG = \Theta \rightarrow \Theta$

value

interpret: Reg \rightarrow REG

The idea is now the following: Any action (i.e., event) of the system, i.e., the application of any stimulus, may be an action (i.e., event) in accordance with the rules, or it may not. Rules therefore express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort. More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul, sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti , applied in that configuration result in a next configuration, θ' , where $\theta' = (meaning(sy_sti))(\theta)$. Let θ' violate the rule,

$\sim\text{valid}(\text{sy_sti}, \text{sy_rul})(\theta)$, then if predicate part, pre_reg , of the meaning of the regulation, sy_reg , holds in that violating next configuration, $\text{pre_reg}(\theta, (\text{meaning}(\text{sy_sti}))(\theta))$, then the action part, act_reg , of the meaning of the regulation, sy_reg , must be applied, $\text{act_reg}(\theta)$, to remedy the situation.

axiom

```

 $\forall (\text{sy\_rul}, \text{sy\_reg}): \text{Rul\_and\_Reg} \cdot$ 
  let  $\text{se\_rul} = \text{meaning}(\text{sy\_rul}),$ 
     $(\text{pre\_reg}, \text{act\_reg}) = \text{meaning}(\text{sy\_reg})$  in
   $\forall \text{sy\_sti}: \text{Stimulus}, \theta: \Theta \cdot$ 
     $\sim\text{valid}(\text{sy\_sti}, \text{se\_rul})(\theta)$ 
     $\Rightarrow \text{pre\_reg}(\theta, (\text{meaning}(\text{sy\_sti}))(\theta))$ 
     $\Rightarrow \exists n\theta: \Theta \cdot \text{act\_reg}(\theta) = n\theta \wedge \text{se\_rul}(\theta, n\theta)$ 
end

```

It may be that the regulation predicate fails to detect applicability of regulations actions. That is, the interpretation of a rule differs, in that respect, from the interpretation of a regulation. Such is life in the domain, i.e., in actual reality.

2.4.2 Requirements

Implementation of rules & regulations implies *monitoring* and partially *controlling* the states symbolised by Θ in Sect. 2.4.1. Thus some *partial implementation* of Θ must be required; as must some monitoring of states $\theta: \Theta$ and implementation of the predicates *meaning*, *valid*, *interpret*, *pre_reg* and action(s) *act_reg*. The emerging requirements follow very much in the line of support technology requirements.

2.4.3 On Modeling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into “the state”, functions, events, and behaviours. Thus the full spectrum of modeling techniques and notations may be needed. Since rules usually express properties one often uses some combination of axioms and wellformedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus or Temporal Logic of Actions) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in Allard [156], B or event-B [1], RSL [131], VDM-SL [90, 91, 126], and Z [260]). In some cases it may be relevant to model using some constraint satisfaction notation [3] or some Fuzzy Logic notations [253].

2.5 Scripts

- By a **domain script** we shall understand the structured, almost, if not outright, formally expressed, wording of a procedure on how to proceed, one that has legally binding power, that is, which may be contested in a court of law ■

2.5.1 Conceptual Analysis

Rules & regulations are usually expressed, even when informally so, as predicates. Scripts, in their procedural form, are like instructions, as for an algorithm.

Example 10 A Casually Described Bank Script: *Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts. The problem area is that of how repayments of mortgage loans are to be calculated. At any one time*

a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment. We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts. (i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. ■

Example 11 A Formally Described Bank Script: First we must informally and formally define the bank state: There are clients ($c:C$), account numbers ($a:A$), mortgage numbers ($m:M$), account yields ($ay:AY$) and mortgage interest rates ($mi:MI$). The bank registers, by client, all accounts ($\rho:A_Register$) and all mortgages ($\mu:M_Register$). To each account number there is a balance ($\alpha:Accounts$). To each mortgage number there is a loan ($\ell:Loans$). To each loan is attached the last date that interest was paid on the loan.

value

$r, r':\mathbf{Real}$ axiom ...

type

C, A, M, Date

$AY' = \mathbf{Real}, AY = \{ | ay:AY' \cdot 0 < ay \leq r | \}$

$MI' = \mathbf{Real}, MI = \{ | mi:MI' \cdot 0 < mi \leq r' | \}$

$\text{Bank}' = A_Register \times Accounts \times M_Register \times Loans$

$\text{Bank} = \{ | \beta:\text{Bank}' \cdot wf_Bank(\beta) | \}$

$A_Register = C \xrightarrow{\text{fin}} A\text{-set}$

$Accounts = A \xrightarrow{\text{fin}} \text{Balance}$

$M_Register = C \xrightarrow{\text{fin}} M\text{-set}$

$Loans = M \xrightarrow{\text{fin}} (Loan \times \text{Date})$

$Loan, \text{Balance} = P$

$P = \mathbf{Nat}$

Then we must define well-formedness of the bank state:

value

$ay:AY, mi:MI$

$wf_Bank: \text{Bank} \rightarrow \mathbf{Bool}$

$wf_Bank(\rho, \alpha, \mu, \ell) \equiv \cup \mathbf{rng} \rho = \mathbf{dom} \alpha \wedge \cup \mathbf{rng} \mu = \mathbf{dom} \ell$

axiom

$ay < mi [\wedge \dots]$

We — perhaps too rigidly — assume that mortgage interest rates are higher than demand/deposit account interest rates: $ay < mi$. Operations on banks are denoted by the commands of the bank script language. First the syntax:

type

$\text{Cmd} = \text{OpA} | \text{CloA} | \text{Dep} | \text{Wdr} | \text{OpM} | \text{CloM} | \text{Pay}$

$\text{OpA} == \text{mkOA}(c:C)$

$\text{CloA} == \text{mkCA}(c:C, a:A)$

$\text{Dep} == \text{mkD}(c:C, a:A, p:P)$

```

Wdr == mkW(c:C,a:A,p:P)
OpM == mkOM(c:C,p:P)
Pay == mkPM(c:C,a:A,m:M,p:P,d:Date)
CloM == mkCM(c:C,m:M,p:P)
Reply = A | M | P | OkNok
OkNok == ok | notok

value
  period: Date × Date → Days [for calculating interest]
  before: Date × Date → Bool [first date is earlier than last date]

```

And then the semantics:

```

int_Cmd(mkPM(c,a,m,p,d))(ρ,α,μ,ℓ) ≡
  let (b,d') = ℓ(m) in
    if α(a) ≥ p
      then
        let i = interest(mi,b,period(d,d')),
              ℓ' = ℓ † [m ↦ ℓ(m) - (p - i)]
              α' = α † [a ↦ α(a) - p, a_i ↦ α(a_i) + i] in
          ((ρ,α',μ,ℓ'),ok) end
      else
          ((ρ,α',μ,ℓ),nok)
      end end
  pre c ∈ dom μ ∧ a ∈ dom α ∧ m ∈ μ(c)
  post before(d,d')

  interest: MI × Loan × Days → P

```

■

The idea about scripts is that they can somehow be objectively enforced: that they can be precisely understood and consistently carried out by all stakeholders, eventually leading to computerisation. But they are, at all times, part of the domain.

2.5.2 Requirements

Script requirements call for the possibly interactive computerisation of algorithms, that is, for rather classical computing problems. But sometimes these scripts can be expressed, computably, in the form of programs in a domain specific language. As an example we refer to [111]. [111] illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety. More specifically (a) product definitions based on standard actuarial models, including arbitrary continuous-time Markov and semi-Markov models, with cyclic transitions permitted; (b) calculation descriptions for reserves and other quantities of interest, based on differential equations; and (c) administration rules.

2.5.3 On Modeling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions program executions). Hence the full variety of techniques and notations for modeling programming (or specification) languages apply [13, 138, 223, 232, 250, 259]. [30, Chaps. 6–9] cover pragmatics, semantics and syntax techniques for defining functional, imperative and concurrent programming languages.

2.6 License Languages

License: a right or permission granted in accordance with law by a competent authority to engage in some business or occupation, to do some act, or to engage in some transaction which but for such license would be unlawful ■

Merriam Webster Online [184]

2.6.1 Conceptual Analysis

The Settings

A special form of scripts are increasingly appearing in some domains, notably the domain of electronic, or digital media. Here *licenses* express that a *licensor*, *o*, *permits* a *licensee*, *u*, to *render* (i.e., play) works of proprietary nature CD ROM-like music, DVD-like movies, etc. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. Classical digital rights license languages, [15, 5, 107, 108, 109, 154, 105, 137, 140, 171, 188, 186, 173, 166, 230, 214, 213, 2, 189], applied to the electronic “downloading”, payment and rendering (playing) of artistic works (for example music, literature readings and movies). In this chapter we generalise such applications languages and we extend the concept of licensing to also cover work authorisation (work commitment and promises) in health care, public government and schedule transport. The digital works for these new application domains are patient medical records, public government documents and bus/train/aircraft transport contracts. Digital rights licensing for artistic works seeks to safeguard against piracy and to ensure proper payments for the rights to render these works. Health care and public government license languages seek to ensure transparent and professional (accurate and timely) health care, respectively ‘good governance’. Transport contract languages seeks to ensure timely and reliable transport services by an evolving set of transport companies. Proper mathematical definition of licensing languages seeks to ensure smooth and correct computerised management of licenses and contracts.

On Licenses

The concepts of licenses and licensing express relations between (i) *actors* (licensors (the authority) and licensees), (ii) *entities* (artistic works, hospital patients, public administration, citizen documents) and bus transport contracts and (iii) *functions* (on entities), and as performed by actors. By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations: which functions on which entities the licensee is allowed (is licensed, is permitted) to perform. In this chapter we shall consider four kinds of entities: (i) digital recordings of artistic and intellectual nature: music, movies, readings (“audio books”), and the like, (ii) patients in a hospital as represented also by their patient medical records, (iii) documents related to public government, and (iv) transport vehicles, time tables and transport nets (of a buses, trains and aircraft).

Permissions and Obligations

The *permissions* and *obligations* issues are, (1) for the owner (agent) of some intellectual property to be paid (an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works; (2) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient; (3) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by

agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents; and (4) for bus passengers to enjoy reliable bus schedules — offered by bus transport companies on contract to, say public transport authorities and on sub-contract to other such bus transport companies where these transport companies are *obliged* to honour a contracted schedule.

2.6.2 The Pragmatics

By pragmatics we understand the study and practice of the factors that govern our choice of language in social interaction and the effects of our choice on others.

In this section we shall rough-sketch-describe pragmatic aspects of the four domains of (1) production, distribution and consumption of artistic works, (2) the hospitalisation of patient, i.e., hospital health care, (3) the handling of law-based document in public government and (4) the operational management of schedule transport vehicles. The emphasis is on the pragmatics of the terms, i.e., the language used in these four domains.

Digital Media

Example 12 Digital Media: *The intrinsic entities of the performing arts are the artistic works: drama or opera performances, music performances, readings of poems, short stories, novels, or jokes, movies, documentaries, newsreels, etc. We shall limit our span to the scope of electronic renditions of these artistic works: videos, CDs or other. In this chapter we shall not touch upon the technical issues of “downloading” (whether “streaming” or copying, or other). That and other issues should be analysed in [261].*

Operations on Digital Works:

For a consumer to be able to enjoy these works that consumer must (normally first) usually “buy a ticket” to their performances. The consumer, i.e., the theatre, opera, concert, etc., “goer” (usually) cannot copy the performance (e.g., “tape it”), let alone edit such copies of performances. In the context of electronic, i.e., digital renditions of these performances the above “cannots” take on a new meaning. The consumer may copy digital recordings, may edit these, and may further pass on such copies or editions to others. To do so, while protecting the rights of the producers (owners, performers), the consumer requests permission to have the digital works transferred (“downloaded”) from the owner/producer to the consumer, so that the consumer can render (“play”) these works on own rendering devices (CD, DVD, etc., players), possibly can copy all or parts of them, then possibly can edit all or parts of the copies, and, finally, possibly can further license these “edited” versions to other consumers subject to payments to “original” licensor.

License Agreement and Obligation:

To be able to obtain these permissions the user agrees with the wording of some license and pays for the rights to operate on the digital works.

Two Assumptions:

Two, related assumptions underlie the pragmatics of the electronics of the artistic works. The first assumption is that the format, the electronic representation of the artistic works is proprietary, that is, that the producer still owns that format. Either the format is publicly known or it is not, that is, it is somehow “secret”. In either case we “derive” the second assumption (from the fulfillment of the first). The second assumption is that the consumer is not allowed to, or cannot operate¹⁰ on the works by own means (software, machines). The second assumption implies that acceptance of a license results in the consumer receiving software that supports the consumer in performing all operations on licensed works, their copies and edited versions: rendering, copying, editing and sub-licensing.

¹⁰ render, copy and edit

Protection of the Artistic Electronic Works:

The issue now is: how to protect the intellectual property (i.e., artistic) and financial (exploitation) rights of the owners of the possibly rendered, copied and edited works, both when, and when not further distributed.

■

Health-care

Example 13 Health-care: *Citizens go to hospitals in order to be treated for some calamity (disease or other), and by doing so these citizens become patients. At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution. This request is directed at medical staff, that is, the patient authorises medical staff to perform a set of actions upon the patient. One could claim, as we shall, that the patient issues a license.*

Patients and Patient Medical Records:

So patients and their attendant patient medical records (PMRs) are the main entities, the “works” of this domain. We shall treat them synonymously: PMRs as surrogates for patients. Typical actions on patients — and hence on PMRs — involve admitting patients, interviewing patients, analysing patients, diagnosing patients, planning treatment for patients, actually treating patients, and, under normal circumstance, to finally release patients.

Medical Staff:

Medical staff may request (‘refer’ to) other medical staff to perform some of these actions. One can conceive of describing action sequences (and ‘referrals’) in the form of hospitalisation (not treatment) plans. We shall call such scripts for licenses.

Professional Health Care:

The issue is now, given that we record these licenses, their being issued and being honoured, whether the handling of patients at hospitals follow, or does not follow properly issued licenses.

■

Government Documents

Example 14 Documents: *By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)¹¹, understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public government. Typically national parliament and local (province and city) councils are part of law-making government. Law-enforcing government is called the executive (the administration). And law-interpreting government is called the judiciary [system] (including lawyers etc.).*

Documents:

A crucial means of expressing public administration is through documents.¹² We shall therefore provide a brief domain analysis of a concept of documents. (This document domain description also applies to patient medical records and, by some “light” interpretation, also to artistic works — insofar as they also are documents.) Documents are created, edited and read; and documents can be copied, distributed, the subject of calculations (interpretations) and be shared and shredded.

¹¹ *De l’esprit des lois (The Spirit of the Laws)*, published 1748

¹² Documents are, for the case of public government to be the “equivalent” of artistic works.

Document Attributes:

With documents one can associate, as attributes of documents, the actors who created, edited, read, copied, distributed (and to whom distributed), shared, performed calculations and shredded documents. With these operations on documents, and hence as attributes of documents one can, again conceptually, associate the location and time of these operations.

Actor Attributes and Licenses:

With actors (whether agents of public government or citizens) one can associate the authority (i.e., the rights) these actors have with respect to performing actions on documents. We now intend to express these authorisations as licenses.

Document Tracing:

An issue of public government is whether citizens and agents of public government act in accordance with the laws — with actions and laws reflected in documents such that the action documents enables a trace from the actions to the laws “governing” these actions. We shall therefore assume that every document can be traced back to its law-origin as well as to all the documents any one document-creation or -editing was based on. ■

Transportation**Example 15 Passenger and Goods Transport:****A Synopsis:**

Contracts obligate transport companies to deliver bus traffic according to a timetable. The timetable is part of the contract. A contractor may sub-contract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable. Contractors are either public transport authorities or contracted transport companies. Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit. The cancellation rights are spelled out in the contract. A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor. Etcetera.

A Pragmatics and Semantics Analysis:

The “works” of the bus transport contracts are two: the timetables and, implicitly, the designated (and obligated) bus traffic. A bus timetable appears to define one or more bus lines, with each bus line giving rise to one or more bus rides. Nothing is (otherwise) said about regularity of bus rides. It appears that bus ride cancellations must be reported back to the contractor. And we assume that cancellations by a sub-contractor is further reported back also to the sub-contractor’s contractor. Hence eventually that the public transport authority is notified. Nothing is said, in the contracts, such as we shall model them, about passenger fees for bus rides nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor. So we shall not bother, in this example, about transport costs nor transport subsidies. But will leave that necessary aspect as an exercise. The opposite of cancellations appears to be ‘insertion’ of extra bus rides, that is, bus rides not listed in the time table, but, perhaps, mandated by special events¹³ We assume that such insertions must also be reported back to the contractor. We assume concepts of acceptable and unacceptable bus ride delays. Details of delay acceptability may be given in contracts, but we ignore further descriptions of delay acceptability. but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors. We finally assume that sub-contractors cannot (otherwise) change timetables. (A timetable change can only occur after, or at, the expiration of a license.) Thus we find that contracts have definite period of validity. (Expired contracts may be replaced by new contracts, possibly with new timetables.)

¹³ Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

Contracted Operations, An Overview:

The actions that may be granted by a contractor according to a contract are: (i) *start*: to commence, i.e., to start, a bus ride (obligated); (ii) *end*: to conclude a bus ride (obligated); (iii) *cancel*: to cancel a bus ride (allowed, with restrictions); (iv) *insert*: to insert a bus ride; and (v) *subcontract*: to sub-contract part or all of a contract. ■

2.6.3 Schematic Rendition of License Language Constructs

There are basically two aspects to licensing languages: (i) the [actual] *licensing* [and sub-licensing], in the form of *licenses*, ℓ , by *licensors*, o , of *permissions* and thereby implied *obligations*, and (ii) the carrying-out of these obligations in the form of *licensee*, u , *actions*. We shall treat licensors and licensees on par, that is, some os are also us and vice versa. And we shall think of licenses as not necessarily material entities (e.g., paper documents), but allow licenses to be tacitly established (understood).

Licensing

The granting of a license ℓ by a licensor o , to a set of licensees $u_{u_1}, u_{u_2}, \dots, u_{u_u}$ in which ℓ expresses that these may perform actions $a_{a_1}, a_{a_2}, \dots, a_{a_a}$ on work items $e_{e_1}, e_{e_2}, \dots, e_{e_e}$ can be schematised:

$$\ell : \text{licensor } o \text{ contracts licensees } \{u_{u_1}, u_{u_2}, \dots, u_{u_u}\} \\ \text{to perform actions } \{a_{a_1}, a_{a_2}, \dots, a_{a_a}\} \text{ on work items } \{e_{e_1}, e_{e_2}, \dots, e_{e_e}\} \\ \text{allowing sub-licensing of actions } \{a_{a_i}, a_{a_j}, \dots, a_{a_k}\} \text{ to } \{u_{u_x}, u_{u_y}, \dots, u_{u_z}\}$$

The two sets of action designators, $das : \{a_{a_1}, a_{a_2}, \dots, a_{a_a}\}$ and $sas : \{a_{a_x}, a_{a_y}, \dots, a_{a_z}\}$ need not relate. **Sub-licensing:** Line 3 of the above schema, ℓ , expresses that licensees $u_{u_1}, u_{u_2}, \dots, u_{u_u}$, may act as licensors and (thereby sub-)license ℓ to licensees $us : \{u_{u_x}, u_{u_y}, \dots, u_{u_z}\}$, distinct from $sus : \{u_{u_1}, u_{u_2}, \dots, u_{u_u}\}$, that is, $us \cap sus = \{\}$. **Variants:** One can easily “cook up” any number of variations of the above license schema. **Revoke Licenses:** We do not show expressions for revoking part or all of a previously granted license.

Licensors and Licensees

Example 16 Licensors and Licensees:

Digital Media:

For digital media the original licensors are the original producers of music, film, etc. The “original” licensees are you and me ! Thereafter some of us may become licensors, etc.

Health-care:

For health-care the original licensors are, say in Denmark, the Danish governments’ National Board of Health¹⁴; and the “original” licensees are the national hospitals. These then sub-license their medical clinics (rheumatology, cancer, urology, gynecology, orthopedics, neurology, etc.) which again sub-licenses their medical staff (doctors, nurses, etc.). A medical doctor may, as is the case in Denmark for certain actions, not [necessarily] perform these but may sub-license their execution to nurses, etc.

¹⁴ In the UK: the NHS, etc.

Documents:

For government documents the original licensors are the (i) heads of parliament, regional and local governments, (ii) government (prime minister) and the heads of respective ministries, respectively the regional and local agencies and administrations. The “original” licensees are (i’) the members of parliament, regional and local councils charged with drafting laws, rules and regulations, (ii’) the ministry, respectively the regional and local agency department heads. These (the ‘s) then become licensors when licensing their staff to handle specific documents.

Transport:

For scheduled passenger (etc.) transportation the original licensors are the state, regional and/or local transport authorities. The “original” licensees are the public and private transport firms. These latter then become licensors licensing drivers to handle specific transport lines and/or vehicles. ■

Actors and Actions

Example 17 Actors and Actions:

Digital Media:

w refers to a digital “work” with w' designating a newly created one; s_i refers to a sector of some work. **render** $w(s_i, s_j, \dots, s_k)$: sectors s_i, s_j, \dots, s_k of work w are rendered (played, visualised) in that order. $w' := \text{copy } w(s_i, s_j, \dots, s_k)$: sectors s_i, s_j, \dots, s_k of work w are copied and becomes work w' . $w' := \text{edit } w$ **with** $\mathcal{E}(w_\alpha(s_a, s_b, \dots, s_c), \dots, w_\gamma(s_p, s_q, \dots, s_r))$: work w is edited while [also] incorporating references to or excerpts from [other] works $w_\alpha(s_a, s_b, \dots, s_c), \dots, w_\gamma(s_p, s_q, \dots, s_r)$. **read** w : work w is read, i.e., information about work w is somehow displayed. ℓ : **licensor m contracts licensees $\{u_{u_1}, u_{u_2}, \dots, u_{u_u}\}$ to perform actions $\{\text{RENDER, COPY, EDIT, READ}\}$ on work items $\{w_{i_1}, w_{i_2}, \dots, w_{i_w}\}$** . Etcetera: other forms of actions can be thought of.

Health-care:

Actors are here limited to the patients and the medical staff. We refer to Fig. 2.3. It shows an archetypal hospitalisation plan and identifies a number of actions; π designates patients, t designates treatment (medication, surgery, ...). Actions are performed by medical staff, say h , with h being an implicit argument of the actions. **interview** π : a PMR with name, age, family relations, addresses, etc., is established for patient π . **admit** π : the PMR records the anamnesis (medical history) for patient π . **establish analysis plan** π : the PMR records which analyses (blood tests, ECG, blood pressure, etc.) are to be carried out. **analyse** π : the PMR records the results of the analyses referred to previously. **diagnose** π : medical staff h diagnoses, based on the analyses most recently performed. **plan treatment for** π : medical staff h sets up a treatment plan for patient π based on the diagnosis most recently performed. **treat** π **wrt.** t : medical staff h performs treatment t on patient π , observes “reaction” and records this in the PMR. Predicate “actions”: **more analysis** π ?, **more treatment** π ? and **more diagnosis** π ?. **release** π : either the patient dies or is declared ready to be sent ‘home’. ℓ : **licensor o contracts medical staff $\{m_{m_1}, m_{m_2}, \dots, m_{m_m}\}$ to perform actions $\{\text{INTERVIEW, ADMIT, PLAN ANALYSIS, ANALYSE, DIAGNOSE, PLAN TREATMENT, TREAT, RELEASE}\}$ on patients $\{\pi_{p_1}, \pi_{p_2}, \dots, \pi_{p_p}\}$** . Etcetera: other forms of actions can be thought of.

Documents:

d refer to documents with d' designating new documents. $d' := \text{create based on } d_x, d_y, \dots, d_z$: A new document, named d' , is created, with no information “contents”, but referring to existing documents d_x, d_y, \dots, d_z . **edit** d **with** \mathcal{E} **based on** $d_{n_\alpha}, d_\beta, \dots, d_\gamma$: document d is edited with \mathcal{E} being the editing function and \mathcal{E}^{-1} being its “undo” inverse. **read** d : document d is being read. $d' := \text{copy } d$: document d is copied into a

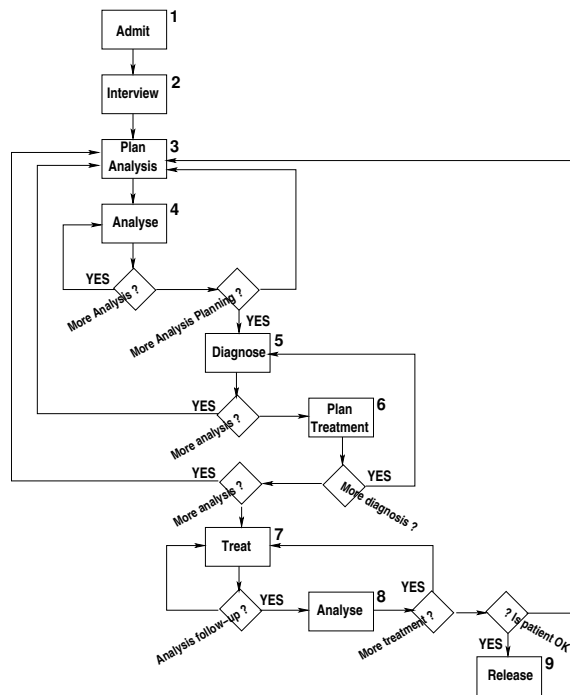


Fig. 2.3. An example single-illness non-fatal hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

new document named d' . **freeze d** : document d can, from now on, only be read. **shred d** : document d is shredded. That is, no more actions can be performed on d . ℓ : **licensor o contracts civil service staff** $\{c_{c_1}, c_{c_2}, \dots, c_{c_c}\}$ **to perform actions** $\{\text{CREATE, EDIT, READ, COPY, FREEZE, SHRED}\}$ **on documents** $\{d_{d_1}, d_{d_2}, \dots, d_{d_d}\}$. Etcetera: other forms of actions can be thought of.

Transport:

We restrict, without loss of generality, to bus transport. There is a timetable, tt . It records bus lines, l , and specific instances of bus rides, b . **start bus ride l, b at time t** : Bus line l is recorded in tt and its departure in tt is recorded as τ . Starting that bus ride at t means that the start is either on time, i.e., $t=\tau$, or the start is delayed $\delta_d : \tau-t$ or advanced $\delta_a : t-\tau$ where δ_d and δ_a are expected to be small intervals. All this is to be reported, in due time, to the contractor. **end bus ride l, b at time t** : Ending bus ride l, b at time t means that it is either ended on time, or earlier, or delayed. This is to be reported, in due time, to the contractor. **cancel bus ride l, b at time t** : t must be earlier than the scheduled departure of bus ride l, b . **insert an extra bus l, b' at time t** : t must be the same time as the scheduled departure of bus ride l, b with b' being a “marked” version of b . ℓ : **licensor o contracts transport staff** $\{b_{b_1}, b_{b_2}, \dots, b_{b_b}\}$ **to perform actions** $\{\text{START, END, CANCEL, INSERT}\}$ **on work items** $\{e_{e_1}, e_{e_2}, \dots, e_{e_e}\}$. Etcetera: other forms of actions can be thought of. ■

2.6.4 Requirements

Requirements for license language implementation basically amounts to requirements for three aspects. (i) The design of the license language, its abstract and concrete syntax, its interpreter, and its interfaces to distributed licensor and licensee behaviours; (ii) the requirements for a distributed system of licensor and licensee behaviours; and (iii) the monitoring and partial control of the states of licensor and licensee behaviours. The structuring of these distributed licensor and licensee behaviours differ from slightly to somewhat, but not that significant in the four license languages examples. Basically the licensor and licensee behaviours form a set of behaviours. Basically everyone can communicate with everyone. For the

case of digital media licensee behaviours communicate back to licensor behaviours whenever a properly licensed action is performed – resulting in the transfer of funds from licensees to licensors. For the case of health care some central authority is expected to validate the granting of licenses and appear to be bound by medical training. For the case of documents such checks appear to be bound by predetermined authorisation rules. For the case of transport one can perhaps speak of more rigid management & organisation dependencies as licenses are traditionally transferred between independent authorities and companies.

2.6.5 On Modeling License Languages

Licensors are expected to maintain a state which records all the licenses it has issued. Whenever a licensee “reports back” (the begin and/or the end) of the performance of a granted action, this is recorded in its state. Sometimes these granted actions are subject to fees. The licensor therefore calculates outstanding fees — etc. Licensees are expected to maintain a state which records all the licenses it has accepted. Whenever an action is to be performed the licensee records this and checks that it is permitted to perform this action. In many cases the licensee is expected to “report back”, both the beginning and the end of performance of that action, to the licensor. A typical technique of modeling licensors, licensees and patients, i.e., their PMRs, is to model them as (never ending) processes, a la CSP [148] with input/output, ch ?/ch ! m, communications between licensors, licensees and PMRs. Their states are modeled as programmable attributes.

2.7 Management & Organisation

- By **domain management** we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. 2.4) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who “backstops” complaints from lower management levels and from “floor” staff ■
- By **domain organisation** we shall understand (vi) the structuring of management and non-management staff “overseeable” into clusters with “tight” and “meaningful” relations; (vii) the allocation of strategic, tactical and operational concerns to within management and non-management staff clusters; and hence (viii) the “lines of command”: who does what, and who reports to whom, administratively and functionally ■

The ‘&’ is justified from the interrelations of items (i–viii).



Chapter 1 outlined the general principle, techniques and tools for analysing & describing discrete, composite endurants. Organisations and the management of these form such composite endurants. We shall therefore, really, not have much really new to add in this section !

2.7.1 Conceptual Analysis

We first bring some examples.

Example 18 Train Monitoring, I: *In China, as an example, till the early 1990s, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations (“up and down the lines”). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net.* ■

Example 19 Railway Management and Organisation: Train Monitoring, II: *We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.¹⁵ A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts. ■*

The above, albeit rough-sketch description, illustrated the following management and organisation issues: (i) There is a set of lowest-level (as here: train traffic scheduling and rescheduling) supervisors and their staff; (ii) they are organised into one such group (as here: per station); (iii) there is a middle-level (as here: regional train traffic scheduling and rescheduling) manager (possibly with some small staff), organised with one such per suitable (as here: railway) region; and (iv) the guidelines issued jointly by local and regional (...) supervisors and managers imply an organisational structuring of lines of information provision and command.

People staff enterprises, the components of infrastructures with which we are concerned, i.e., for which we develop software. The larger these enterprises — these infrastructure components — the more need there is for management and organisation. The role of management is roughly, for our purposes, twofold: first, to perform strategic, tactical and operational work, to set strategic, tactical and operational policies — and to see to it that they are followed. The role of management is, second, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution. Policy setting should help non-management staff operate normal situations — those for which no management interference is thus needed. And management “backstops” problems: management takes these problems off the shoulders of non-management staff. To help management and staff know who’s in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams — the usually hierarchical box and arrow/line diagrams.

The *management and organisation* model of a domain is a partial specification; hence all the usual abstraction and modeling principles, techniques and tools apply. More specifically, management is a set of predicate functions, or of observer and generator functions. These either parametrise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions. Organisation is thus a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modeled as sets (of recursively invoked sets) of equations.

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. 2.4), and then we show schematic processes which — for a rather simple scenario — model managers and the managed! Based on such a diagram, and modeling only one neighbouring group of a manager and the staff working for that manager we get a system in which one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modeled by ψ . Each staff, stf(i) operates in a context and a state modeled by $s\sigma(i)$.

type

¹⁵ That enactment may possibly imply the movement of several trains incident upon several stations: the one at which the manager is located, as well as possibly at neighbouring stations.

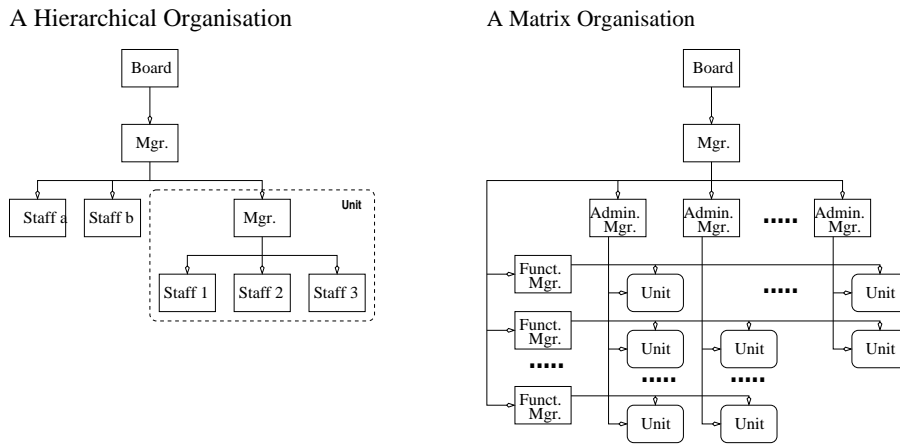


Fig. 2.4. Organisational structures

Msg, Ψ, Σ, Sx
 $S\Sigma = Sx \rightarrow \dot{m} \Sigma$
channel
 $\{ ms[i]:Msg \mid i:Sx \}$
value
 $s\sigma:S\Sigma, \psi:\Psi$

sys: Unit \rightarrow Unit
 $sys() \equiv \parallel \{ stf(i)(s\sigma(i)) \mid i:Sx \} \parallel mgr(\psi)$

In this system the manager, mgr, (1) either broadcasts messages, m, to all staff via message channel ms[i]. The manager's concoction, m_out(ψ), of the message, msg, has changed the manager state. Or (2) is willing to receive messages, msg, from whichever staff i the manager sends a message. Receipt of the message changes, m_in(i,m)(ψ), the manager state. In both cases the manager resumes work as from the new state. The manager chooses — in this model — which of the two things (1 or 2) to do by a so-called non-deterministic internal choice (\square).

$mg: \Psi \rightarrow \mathbf{in, out} \{ ms[i] \mid i:Sx \} \mathbf{Unit}$
 $mgr(\psi) \equiv$
(1) **let** (ψ', m) = m_out(ψ) **in** $\parallel \{ ms[i]!m \mid i:Sx \}; mgr(\psi') \mathbf{end}$
 \square
(2) **let** $\psi' = \square \{ \mathbf{let} m = ms[i]? \mathbf{in} m_in(i,m)(\psi) \mathbf{end} \mid i:Sx \}$ **in** $mgr(\psi') \mathbf{end}$

$m_out: \Psi \rightarrow \Psi \times MSG,$
 $m_in: Sx \times MSG \rightarrow \Psi \rightarrow \Psi$

And in this system, staff i, stf(i), (1) either is willing to receive a message, msg, from the manager, and then to change, st_in(msg)(σ), state accordingly, or (2) to concoct, st_out(σ), a message, msg (thus changing state) for the manager, and send it ms[i]!msg. In both cases the staff resumes work as from the new state. The staff member chooses — in this model — which of the two “things” (1 or 2) to do by a non-deterministic internal choice (\square).

$stf: i:Sx \rightarrow \Sigma \rightarrow \mathbf{in, out} ms[i] \mathbf{Unit}$
 $stf(i)(\sigma) \equiv$
(1) **let** $m = ms[i]? \mathbf{in} stf(i)(st_in(m)(\sigma)) \mathbf{end}$

$$(2) \quad \prod \text{let } (\sigma', m) = \text{st_out}(\sigma) \text{ in ms}[i]!m; \text{stf}(i)(\sigma') \text{ end}$$

$$\begin{aligned} \text{st_in}: \text{MSG} &\rightarrow \Sigma \rightarrow \Sigma, \\ \text{st_out}: \Sigma &\rightarrow \Sigma \times \text{MSG} \end{aligned}$$

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process non-deterministically, internal choice, “alternates” between “broadcast”-issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, internal choice, alternate between receiving orders from management and issuing individual messages to management. The conceptual example also illustrates modeling stakeholder behaviours as interacting (here CSP-like) processes.

Example 20 Strategic, Tactical and Operations Management: We think of (i) strategic, (ii) tactic, and (iii) operational managers as well as (iv) supervisors, (v) team leaders and the rest of the (vi) staff (i.e., workers) of a domain enterprise as functions. Each category of staff, i.e., each function, works in state and updates that state according to schedules and resource allocations — which are considered part of the state. To make the description simple we do not detail the state other than saying that each category works on an “instantaneous copy” of “the” state. Now think of six staff category activities, strategic managers, tactical managers, operational managers, supervisors, team leaders and workers as six simultaneous sets of actions. Each function defines a step of collective (i.e., group) (strategic, tactical, operational) management, supervisor, team leader and worker work. Each step is considered “atomic”. Now think of an enterprise as the “repeated” step-wise simultaneous performance of these category activities. Six “next” states arise. These are, in the reality of the domain, ameliorated, that is reconciled into one state. however with the next iteration, i.e., step, of work having each category apply its work to a reconciled version of the state resulting from that category’s previously yielded state and the mediated “global” state. Caveat: The below is not a mathematically proper definition. It suggests one !

type

0. $\Sigma, \Sigma_s, \Sigma_t, \Sigma_o, \Sigma_u, \Sigma_e, \Sigma_w$

value

1. str, tac, opr, sup, tea, wrk: $\Sigma_i \rightarrow \Sigma_i$
2. stra, tact, oper, supr, team, work: $\Sigma \rightarrow (\Sigma_{x_1} \times \Sigma_{x_2} \times \Sigma_{x_3} \times \Sigma_{x_4} \times \Sigma_{x_5}) \rightarrow \Sigma$
3. objective: $(\Sigma_s \times \Sigma_t \times \Sigma_o \times \Sigma_u \times \Sigma_e \times \Sigma_w) \rightarrow \mathbf{Bool}$
3. enterprise, ameliorate: $(\Sigma_s \times \Sigma_t \times \Sigma_o \times \Sigma_u \times \Sigma_e \times \Sigma_w) \rightarrow \Sigma$
4. enterprise: $(\sigma_s, \sigma_t, \sigma_u, \sigma_e, \sigma_w) \equiv$
6. **let** $\sigma'_s = \text{stra}(\text{str}(\sigma_s))(\sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w),$
7. $\sigma'_t = \text{tact}(\text{tac}(\sigma_t))(\sigma'_s, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w),$
8. $\sigma'_o = \text{oper}(\text{opr}(\sigma_o))(\sigma'_s, \sigma'_t, \sigma'_u, \sigma'_e, \sigma'_w),$
9. $\sigma'_u = \text{supr}(\text{sup}(\sigma_u))(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_e, \sigma'_w),$
10. $\sigma'_e = \text{team}(\text{tea}(\sigma_e))(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_w),$
11. $\sigma'_w = \text{work}(\text{wrk}(\sigma_w))(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e) \text{ in}$
12. **if** $\text{objective}(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w)$
13. **then** $\text{ameliorate}(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w)$
14. **else** $\text{enterprise}(\sigma'_s, \sigma'_t, \sigma'_o, \sigma'_u, \sigma'_e, \sigma'_w)$
15. **end end**

0. Σ is a further undefined and unexplained enterprise state space. The various enterprise players view this state in their own way.

1. Six staff group operations, str, tac, opr, sup, tea and wrk, each act in the enterprise state such as conceived by respective groups to effect a resulting enterprise state such as achieved by respective groups.

2. Six staff group state amelioration functions, $\text{ame}_s, \text{ame}_t, \text{ame}_o, \text{ame}_u, \text{ame}_e$ and ame_w , each apply to the resulting enterprise states such as achieved by respective groups to yield a result state such as achieved by that group.
3. An overall objective function tests whether a state summary reflects that the objectives of the enterprise has been achieved or not.
4. The enterprise function applies to the tuple of six group-biased (i.e., ameliorated) states. Initially these may all be the same state. The result is an ameliorated state.
5. An iteration, that is, a step of enterprise activities, lines 5.–13. proceeds as follows:
6. strategic management operates
 - in its state space, $\sigma_s : \Sigma$;
 - effects a next (un-ameliorated strategic management) state σ'_s ;
 - and ameliorates this latter state in the context of all the other player's ameliorated result states.
- 7.–11. The same actions take place, simultaneously for the other players: tac , opr , sup , tea and wrk .
12. A test, *has objectives been met*, is made on the six ameliorated states.
13. If test is successful, then the enterprise terminates in an ameliorated state.
14. Otherwise the enterprise recurses, that is, “repeats” itself in new states.

The above “function” definition is suggestive. It suggests that a solution to the fix-point 6-tuple of equations over “intermediate” states, σ'_x , where x is any of s, t, o, u, e, w , is achievable by iteration over just these 6 equations. ■

2.7.2 Requirements

Top-level, including strategic management tends to not be amenable to “automation”. Increasingly tactical management tends to “divide” time between “bush-fire, stop-gap” actions – hardly automatable and formulating, initiating and monitoring main operations. The initiation and monitoring of tactical actions appear amenable to partial automation. Operational management – with its reliance on rules & regulations, scripts and licenses – is where computer monitoring and partial control has reaped the richest harvests.

2.7.3 On Modeling Management and Organisation

Management and organisation basically spans entity, function, event and behaviour intensities and thus typically require the full spectrum of modeling techniques and notations — summarised in Sect. 2.2.3.

2.8 Human Behaviour

- By **domain human behaviour** we shall understand any of a quality spectrum of carrying out assigned work: from (i) careful, diligent and accurate, via (ii) sloppy dispatch, and (iii) delinquent work, to (iv) outright criminal pursuit ■

Although we otherwise do not go into any depth with respect to the analysis & description of humans, we shall momentarily depart from this “abstinence”.

2.8.1 Conceptual Analysis

To model human behaviour “smacks” like modeling human actors, the psychology of humans, etc. ! We shall not attempt to model the psychological side of humans — for the simple reason that we neither know how to do that nor whether it can at all be done. Instead we shall be focusing on the effects on non-human manifest entities of human behaviour.

Example 21 Banking — or Programming — Staff Behaviour: Let us assume a bank clerk, “in ye olde” days, when calculating, say mortgage repayments (cf. Example 10). We would characterise such a clerk as being diligent, etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations “tally up”, or lets others do so. We would characterise a clerk as being sloppy if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being delinquent if that person systematically forgets these checks. And we would call such a person a criminal if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater. Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 11). We would characterise the programmer as being diligent if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being sloppy if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being delinquent if that person systematically forgets these checks and tests. And we would characterise the programmer as being a criminal if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. ■

Example 22 A Human Behaviour Mortgage Calculation: Example 11 gave a semantics to the mortgage calculation request (i.e., command) as would a diligent bank clerk be expected to perform it. To express, that is, to model, how sloppy, delinquent, or outright criminal persons (staff?) could behave we must modify the $\text{int_Cmd}(\text{mkPM}(c,a,m,p,d'))(\rho,\alpha,\mu,\ell)$ definition.

```

int_Cmd(mkPM(c,a,m,p,d))(\rho,\alpha,\mu,\ell) ≡
  let (b,d') = \ell(m) in
  if q(\alpha(a),p) [\alpha(a) ≤ p ∨ \alpha(a) = p ∨ \alpha(a) ≤ p ∨ ...]
  then
    let i = f1(interest(mi,b,period(d,d'))),
        \ell' = \ell † [m → f2(\ell(m) - (p - i))],
        \alpha' = \alpha † [a → f3(\alpha(a) - p), ai → f4(\alpha(ai) + i), a “staff” → f “staff” (\alpha(a “staff”) + i)] in
    ((\rho,\alpha',\mu,\ell'),ok) end
  else
    ((\rho,\alpha',\mu,\ell),nok)
  end end
pre c ∈ dom \mu ∧ m ∈ \mu(c)

```

```

q: P × P → Bool
f1, f2, f3, f4, f “staff”: P → P [typically: f “staff” = \lambda p.p]

```

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ of Example 22 are deliberately left undefined. They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine. The point of Example 22 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) to perform (incl., correctly program) it before one can “pinpoint” all the places where lack of diligence may “set in”. The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ designate those places. The point of Example 22 is also that we must first domain-define, “to the best of our ability” all the places where human behaviour may play other than a desirable role. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

Commensurate with the above, humans interpret rules and regulations differently, and, for some humans, not always consistently — in the sense of repeatedly applying the same interpretations. Our final specification pattern is therefore:

type

$$\text{Action} = \Theta \xrightarrow{\sim} \Theta\text{-infset}$$
value

$$\text{hum_int}: \text{Rule} \rightarrow \Theta \rightarrow \text{RUL-infset}$$

$$\text{action}: \text{Stimulus} \rightarrow \Theta \rightarrow \Theta$$

$$\text{hum_beha}: \text{Stimulus} \times \text{Rules} \rightarrow \text{Action} \rightarrow \Theta \xrightarrow{\sim} \Theta\text{-infset}$$

$$\text{hum_beha}(\text{sy_sti}, \text{sy_rul})(\alpha)(\theta) \text{ as } \theta\text{set}$$
post

$$\theta\text{set} = \alpha(\theta) \wedge \text{action}(\text{sy_sti})(\theta) \in \theta\text{set}$$

$$\wedge \forall \theta': \Theta \cdot \theta' \in \theta\text{set} \Rightarrow$$

$$\exists \text{se_rul}: \text{RUL} \cdot \text{se_rul} \in \text{hum_int}(\text{sy_rul})(\theta) \Rightarrow \text{se_rul}(\theta, \theta')$$

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. “Suits” means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not. The above-stated axioms express how it is in the domain, not how we would like it to be. For that we have to establish requirements.

2.8.2 Requirements

Requirements in relation to the human behaviour facet is not requirements about software that “replaces” human behaviour. Such requirements were hinted at in Sects. 2.5.2–2.7.2. Human behaviour facet requirements are about software that checks human behaviour; that its remains diligent; that it does not transgress into sloppy, delinquent, let alone criminal behaviour. When transgressions are discovered, appropriate remedial actions may be prescribed.

2.8.3 On Modeling Human Behaviour

To model human behaviour is, “initially”, much like modeling management and organisation. But only ‘initially’. The most significant human behaviour modeling aspect is then that of modeling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like CafeOBJ [129] and RSL [131]) is to be preferred. To prescribe requirements is to prescribe the monitoring of the human input at the computer interface.

2.9 Conclusion

We have introduced the scientific and engineering concept of domain theories and domain engineering; and we have brought but a mere sample of the principles, techniques and tools that can be used in creating domain descriptions.

2.9.1 Completion

Domain acquisition results in typically up to thousands of units of domain descriptions. Domain analysis subsequently also serves to classify which facet any one of these description units primarily characterises. But some such “compartmentalisations” may be difficult, and may be deferred till the step of “completion”. It may then be, “at the end of the day”, that is, after all of the above facets have been modeled that some description units are left as not having been described, not deliberately, but “circumstantially”. It then behooves the domain engineer to fit these “dangling” description units into suitable parts of the domain

description. This “slotting in” may be simple, and all is fine. Or it may be difficult. Such difficulty may be a sign that the chosen model, the chosen description, in its selection of entities, functions, events and behaviours to model — in choosing these over other possible selections of phenomena and concepts is not appropriate. Another attempt must be made. Another selection, another abstraction of entities, functions, etc., may need be chosen. Usually however, after having chosen the abstractions of the intrinsic phenomena and concepts, one can start checking whether “dangling” description units can be fitted in “with ease”.

2.9.2 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and it appears not to be desirable to obtain a single, “universal” specification language capable of “equally” elegantly, suitably abstractly modeling all aspects of a domain. Hence one must conclude that the full modeling of domains shall deploy several formal notations – including plain, good old mathematics in all its forms. The issues are then the following which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing series of “Integrating Formal Methods” conferences [4] is a good source for techniques, compositions and meanings.

2.9.3 The Impossibility of Describing Any Domain Completely

Domain descriptions are, by necessity, abstractions. One can never hope for any notion of complete domain descriptions. The situation is no better for domains such as we define them than for physics. Physicists strive to understand the manifest world around us – the world that was there before humans started creating “their domains”. The physicists describe the physical world “in bits and pieces” such that large collections of these pieces “fit together”, that is, are based on some commonly accepted laws and in some commonly agreed mathematics. Similarly for such domains as will be the subject of domain science & engineering such as we cover that subject in [81, 66] and in the present chapter and reports [76, 68]. Individual such domain descriptions will be emphasizing some clusters of facets, others will be emphasizing other aspects.

2.9.4 Rôles for Domain Descriptions

We can distinguish between a spectrum of rôles for domain descriptions. Some of the issues brought forward below may have been touched upon in [81, 66].

Alternative Domain Descriptions:

It may very well be meaningful to avail oneself of a variety of domain models (i.e., descriptions) for any one domain, that is, for what we may consider basically one and the same domain. In control theory (a science) and automation (an engineering) we develop specific descriptions, usually on the form of a set of differential equations, for any one control problem. The basis for the control problem is typically the science of mechanics. This science has many renditions (i.e., interpretations). For the control problem, say that of keeping a missile carried by a train wagon, erect during train movement and/or windy conditions, one may then develop a “self-contained” description of the problem based on some mechanics theory presentation. Similarly for domains. One may refer to an existing domain description. But one may re-develop a textually “smaller” domain description for any one given, i.e., specific problem.

Domain Science:

A domain description designates a domain theory. That is, a bundle of propositions, lemmas and theorems that are either rather explicit or can be proven from the description. So a domain description is the basis for a theory as well as for the discovery of domain laws, that is, for a domain science. We have sciences of physics (incl. chemistry), biology, etc. Perhaps it is about time to have proper sciences, to the extent one can have such sciences for human-made domains.

Business Process Re-engineering:

Some domains manifest serious amounts of human actions and interactions. These may be found to not be efficient to a degree that one might so desire. A given domain description may therefore be a basis for suggesting other *management & organisation* structures, and/or *rules & regulations* than present ones. Yes, even making explicit *scripts* or a *license language* which have hitherto been tacitly understood – without necessarily computerising any support for such a *script* or *license language*. The given and the resulting domain descriptions may then be the basis for *operations research* models that may show desired or acceptable efficiency improvements.

Software Development:

[66] shows one approach to requirements prescription. Domain analysis & description, i.e., domain engineering, is here seen as an initial phase, with requirements prescription engineering being a second phase, and software design being a third phase. We see domain engineering as indispensable, that is, an absolute must, for software development. [53, *Domains: Their Simulation, Monitoring and Control*] further illustrates how domain engineering is a base for the development of domain simulators, demos, monitors and controllers.

2.9.5 Grand Challenges of Informatics¹⁷

To establish a reasonably trustworthy and believable theory of a domain, say the transportation, or just the railway domain, may take years, possibly 10–15! Similarly for domains such as the financial service industry, the market (of consumers and producers, retailers, wholesaler, distribution cum supply chain), health care, and so forth. The current author urges younger scientists to get going! It is about time.

2.10 Bibliographical Notes

To create domain descriptions, or requirements prescriptions, or software designs, properly, at least such as this author sees it, is a joy to behold. The beauty of carefully selected and balanced abstractions, their interplay with other such, the relations between phases, stages and steps, and many more conceptual constructions make software engineering possibly the most challenging intellectual pursuit today. For this and more consult [27, 30, 32].

¹⁷ In the early-to-mid 2000s there were a rush of research foundations and scientists enumerating “*Grand Challenges of Informatics*”

Towards Formal Models of Processes and Prompts – a Sketch

We¹ sketch an approach to a formal semantics of the domain analysis & description process of Chapter 1.

3.1 Introduction

Chapter 1 introduced a method for analysing and describing manifest domains. In this chapter we formalise the calculus of this method. The formalisation has two aspects: the formalisation of the process of sequencing the prompts of the calculus, and the formalisation of the individual prompts.

The presentation of a calculus for analysing and describing manifest domains, introduced in Chapter 1 was and is necessarily informal. The human process of “extracting” a description of a domain, based on analysis, “wavers” between the domain, as it is revealed to our senses, and therefore necessarily informal, and its recorded description, which we present in two forms, an informal narrative and a formalisation. In the present chapter we shall provide a formal, operational semantics formalisation of the analysis and description calculus. There are two aspects to the semantics of the analysis and description calculus. There is the formal explanation of the process of applying the analysis and description prompts, in particular the practical meaning² of the results of applying the analysis prompts, and there is the formal explanation of the meaning of the results of applying the description prompts. The former (i.e., the practical meaning of the results of applying the analysis prompts) amounts to a model of the process whereby the domain analyser cum describer navigates “across” the domain, alternating between applying sequences of one or more analysis prompts and applying description prompts. The latter (formal explanation of the meaning of the results of applying the description prompts) amounts to a model of the domain (as it evolves in the mind of the analyser cum describer³), the meaning of the evolving description, and thereby the relation between the two.

¹ Chapter 3 is primarily based on [60]. which evolved into [64]. The present chapter presents an analysis & description process and prompt semantics model based on the domain ontology and the analysis & description process as presented in [60]. It may not be exactly the domain ontology of this thesis; but the differences are not substantial enough, we think, to warrant a rewrite of the formulas of the present chapter.

² in contrast to a formal mathematical meaning

³ By ‘domain analyser cum describer’ we mean a group of one or more professionals, well-educated and trained in the domain analysis & description techniques outlined in, for example, [70], and where these professionals work closely together. By ‘working closely together’ we mean that they, together, day-by-day work on each their sections of a common domain description document which they “buddy check”, say every morning, then discuss, as a group, also every day, and then revise and further extend, likewise every day. By “buddy checking” we mean that group member *A* reviews group member *B*’s most recent sections – and where this reviewing alternates regularly: *A* may first review *B*’s work, then *C*’s, etcetera.

We shall, occasionally refer to the ‘domain analyser cum describer’ as the ‘domain engineer’.

3.1.1 Related Work

To this author’s knowledge there are not many papers, other than the author’s own, [80, 70, 76, 66, 65] and the present chapter, which proposes a calculus of analysis and description prompts for capturing a domain, let alone, as this chapter tries, to formalise aspects of this calculus.

There is, however a “school of software engineering”, “anchored” in the 1987 publication: [195, Leon Osterweil]. As the title of that paper reveals: “*Software Processes Are Software Too*” the emphasis is on considering the software development process as prescribable by a software program. That is not what we are aiming at. We are aiming at an abstract and formal description of a large class of domain analysis & description processes *in terms of possible development calculi*. And in such a way that one can reason about such processes. The Osterweil paper suggests that any particular software development can be described by a program, and, if we wish to reason about the software development process we must reason over that program, but there is no requirement that the “software process programs” be expressed in a language with a proof system.⁴ In contrast we can reason over the properties of the development calculi as well as over the resulting description.

There is another “school of programming”, one that more closely adheres to the use of a calculus [11, 187]. The calculus here is a set of refinement rules, a *Refinement Calculus*⁵, that “drives” the developer from a specification to an executable program. Again, that is not what we are doing here. The proposed calculi of analysis and of description prompts [70] “drives” the domain engineer in developing a domain description. That description may then be ‘refined’ using a refinement calculus.

3.1.2 Structure of Chapter

Section 3.2 provides a terse summary of the analysis & description of endurants. It is without examples. For such we refer to [70, Sects. 2.–3., Pages 7–29.]. Section 3.3 is informal. It discusses issues of syntax and semantics. The reason we bring this short section is that the current chapter turns “things upside/down”: from semantics we extract syntax ! From the real entities of actual domains we extract domain descriptions. Section 3.4 presents a pseudo-formal operational semantics explication of the process of proceeding through iterated sequences of analysis prompts to description prompts. The formal meaning of these prompts are given in Sect. 3.8. But first we must “prepare the ground”: The meaning of the analysis and description prompts is given in terms of some formal “context” in which the domain engineer works. Section 3.5 discusses this notion of “image” — an informal aspect of the ‘context’. It is a brief discussion. Section 3.6 presents the formal aspect of the ‘context’: perceived abstract syntaxes of the ontology of domain endurants and of endurant values. Section 3.7 Discusses, in a sense, the mental processes – *from syntax to semantics and back again !* – that the domain engineer appears to undergo while analysing (the semantic) domain entities and synthesizing (the syntactic) domain descriptions. Section 3.8 presents the analysis and description prompts meanings. It represents a high point of this chapter. It so-to-speak justifies the whole “exercise” ! Section 3.9 concludes the chapter. We summarize what we have “achieved”. And we discuss whether this “achievement” is a valid one !

3.2 Domain Analysis and Description

We refer to Chapter 1

Both [60] and [64] brought at this point extensive sections on the *analysis & description method* of this thesis, i.e., Chapter 1. Here we just refer to that chapter.

⁴ The **RAISE** Specification Language [132] does have a proof system.

⁵ Ralph-Johan Back appears to be the first to have proposed the idea of refinement calculi, cf. his 1978 PhD thesis *On the Correctness of Refinement Steps in Program Development*, [http://users.abo.fi/backrj/index.php?page=Refinement calculus.all.html&menu=3](http://users.abo.fi/backrj/index.php?page=Refinement%20calculus.all.html&menu=3).

3.3 Syntax and Semantics

3.3.1 Form and Content

Sections 1.4, 1.5 and 1.8 [Chapter 1] appears to be expressed in the syntax of the **Raise** [132] Specification Language, **RSL** [131]. But it only “appears” so. When, in the “conventional” use of **RSL**, we apply meaning functions, we apply them to syntactic quantities. In Sect. 3.2 the “meaning” functions are the analysis, a.-j., and description, 1.–8., prompts:

- | | |
|--------------------------|-------------------------------|
| a. is_entity, 10 | l. is_living_species, 15 |
| b. is_endurant, 11 | m. is_plant, 15 |
| c. is_perdurant, 11 | n. is_animal, 16 |
| d. is_discrete, 11 | o. is_human, 16 |
| e. is_continuous, 11 | p. has_components, 17 |
| f. is_physical_part, 12 | q. has_materials, 17 |
| g. is_living_species, 12 | r. is_artefact, 17 |
| h. is_structure, 13 | s. observe_endurant_sorts, 18 |
| i. is_part, 14 | t. has_concrete_type, 21 |
| j. is_atomic, 15 | u. has_mereology, 26 |
| k. is_composite, 15 | v. attribute_types, 29 |

and

- | | |
|---------------------------------|-----------------------------------|
| [1] observe_endurant_sorts, 19 | [5] observe_unique_identifier, 25 |
| [2] observe_part_type, 21 | [6] observe_mereology, 27 |
| [3] observe_component_sorts, 22 | [7] observe_attributes, 29 |
| [4] observe_material_sorts, 24 | |

The quantities that these prompts are “applied to” are semantic ones, in effect, they are the “ultimate” semantic quantities that we deal with: *the real*, i.e., *actual domain* entities! The quantities that these prompts “yield” are syntactic ones! That is, we have “turned matters inside/out”. From semantics we “extract” syntax. The arguments of the above-listed 22 prompts are domain entities, i.e., in principle, in-formalisable things. Their types, typically listed as P , denote possibly infinite classes, \mathcal{P} , of domain entities. When we write P we thus mean \mathcal{P} .

3.3.2 Syntactic and Semantic Types

When we, classically, define a programming language, we first present its syntax, then its semantics. The latter is presented as two – or three – possibly interwoven texts: the static semantics, i.e., the well-formedness of programs, the dynamic semantics, i.e., the mathematical meaning of programs — with a corresponding proof system being the “third text”. We shall briefly comment on the ideas of static and dynamic semantics. In designing a programming language, and therefore also in narrating and formalising it, one is well advised in deciding first on the semantic types, then on the syntactic ones. With describing [f.ex., manifest] domains, matters are the other way around: The semantic domains are given in the form of the endurants and perdurants; and the syntactic domains are given in the form that we, the humans of the domain, mention in our speech acts [234, 7]. That is, from a study of actual life domains, we extract the essentials that speech acts deal with when these speech acts are concerned with performing or talking about entities in some actual world.

3.3.3 Names and Denotations

Above, we may have been somewhat cavalier with the use of names for sorts and names for their meaning. Being so, i.e., “cavalier”, is, unfortunately a “standard” practice. And we shall, regrettably, continue to be cavalier, i.e., “loose” in our use of names of syntactic “things” and names for the denotation of these syntactic “things”. The context of these uses usually makes it clear which use we refer to: a syntactic use or a semantic one. As from Sect. 3.6 we shall be more careful in distinguishing clearly between the names of sorts and the values of sorts, i.e., between syntax and semantics.

3.4 A Model of the Domain Analysis & Description Process

3.4.1 Introduction

A Summary of Prompts

In Sect. 3.3.1 we listed the two classes of prompts: the *domain [endurant] analysis prompts*: and the *domain [endurant] description prompts*: These prompts are “imposed” upon the domain by the domain analyser cum describer. They are “figuratively” applied to the domain. Their orderly, sequenced application follows the method hinted at in the previous section, detailed in Chapter 1. This process of application of prompts will be expressed in a pseudo-formal notation in this section. The notation looks formal but since we have not formalised these prompts it is only pseudo-formal. We formalise these prompts in Sect. 3.8.

Preliminaries

Let P be a sort, that is, a collection of endurants. By P we shall understand both a syntactic quantity: the name of P , and a semantic quantity, the type (of all endurant values of type) P . By $\iota p:P^6$ we shall understand a semantic quantity: an (arbitrarily selected) endurant in P . To guide our analysis & description process we decompose it into steps. Each step “handles” a part sort $p:P$ or a material sort $m:M$ or a component sort $k:K$. Steps handling discovery of composite part sorts generates a set of part sort names $P_1, P_2, \dots, P_n:PNm$. Steps handling discovery of atomic part sorts may generate a material sort name, $m:MNm$, or component sort name, $k:KNm$. The part, material and component sort names are put in a reservoir for *sorts to be inspected*. Once handled, the sort name is removed from that reservoir. Handling of material sorts besides discovering their attributes may involve the discovery of further part sorts — which we assume to be atomic. Each domain description prompt results in domain specification text (here we show only the formal texts, not the narrative texts) being deposited in the domain description reservoir, a global variable τ . We do not formalise this text. Clauses of the form `observe_XXX(p)`, where `XXX` ranges over `part_sorts`, `concrete_type`, `unique_identifier`, `mereology`, `part_attributes`, `part_component_sorts`, `part_material_sorts`, and `material_part_sorts`, stand for “text” generating functions. They are defined in Sect. 3.8.3.

Initialising the Domain Analysis & Description Process

We remind the reader that we are dealing only with endurant domain entities. The domain analysis approach covered in Sect. 3.2 was based on decomposing an understanding of a domain from the “overall domain” into its separate entities, and these, if not atomic, into their sub-entities. So we need to initialise the domain analysis & description process by selecting (or choosing) the domain Δ . Here is how we think of that “initialisation” process. The domain analyser & describer spends some time focusing on the domain,

⁶ ι is Whitehead and Russell’s description operator [257, Principia Mathematica]: an inverted ι ; plato.stanford.edu/entries/pm-notation

maybe at the “white board”⁷, rambling, perhaps in an un-structured manner, across its domain, Δ , and its sub-domains. Informally jotting down more-or-less final sort names, building, in the domain analyser & describer’s mind an image of that domain. After some time doing this the domain analyser & describer is ready. An image of the domain includes the or a domain endurant, $\delta:\Delta$. Let Δnm be the name of the sort Δ . That name may be either a part sort name, or a material sort name, or a component sort name.

3.4.2 A Model of the Analysis & Description Process

A Process State

- 150 Let Nm denote either a part or a material or a component sort name.
 151 A global variable αps will accumulate all the sort names being discovered.
 152 A global variable vps will hold names of sorts that have been “discovered”, but have yet to be analysed & described.

type

150. $Nm = PNm \mid MNm \mid KNm$

variable

151. $\alpha ps := [\Delta nm]$ **type** $Nm\text{-set}$

152. $vps := [\Delta nm]$ **type** $Nm\text{-set}$

We shall explain the use of [...]s and operations on the above variables in Sect. 3.4.3 on Page 114. Each iteration of the “root” function, $analyse_and_describe_endurant_sort(Nm, 1:nm)$, as we shall call it, involves the selection of a sort (value) (which is that of either a part sort or a material sort) with this sort (value) then being removed.

- 153 The selection occurs from the global state component vps (hence: ()) and changes that state (hence **Unit**).

value

153. $sel_and_rem_Nm: \mathbf{Unit} \rightarrow Nm$

153. $sel_and_rem_Nm() \equiv \mathbf{let} \ nm:Nm \bullet \ nm \in vps \ \mathbf{in} \ vps := vps \setminus \{nm\} ; \ nm \ \mathbf{end}; \ \mathbf{pre}: vps \neq \{\}$

A Technicality

- 154 The main analysis & description functions of the next sections, except the “root” function, are all expressed in terms of a pair, $(nm, val):NmVAL$, of a sort name and an endurant value of that sort.

type

154. $NmVAL = (PNm \times PVAL) \mid (MNm \times MVAL) \mid (KNm \times KVAL)$

Analysis & Description of Endurants

- 155 To analyse and describe endurants means to first
 a examine those endurants which have yet to be so analysed and described
 b by selecting (and removing from vps) a yet un-examined sort nm ;

⁷ Here ‘white board’ is a conceptual notion. It could be physical, it could be yellow “post-it” stickers, or it could be an electronic conference “gadget”.

- c then analyse and describe an enduring entity ($1:nm$) of that sort — this analysis, when applied to composite parts, leads to the insertion of zero⁸ or more sort names⁹.

As is indicated in Sect. 1.5.2 [Chapter 1], the mereology of a part, if it has one, may involve unique identifiers of any part sort, hence must be done after all such part sort unique identifiers have been identified. Similarly for attributes which also may involve unique identifiers,

- 156 then, if it has a mereology,
 a to analyse and describe the mereology of each part sort,
 157 and finally to analyse and describe the attributes of each sort.

value

155. analyse_and_describe_endurants: **Unit** → **Unit**
 155. analyse_and_describe_endurants() ≡
 155a. **while** \sim is_empty(vps) **do**
 155b. **let** nm = sel_and_rem_Nm() **in**
 155c. analyse_and_describe_endurant_sort(nm, $1:nm$) **end end** ;
 156. **for all** nm:PNm • nm ∈ α ps **do if** has_mereology(nm, $1:nm$)¹⁰
 156a. **then** observe_mereology(nm, $1:nm$)¹¹ **end end**
 157. **for all** nm:Nm • nm ∈ α ps **do** observe_attributes(nm, $1:nm$)¹² **end**

The $1:nm$ of Items 155c, 156, 156a and 157 are crucial. The domain analyser is focused on (part or material or component) sort nm and is “directed” (by those items) to choose (select) an enduring (a part or a material or component) $1:nm$ of that sort.

- 158 To analyse and describe an enduring
 a is to find out whether it is a part. If so then it is to analyse and describe it.
 b If it instead is a material, then to analyse and describe it as a material.
 c If it instead is a component, then to analyse and describe it as a component.

value

158. analyse_and_describe_endurant_sort: NmVAL → **Unit**
 158. analyse_and_describe_endurant_sort(nm, val) ≡
 158a. **is_part**(nm, val)¹³ →¹⁴ analyse_and_describe_part_sorts(nm, val),
 158b. **is_material**(nm, val)¹⁵ → **observe_material_part_sort**(nm, val)¹⁶,
 158c. **is_component**(nm, val)¹⁷ → **observe_component_sort**(nm, val)¹⁸

- 159 To analyse and describe the internal qualities of a part
 a first describe its unique identifier.
 b If the part is atomic it is analysed and described as such;

⁸ If the sub-parts of $1nm$ are all either atomic and have no materials or components or have already been analysed, then no new sort names are added to the repository vps).

⁹ These new sort names are then “picked-up” for sort analysis &c. in a next iteration of the while loop.

¹² We formalise has_mereology in Sect. 3.8.2 on Page 126.

¹² We formalise observe_mereology in Sect. 3.8.3 on Page 128.

¹² We formalise observe_attributes in Sect. 3.8.3 on Page 128.

¹⁸ We formalise is_part in Sect. 3.8.2 on Page 125.

¹⁸ The conditional clause: $cond_1 \rightarrow clau_1, cond_2 \rightarrow clau_2, \dots, cond_n \rightarrow clau_n$ is same as **if** $cond_1$ **then** $clau_1$ **else if** $cond_2$ **then** $clau_2$ **else ... if** $cond_n$ **then** $clau_n$ **end end ... end** .

¹⁸ We formalise is_material in Sect. 3.8.2 on Page 125.

¹⁸ We formalise observe_material_part_sort in Sect. 3.8.3 on Page 129.

¹⁸ We formalise is_component in Sect. 3.8.2 on Page 125.

¹⁸ We formalise observe_component_sort in Sect. 3.8.3 on Page 129.

- c If composite it is analysed and described as such.
- d Part p must be discrete.

value

159. analyse_and_describe_part_sorts: NmVAL \rightarrow **Unit**
 159. analyse_and_describe_part_sorts(nm,val) \equiv
 159a. **observe_unique_identifier**(nm,val)¹⁹;
 159b. **is_atomic**(nm,val)²⁰ \rightarrow analyse_and_describe_atomic_part(nm,val),
 159c. **is_composite**(nm,val)²¹ \rightarrow analyse_and_describe_composite_parts(nm,val)
 159d. **pre: is_discrete**(nm,val)²²

- 160 To analyse and describe an atomic part is to inquire whether
 a it embodies materials, then we analyse and describe these;
 b and if it further has components, then we describe their sorts.

value

160. analyse_and_describe_atomic_part: NmVAL \rightarrow **Unit**
 160. analyse_and_describe_atomic_part(nm,val) \equiv
 160a. **if has_material**(nm,val)²³ **then observe_part_material_sort**(nm,val)²⁴ **end** ;
 160b. **if has_components**(nm,val)²⁵ **then observe_part_component_sort**(nm,val)²⁶ **end**

- 161 To analyse and describe a composite endurant of sort nm (and value val)
 a is to analyse if the sort has a concrete type
 b then we analyse and describe that concrete sort type
 c else we analyse and describe the abstract sort.

value

161. analyse_and_describe_composite_endurant: NmVAL \rightarrow **Unit**
 161. analyse_and_describe_composite_endurant(nm,val) \equiv
 161a. **if has_concrete_type**(nm,val)²⁷
 161b. **then observe_concrete_type**(nm,val)²⁸
 161c. **else observe_abstract_sorts**(nm,val)²⁹
 161a. **end**
 161. **pre is_composite**(nm,val)³⁰

We do not associate materials or components with composite parts.

3.4.3 Discussion of The Process Model

The above model lacks a formal understanding of the individual prompts as listed in Sect. 3.4.1; such an understanding is attempted in Sect. 3.8.

²² We formalise `observe_unique_identifier` in Sect. 3.8.3 on Page 128.

²² We formalise `is_atomic` in Sect. 3.8.2 on Page 125.

²² We formalise `is_composite` in Sect. 3.8.2 on Page 125.

²² We formalise `is_discrete` in Sect. 3.8.2 on Page 125.

²⁶ We formalise `has_material` in Sect. 3.8.2 on Page 126.

²⁶ We formalise `observe_part_material_sort` in Sect. 3.8.3 on Page 128.

²⁶ We formalise `has_components` in Sect. 3.8.2 on Page 126.

²⁶ We formalise `observe_part_component_sort` in Sect. 3.8.3 on Page 129.

²⁷ We formalise `has_concrete_type` in Sect. 3.8.2 on Page 126.

²⁷ We formalise `observe_concrete_type` in Sect. 3.8.3 on Page 127.

²⁷ We formalise `observe_part_sorts` in Sect. 3.8.3 on Page 127.

²⁷ We formalise `is_composite` in Sect. 3.8.2 on Page 125.

Termination

The sort name reservoir **vps** is “reduced” by one name in each iteration of the **while** loop of the `analyse_and_describe_endurants`, cf. Item 155b on Page 111, and is augmented by new part, material and component sort names in some iterations of that loop. We assume that (manifest) domains are finite, hence there are only a finite number of domain sorts. It remains to (formally) prove that the analysis & description process terminates.

Axioms and Proof Obligations

We have omitted, from Sect. 3.2, treatment of axioms concerning well-formedness of parts, materials and attributes and proof obligations concerning disjointedness of observed part and material sorts and attribute types. [70] exemplifies axioms and sketches some proof obligations.

Order of Analysis & Description: A Meaning of ‘ \oplus ’

The variables α ps, vps and τ can be defined to hold either sets or lists. The operator \oplus can be thought of as either set union (\cup and $[...] \equiv \{...\}$) — in which case the domain description text in τ is a set of domain description texts — or as list concatenation ($\hat{\ }^{\ } and $[...] \equiv \langle \dots \rangle$) of domain description texts. The list operator $\ell_1 \oplus \ell_2$ now has at least two interpretations: either $\ell_1 \hat{\ }^{\ } \ell_2$ or $\ell_2 \hat{\ }^{\ } \ell_1$. Thus, in the case of lists, the \oplus , i.e., $\hat{\ }^{\ }$, does not (suffix or prefix) append ℓ_2 elements already in ℓ_1 . The `sel_and_rem_Nm` function on Page 111 applies to the set interpretation. A list interpretation is:$

value

155b. `sel_and_rem_Nm: Unit \rightarrow Nm`

155b. `sel_and_rem_Nm() \equiv let nm = hd v ps in v ps := tl v ps; nm end; pre: v ps \neq $\langle \rangle$`

In the first case ($\ell_1 \hat{\ }^{\ } \ell_2$) the analysis and description process proceeds from the root, breadth first, In the second case ($\ell_2 \hat{\ }^{\ } \ell_1$) the analysis and description process proceeds from the root, depth first. .

Laws of Description Prompts

The domain ‘method’ outlined in the previous section suggests that many different orders of analysis & description may be possible. But are they? That is, will they all result in “similar” descriptions? If, for example, \mathcal{D}_a and \mathcal{D}_b are two domain description prompts where \mathcal{D}_a and \mathcal{D}_b can be pursued in any order will that yield the same description? And what do we mean by ‘can be pursued in any order’, and ‘same description’? Let us assume that sort P decomposes into sorts P_a and P_b (etcetera). Let us assume that the domain description prompt \mathcal{D}_a is related to the description of P_a and \mathcal{D}_b to P_b . Here we would expect \mathcal{D}_a and \mathcal{D}_b to commute, that is $\mathcal{D}_a; \mathcal{D}_b$ yields same result as does $\mathcal{D}_b; \mathcal{D}_a$. In [51] we made an early exploration of such laws of domain description prompts. To answer these questions we need a reasonably precise model of domain prompts. We attempt such a model in Sect. 3.8. But we do not prove theorems.

3.5 A Domain Analyser’s & Describer’s Domain Image

Assumptions: We assume that the domain analysers cum describers are well educated and well trained in the domain analysis & description techniques such as laid out in [70]. This assumption entails that the domain analysis & description development process is structured in sequences of alternating (one or more) analysis prompts and description prompts. We refer to Footnote 3 (Page 107) as well as to the discussion, “Towards a methodology of manifest domain analysis & description” of [70, Sect. 1.6]. We further assume that the domain analysers cum describers makes repeated attempts to analyse & describe a domain. We assume, further, that it is “the same domain” that is being analysed & described – two, three or more times, “all-over”, before commitment is made to attempt a – hopefully – final analysis & description²⁸, from

²⁸ – and if that otherwise planned, final analysis & description is not satisfactory, then yet one more iteration is taken.

“scratch”, that is, having “thrown away”, previous drafts²⁹. We then make the further assumption, as this iterative analysis & description process proceeds, from iteration i to $i + 1$, that each and all members of the analysis & description group are forming, in their minds (i.e., brains) an “image” of the domain being analysed. As iterations proceed one can then say that what is being analysed & described increasingly becomes this ‘image’ as much as it is being the domain — which we assume is not changing across iterations. The iterated descriptions are now postulated to converge: a “final” iteration “differs” only “immaterially.” from the description of the “previous” iteration.

• • •

The Domain Engineer’s Image of Domains: In the opening (‘Assumptions’) of this section, i.e., above, we hinted at “an image”, in the minds of the domain analysers & describers, of the domain being researched and for which a description document is being engineered. In this paragraph we shall analyse what we mean by such a image. Since the analysis & description techniques are based on applying the analysis and description prompts (reviewed in Sect. 3.2) we can assume that the image somehow relates to the ‘ontology’ of the domain entities, whether endurants or perdurants, such as graphed in Fig. 1.4. Rather than further investigating (i.e., analysing / arguing) the form of this, until now, vague notion, we simply conjecture that the image is that of an ‘**abstract syntax of domain types**’.

• • •

The Iterative Nature of The Description Process: Assume that the domain engineers are analysing & describing a particular endurant; that is, as we shall understand it, are examining a given endurant node in the *domain description tree* ! The **domain description tree** is defined by the facts that composite parts have sub-parts which may again be composite (tree branches), ending with atomic parts (the leaves of the tree) but not “circularly”, i.e. recursively ■

To make this claim: *the domain analysers cum describers are examining a given endurant node in the domain description tree* amounts to saying that *the domain engineers have in their mind a reasonably “stable” “picture” of a domain in terms of a domain description tree.*

We need explain this assumption. In this assumption there is “buried” an understanding that the domain analysers cum describers during the — what we can call “the final” — domain analysis & description process, that leads to a “deliverable” domain description, are not investigating the domain to be described for the first time. That is, we certainly assume that any “final” domain analysis & description process has been preceded by a number of iterations of “trial” domain analysis & description processes.

Hopefully this iteration of experimental domain analysis & description processes converges. Each iteration leads to some domain description, that is, some domain description tree. A first iteration is thus based on a rather incomplete domain description tree which, however, “quickly” emerges into a less incomplete one in that first iteration. When the domain engineers decide that a “final” iteration seems possible then a “final” description emerges If acceptable, OK, otherwise yet an “final” iteration must be performed. Common to all iterations is that the domain analysers cum describers have in mind some more-or-less “complete” domain description tree and apply the prompts introduced in Sect. 3.4.

3.6 Domain Types

There are two kinds of types associated with domains: the syntactic types of endurant descriptions, and the semantic types of endurant values.

²⁹ It may be useful, though, to keep a list of the names of all the endurant parts and their attribute names, should the group members accidentally forget such endurants and attributes: at least, if they do not appear in later document iterations, then it can be considered a deliberate omission.

3.6.1 Syntactic Types: Parts, Materials and Components

In this section we outline an ‘**abstract syntax of domain types**’. In Sect. 3.6.1 we introduce the concept of sort names. Then, in Sects. 3.6.1–3.6.1, we describe the syntax of part, material and component types. Finally, in Sects. 3.6.1–3.6.1, we analyse this syntax with respect to a number of well-formedness criteria.

Syntax of Part, Material and Component Sort Names

162 There is a further undefined sort, N , of tokens (which we shall consider atomic and the basis for forming names).

163 From these we form three disjoint sets of sort names:

- a part sort names,
- b material sort names and
- c component sort names,

162 N

163a $PNm :: mkPNm(N)$

163b $MNm :: mkMNm(N)$

163c $KNm :: mkKNm(N)$

An Abstract Syntax of Domain Endurants

164 We think of the types of parts, materials and components to be a map from their type names to respective type expressions.

165 Thus part types map part sort names into part types;

166 material types map material sort names into material types; and

167 component types map components sort names into component types.

168 Thus we can speak of endurant types to be either part types or material types or component types.

169 A part type expression is either an atomic part type expression or is a composite part type expression or is a concrete composite part type expression.

170 An atomic part type expression consists of a type expression for the qualities of the atomic part and, optionally, a material type name or a component type name.

171 An abstract composite part type expression consists of a type expression for the qualities of the composite part and a finite set of one or more part type names.

172 A concrete composite part type expression consists of a type expression for the qualities of the part and a part sort name standing for a set of parts of that sort.

173 A material part type expression consists of of a type expression for the qualities of the material and an optional part type name.

174 We omit consideration of component types.

Endurants: Syntactic Types

164 $TypDef = PTypes \cup MTypes \cup KTypes$

165 $PTypes = PNm \xrightarrow{\#} PaTyp$

166 $MTypes = MNm \xrightarrow{\#} MaTyp$

167 $KTypes = KNm \xrightarrow{\#} KoTyp$

168 $ENDType = PaTyp \mid MaTyp \mid KoTyp$

169 $PaTyp == AtPaTyp \mid AbsCoPaTyp \mid ConCoPaTyp$

170 $AtPaTyp :: mkAtPaTyp(s_{qs}:PQ, s_{omkn}:\{\mid \text{nil} \mid\} \mid MNn \mid KNm)$

171 $AbsCoPaTyp :: mkAbsCoPaTyp(s_{qs}:PQ, s_{pns}:PNm\text{-set})$

171 **axiom** $\forall mkAbsCoPaTyp(pq, pns): AbsCoPaTyp \cdot pns \neq \{\}$

172 $ConCoPaTyp :: mkConCoPaTyp(s_{qs}:PQ, s_p:PNm)$

173 $MaTyp :: mkMaTyp(s_{qs}:MQ, s_{opn}:\{\mid \text{nil} \mid\} \mid PNm)$

174 $KoTyp :: mkKoTyp(s_{qs}:KQ)$

Quality Types

- 175 There are three aspects to part qualities: the type of the part unique identifiers, the type of the part mereology, and the name and type of attributes.
- 176 The type unique part identifiers is a not further defined atomic quantity.
- 177 A part mereology is either "nil" or it is an expression over part unique identifiers, where such expressions are those of either simple unique identifier tokens, or of set, or otherwise over simple unique identifier tokens, or ..., etc.
- 178 The type of attributes pairs distinct attribute names with attribute types —
- 179 both of which we presently leave further undefined.
- 180 Material attributes is the only aspect to material qualities.
- 181 Components have unique identifiers. Component attribute types are left undefined.

Qualities: Syntactic Types

- 175 $PQ = s_ui:UI \times s_me:ME \times s_attrs:ATRS\}$
- 176 UI
- 177 $ME == \text{"nil"} | mkUI(s_ui:UI) | mkUIset(s_uil:UI) | \dots$
- 178 $ATRS = ANm \rightarrow_{\#} ATyp$
- 179 $ANm, ATyp$
- 180 $MQ = s_attrs:ATRS$
- 181 $KQ = s_uid:UI \times s_attrs:ATRS$

It is without loss of generality that we do not distinguish between part and material attribute names and types. Material and component attributes do not refer to any part or any other material and component attributes.

Well-formed Syntactic Types

Well-formed Definitions

- 182 We need define an auxiliary function, names, which, given an endurant type expression, yields the sort names that are referenced immediately by that type.
- If the endurant type expression is that of an atomic part type then the sort name is that of its optional component sort.
 - If an abstract composite part type then the sort names of its parts.
 - If a concrete composite part type then the sort name is that of the sort of its set of parts.
 - If a material type then sort name is that of the sort of its optional parts.
 - Component sorts have no references to other sorts.

value

182. $names: TypDef \rightarrow (PNm|MNm|KNm) \rightarrow (PNm|MNm|KNm)\text{-set}$
182. $names(td)(n) \equiv$
182. $\cup \{ ns \mid ns:(PNm|MNm|KNm)\text{-set} \cdot$
182. **case** $td(n)$ **of**
- 182a. $mkAtPaTyp(_,n') \rightarrow ns=\{n'\},$
- 182b. $mkAbsCoPaTyp(_,ns') \rightarrow ns=ns',$
- 182c. $mkConCoPaTyp(_,pn) \rightarrow ns=\{pn\},$
- 182d. $mkMaTyp(_,n') \rightarrow ns=\{n'\},$
- 182e. $mkKoTyp(_) \rightarrow ns=\{\}$
182. **end** $\}$

183 Endurant sort names being referenced in part types, PaTyp, in material types, MaTyp, and in component types, KoTyp, of the `typedef:Typdef` definition, *must be defined in* the defining set, **dom** `typedef`, of the `typedef:Typdef` definition.

value

183. `wf_TypDef_1: TypDef → Bool`

183. `wf_TypDef_1(td) ≡ ∀ n:(PNm|MNm|CNm)•n ∈ dom td ⇒ names(td)(n) ⊆ dom td`

Perhaps Item 183. should be sharpened:

184 from “*must be defined in*” [183.] to “*must be equal to*”:

184. $\wedge \forall n:(PNm|MNm|CNm) \cdot n \in \mathbf{dom} \text{ td} \Rightarrow \text{names}(\text{td})(n) = \mathbf{dom} \text{ td}$

No Recursive Definitions

185 Type definitions must not define types recursively.

a A type definition, `typedef:TypDef`, defines, typically composite part sorts, named, say, n , in terms of other part (material and component) types. This is captured in the

- `mncs` (Item 170),
- `pns` (Item 171),
- `p` (Item 172) and
- `pns` (Item 173),

selectable elements of respective type definitions. These elements identify type names of materials and components, parts, a part, and parts, respectively. None of these names may be n .

b The identified type names may further identify type definitions none of whose selected type names may be n .

c And so forth.

value

185. `wf_TypDef_2: TypDef → Bool`

185. `wf_TypDef_2(typdef) ≡ ∀ n:(PNm|MNm)•n ∈ dom typdef ⇒ n ∉ type_names(typdef)(n)`

185a. `type_names: TypDef → (PNm|MNm) → (PNm|MNm)-set`

185a. `type_names(typdef)(nm) ≡`

185b. `let ns = names(typdef)(nm) ∪ { names(typdef)(n) | n:(PNm|MNm) • n ∈ ns } in`

185c. `nm ∉ ns end`

`ns` is the least fix-point solution to the recursive definition of `ns`.

3.6.2 Semantic Types: Parts, Materials and Components

Part, Material and Component Values

We define the values corresponding to the type definitions of Items 162.–181, structured as per type definition Item 168 on Page 116.

186 An enduring value is either a part value, a material values or a component value.

187 A part value is either the value of an atomic part, or of an abstract composite part, or of a concrete composite part.

188 A atomic part value has a part quality value and, optionally, either a material or a possibly empty set of component values.

189 An abstract composite part value has a part quality value and of at least (hence the **axiom**) of

- 190 one or more (distinct part type) part values.
 191 A concrete composite part value has a part quality value and a set of part values.
 192 A material value has a material quality value (of material attributes) and a (usually empty) finite set of part values.
 193 A component value has a component quality value (of a unique identifier and component attributes).

Endurant Values: Semantic Types

- 186 $ENDVAL = PVAL \mid MVAL \mid KVAL$
 187 $PVAL == AtPaVAL \mid AbsCoPVAL \mid ConCoPVAL$
 188 $AtPaVAL :: mkAtPaVAL(s_qval:PQVAL, s_omkvals:({}|"nil"|}) \mid MVAL \mid KVAL\text{-set})$
 189 $AbsCoPVAL :: mkAbsCoPaVAL(s_qval:PQVAL, s_pvals:(PNm \rightarrow_{\vec{m}} PVAL))$
 190 **axiom** $\forall mkAbsCoPaVAL(pqs, ppm): AbsCoPVAL \cdot ppm \neq []$
 191 $ConCoPVAL :: mkConCoPaVAL(s_qval:PQVAL, s_pvals:PVAL\text{-set})$
 192 $MVAL :: mkMaVAL(s_qval:MQVAL, s_pvals:PVAL\text{-set})$
 193 $KVAL :: mkKoVAL(s_qval:KQVAL)$

Quality Values

- 194 A part quality value consists of three qualities:
 195 a unique identifier type name, resp. value, which are both further undefined (atomic value) tokens;
 196 a mereology expression, resp. value, which is either a single unique identifier (type, resp.) value, or a set of such unique identifier (types, resp.) values, or ...; and
 197 an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.
 198 In this chapter we leave attribute type names and attribute values further undefined.
 199 A material quality value consists just of an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.
 200 A component quality value consists of a pair: a unique identifier value and an aggregate of attribute values, modeled here as a map from attribute type names to attribute values.

Qualities: Semantic Types

- 194 $PQVAL = UIVAL \times MEVAL \times ATTRVALS$
 195 $UIVAL$
 196 $MEVAL == mkUIVAL(s_ui:UIVAL) \mid mkUIVALset(s_uis:UIVAL\text{-set}) \mid \dots$
 197 $ATTRVALS = ANm \rightarrow_{\vec{m}} AVAL$
 198 $ANm, AVAL$
 199 $MQVAL = ATTRVALS$
 200 $KQVAL = UIVAL \times ATTRVALS$

We have left to define the values of attributes. For each part and material attribute value we assume a finite set of values. And for each unique identifier type (i.e., for each UI) we likewise assume a finite set of unique identifiers of that type. The value sets may be large. These assumptions help secure that the set of part, material and component values are also finite.

Type Checking

For part, material and component qualities we postulate an overloaded, simple type checking function, `type_of`, that applies to unique identifier values, $uiv:UIVAL$, and yield their unique identifier type name, $ui:UI$, to mereology values, $mev:MEVAL$, and yield their mereology expression, $me:ME$, and to attribute values, $AVAL$ and $ATTRSVAL$, and yield their types: $ATyp$, respectively $(ANm \rightarrow_{\vec{m}} AVAL) \rightarrow (ANm \rightarrow_{\vec{m}} ATyp)$. Since we have let undefined both the syntactic type of attributes types, $ATyp$, and the semantic type of attribute values, $AVAL$, we shall leave `type_of` further unspecified.

value `type_of`: $(UIVAL \rightarrow UI) \mid (MEVAL \rightarrow ME) \mid (AVAL \rightarrow ATyp) \mid ((ANm \rightarrow_{\vec{m}} AVAL) \rightarrow (ANm \rightarrow_{\vec{m}} ATyp))$

The definition of the syntactic type of attributes types, $ATyp$, and the semantic type of attribute values, $AVAL$, is a simple exercise in a first-year programming language semantics course.

3.7 From Syntax to Semantics and Back Again !

The two syntaxes of the previous section: that of the *syntactic domains*, formula Items 162–181 (Pages 116–117), and that of the *semantic domains*, formula Items 186–200 (Pages 119–119), are not the syntaxes of domain descriptions, but of some aspects common to all domain descriptions developed according to the calculi of this chapter. The *syntactic domain* formulas underlie (“are common to”, i.e., “abstracts”) aspects of all domain descriptions. The *semantic domain* formulas underlay (“are common to”, i.e., “abstracts”) aspects of the meaning of all domain descriptions. These two syntaxes, hence, are, so-to-speak, in the minds of the domain engineer (i.e., the analyser cum describer) while analysing the domain.

3.7.1 The Analysis & Description Prompt Arguments

The domain engineer analyse & describe endurants on the basis of a sort name i.e., a piece of syntax, $nm:Nm$, and an endurant value, i.e. a “piece” of semantics, $val:VAL$, that is, the arguments, $(nm, l:nm)$, of the analysis and description prompts of Sect. 3.4. Those two quantities are what the domain engineer are “operating” with, i.e., are handling: One is tangible, i.e. can be noted (i.e., “scribbled down”), the other is “in the mind” of the analysers cum describers. We can relate the two in terms of the two syntaxes, the syntactic types, and the meaning of the semantic types. But first some “preliminaries”.

3.7.2 Some Auxiliary Maps: Syntax to Semantics and Semantics to Syntax

We define two kinds of map types:

201 $Nm_to_ENDVALS$ are maps from endurant sort names to respective sets of all corresponding endurant values of, and

202 $ENDVAL_to_Nm$ are maps from endurant values to respective sort names.

type

201. $Nm_to_ENDVALS = (PNm \rightarrow_{\#} PVAL\text{-set}) \cup (MNm \rightarrow_{\#} MVAL\text{-set}) \cup (KNm \rightarrow_{\#} KVAL\text{-set})$

202. $ENDVAL_to_Nm = (PVAL \rightarrow_{\#} PNm) \cup (MVAL \rightarrow_{\#} MNm) \cup (KVAL \rightarrow_{\#} KNm)$

We can derive values of these map types from type definitions:

203 a function, $typval$, from type definitions, $typdef: TypDef$ to $Nm_to_ENDVALS$, and

204 a function $valtyp$, from $Nm_to_ENDVALS$, to $ENDVAL_to_Nm$.

value

203. $typval: TypDef \xrightarrow{\sim} Nm_to_ENDVALS$

204. $valtyp: Nm_to_ENDVALS \xrightarrow{\sim} ENDVAL_to_Nm$

205 The $typval$ function is defined in terms of a meaning function M (let $\rho: ENV$ abbreviate $Nm_to_ENDVALS$:

205. $M: (PaTyp \rightarrow ENV \xrightarrow{\sim} PVAL\text{-set}) | (MaTyp \rightarrow ENV \xrightarrow{\sim} MVAL\text{-set}) | (KoTyp \rightarrow ENV \xrightarrow{\sim} KVAL\text{-set})$

203. $typval(td) \equiv \mathbf{let} \rho = [n \mapsto M(td(n))(\rho)] | n: (PNm | MNm | KNm) \cdot n \in \mathbf{dom} td \mathbf{ in} \rho \mathbf{ end}$

204. $valtyp(\rho) \equiv [v \mapsto n | n: (PNm | MNm | KNm), v: (PVAL | MVAL | KVAL) \cdot n \in \mathbf{dom} \rho \wedge v \in \rho(n)]$

The environment, ρ , of $typval$, Item 203, is the least fix point of the recursive equation

- 203. $\mathbf{let} \rho = [n \mapsto M(td(n))(\rho)] | n: (PNm | MNm | KNm) \cdot n \in \mathbf{dom} td \mathbf{ in} \dots$

The M function is defined next.

3.7.3 M: A Meaning of Type Names

Preliminaries

The `typval` function provides for a homomorphic image from `TypDef` to `TypNm_to_VALS`. So, the narrative below, describes, item-by-item, this image. We refer to formula Items 203 and 205. The definition of `M` is decomposed into five sub-definitions, one for each kind of enduring type:

- Atomic parts: `mkAtPaTyp(s_qs:(UI×ME×ATRS),s_omkn:({|" nil" |}|MNn|KNm))`, Items 206;
- Abstract composite parts: `mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-set)`, 207 on the following page;
- Concrete composite parts: `mkConCoPaTyp(s_qs:PQ,s_p:PNm)`, Items 208 on the next page;
- Materials: `mkMaTyp(s_qs:MQ,s_opn:({|" nil" |}|PNm))`, Items 209 on Page 123; and
- Components: `mkKoTyp(s_qs:KQ)`, Items 210 on Page 123.

We abbreviate, by ENV, the `M` function argument, ρ , of type: `Nm_to_ENDVALS`.

Atomic Parts

- 206 The meaning of an atomic part type expression,
 Item 170. `mkAtPaTyp((ui,me,attrs),omkn)`
 in `mkAtPaTyp(s_qs:PQ,s_omkn:({|" nil" |}|MNn|KNm))`,
 is the set of all atomic part values,
 Items 188., 194., 197. `mkAtPaVAL((uiv,mev,attrvals),omkval)`
 in `mkAtPaVAL(s_qval:(UIVAL×MEVAL×(ANm →n AVAL)),`
`s_omkvals:({|" nil" |}|MVAL|KVAL-set))`.
 a `uiv` is a value in `UIVAL` of type `ui`,
 b `mev` is a value in `MEVAL` of type `me`,
 c `attrvals` is a value in `(ANm →n AVAL)` of type `(ANm →n ATyp)`, and
 d `omkvals` is a value in `({|" nil" |}|MVAL|KVAL-set)`:
 i either `' nil '`,
 ii or one material value of type `MNm`,
 iii or a possibly empty set of component values, each of type `KNm`.
206. `M`: `mkAtPaTyp((UI×ME×(ANm →n ATyp))×({|" nil" |}|MVAL|KVAL-set))→ENV→PVAL-set`
 206. `M(mkAtPaTyp((ui,me,attrs),omkn))(ρ) ≡`
 206. `{ mkATPaVAL((uiv,mev,attrval),omkvals) |`
 206a. `uiv:UIVAL•type_of(uiv)=ui,`
 206b. `mev:MEVAL•type_of(mev)=me,`
 206c. `attrval:(ANm →n AVAL)•type_of(attrval)=attrs,`
 206d. `omkvals: case omkn of`
 206(d)i. `" nil" → " nil" ,`
 206(d)ii. `mkMNn(⊔) → mval:MVAL•type_of(mval)=omkn,`
 206(d)iii. `mkKNm(⊔) → kvals:KVAL-set•kvals⊆{kv|kv:KVAL•type_of(kval)=omkn}`
 206d. `end }`

Formula terms 206a–206(d)iii express that any applicable `uiv` is combined with any applicable `mev` is combined with any applicable `attrval` is combined with any applicable `omkvals`.

Abstract Composite Parts

- 207 The meaning of an abstract composite part type expression,
 Item 171. `mkAbsCoPaTyp((ui,me,attrs),pns)`
 in `mkAbsCoPaTyp(s_qs:PQ,s_pns:PNm-set)`,

is the set of all abstract, composite part values,

Items 189., 194., 197., $\text{mkAbsCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals})$

in $\text{mkAbsCoPaVAL}(s_qval: (\text{UIVAL} \times \text{MEVAL} \times (\text{ANm} \rightarrow_{\#} \text{AVAL})), s_pvals: (\text{PNm} \rightarrow_{\#} \text{PVAL}))$.

a uiv is a value in UIVAL of type ui : UI ,

b mev is a value in MEVAL of type me : ME ,

c attrvals is a value in $(\text{ANm} \rightarrow_{\#} \text{AVAL})$ of type $(\text{ANm} \rightarrow_{\#} \text{ATyp})$, and

d pvals is a map of part values in $(\text{PNm} \rightarrow_{\#} \text{PVAL})$, one for each name, $\text{pn}:\text{PNm}$, in pns such that these part values are of the type defined for pn .

207. $M: \text{mkAbsCoPaTyp}((\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\#} \text{ATyp})), \text{PNm}\text{-set}) \rightarrow \text{ENV} \xrightarrow{\sim} \text{PVAL}\text{-set}$

207. $M(\text{mkAbsCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pns}))(\rho) \equiv$

207. $\{ \text{mkAbsCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals}) \mid$

207a. $\text{uiv}:\text{UIVAL} \cdot \text{type_of}(\text{uiv})=\text{ui}$

207b. $\text{mev}:\text{MEVAL} \cdot \text{type_of}(\text{mev})=\text{me}$,

207c. $\text{attrvals}:(\text{ANm} \rightarrow_{\#} \text{ATyp}) \cdot \text{type_of}(\text{attrvals})=\text{attrs}$,

207d. $\text{pvals}:(\text{PNm} \rightarrow_{\#} \text{PVAL}) \cdot \text{pvals} \in \{ [\text{pn} \mapsto \text{pval} \mid \text{pn}:\text{PNm}, \text{pval}:\text{PVAL} \cdot \text{pn} \in \text{pns} \wedge \text{pval} \in \rho(\text{pn})] \}$ }

Concrete Composite Parts

208 The meaning of a concrete composite part type expression, Item 172.

$\text{mkConCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pn})$

in $\text{mkConCoPaTyp}(s_qs: (\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\#} \text{ATyp})), s_pn:\text{PNm})$,

is the set of all concrete, composite set part values,

Item 191. $\text{mkConCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals})$

in $\text{mkConCoPaVAL}(s_qval: (\text{UIVAL} \times \text{MEVAL} \times (\text{ANm} \rightarrow_{\#} \text{AVAL})), s_pvals:\text{PVAL}\text{-set})$.

a uiv is a value in UIVAL of type ui ,

b mev is a value in MEVAL of type me ,

c attrvals is a value in $(\text{ANm} \rightarrow_{\#} \text{AVAL})$ of type attrs , and

d pvals is a $[\text{ny}]$ value in $\text{PVAL}\text{-set}$ where each part value in pvals is of the type defined for pn .

208. $M: \text{mkConCoPaTyp}((\text{UI} \times \text{ME} \times (\text{ANm} \rightarrow_{\#} \text{ATyp})) \times \text{PNm}) \rightarrow \text{ENV} \xrightarrow{\sim} \text{PVAL}\text{-set}$

208. $M(\text{mkConCoPaTyp}((\text{ui}, \text{me}, \text{attrs}), \text{pn}))(\rho) \equiv$

208. $\{ \text{mkConCoPaVAL}((\text{uiv}, \text{mev}, \text{attrvals}), \text{pvals}) \mid$

208a. $\text{uiv}:\text{UIVAL} \cdot \text{type_of}(\text{uiv})=\text{ui}$,

208b. $\text{mev}:\text{MEVAL} \cdot \text{type_of}(\text{mev})=\text{me}$,

208c. $\text{attrvals}:(\text{ANm} \rightarrow_{\#} \text{AVAL}) \cdot \text{type_of}(\text{attrvals})=\text{attrs}$,

208d. $\text{pvals}:\text{PVAL}\text{-set} \cdot \text{pvals} \subseteq \rho(\text{pn}) \}$

Materials

209 The meaning of a material type, 173.,

expression $\text{mkMaTyp}(\text{mq}, \text{pn})$ in $\text{mkMaTyp}(s_qs:\text{MQ}, s_pn:\text{PNm})$

is the set of values $\text{mkMaVAL}(\text{mqval}, \text{ps})$

in $\text{mkMaVAL}(s_qval:\text{MQVAL}, s_pvals:\text{PVAL}\text{-set})$ such that

a mqval in MQVAL is of type mq , and

b ps is a set of part values all of type pn .

209. $M: \text{mkMaTyp}(s_mq: (\text{ANm} \rightarrow_{\#} \text{ATyp}), s_pn:\text{PNm}) \rightarrow \text{ENV} \xrightarrow{\sim} \text{MVAL}\text{-set}$

209. $M(\text{mq}, \text{pn})(\rho) \equiv$

209. $\{ \text{mkMVAL}(\text{mqval}, \text{ps}) \mid$

209a. $\text{mqval}:\text{MVAL} \cdot \text{type_of}(\text{mqval})=\text{mq}$,

209b. $\text{ps}:\text{PVAL}\text{-set} \cdot \text{ps} \subseteq \rho(\text{pn}) \}$

Components

- 210 The meaning of a component type, 174., expression $\text{mkKoType}(ui, \text{attrs})$ in $\text{mkKoTyp}(s_qs:(s_uid:U \times s_attrs:ATRS))$ is the set of values, 173., $\text{mkKQVAL}(uiv, \text{attrsv})$ in, 193, $\text{mkKoVAL}(s_qval:(uiv, \text{attrsv}))$.
- a uiv is in $UIVAL$ of type ui , and
 - b attrsv is in $ATTRSVAL$ of type attrs .
210. $M: \text{mkKoTyp}(U \times ATRS) \rightarrow ENV \rightarrow KVAL\text{-set}$
 210. $M(\text{mkKoType}(ui, \text{attrs}))(\rho) \equiv$
 210. $\{ \text{mkKoVAL}(uiv, \text{attrsv}) \mid$
 210a. $uiv:UIVAL \cdot \text{type_of}(uiv) = ui,$
 210b. $\text{attrsv}:ATTRSVAL \cdot \text{type_of}(\text{attrsv}) = \text{attrs} \}$

3.7.4 The ι Description Function

We can now define the meaning of the syntactic clause:

- $\iota Nm:Nm$

211 $\iota Nm:Nm$ “chooses” an arbitrary value from amongst the values of sort Nm :

value

211. $\iota nm:Nm \equiv \text{iota}(nm)$
 211. $\text{iota}: Nm \rightarrow \text{TypDef} \rightarrow VAL$
 211. $\text{iota}(nm)(td) \equiv \mathbf{let\ val:(PVAL|MVAL|KVAL) \cdot val \in (\text{typval}(td))(nm) \mathbf{in\ val\ end}}$

Discussion

From the above two functions, **typval** and **valtyp**, and the type definition “table” $td:\text{TypDef}$ and “argument value” $val:PVAL|MVAL|KVAL$, we can form some expressions. One can understand these expressions as, for example reflecting the following analysis situations:

- **typval**(td): From the type definitions we form a map, by means of function **typval**, from sort names to the set of all values of respective sorts: $Nm_to_ENDVALS$.
That is, whenever we, in the following, as part of some formula, write **typval**(td), then we mean to express that the domain engineer forms those associations, in her mind, from sort names to usually very large, non-trivial sets of endurant values.
- **valtyp**(**typval**(td)): The domain analyser cum describer “inverts”, again in his mind, the **typval**(td) into a simple map, $ENDVAL_to_Nm$, from single endurant values to their sort names.
- (**valtyp**(**typval**(td)))(val): The domain engineer now “applies”, in her mind, the simple map (above) to an endurant value and obtains its sort name $nm:Nm$.
- $td((\mathbf{valtyp}(\mathbf{typval}(td)))(val))$: The domain analyser cum describer then applies the type definition “table” $td:\text{TypDef}$ to the sort name $nm:Nm$ and obtains, in his mind, the corresponding type definition, $PaTyp|MaTyp|KoTyp$.

We leave it to the reader to otherwise get familiarised with these expressions.

3.8 A Formal Description of a Meaning of Prompts

3.8.1 On Function Overloading

In Sect. 3.4 the analysis and description prompt invocations were expressed as

- $is_XXX(e)$, $has_YYY(e)$ and $observe_ZZZ(e)$

where XXX , YYY , and ZZZ were appropriate entity sorts and e were appropriate endurants (parts, components and materials). The function invocations, $is_XXX(e)$, etcetera, takes place in the context of a type definition, $td: TypDef$, that is, instead of $is_XXX(e)$, etc. we get

- $is_XXX(e)(td)$, $has_YYY(e)(td)$ and $observe_ZZZ(e)(td)$.

We say that the functions is_XXX , etc., are “lifted”.

3.8.2 The Analysis Prompts

The analysis is expressed in terms of the analysis prompts:

- | | |
|-------------------------------|------------------------------------|
| a. is_entity , 10 | l. $is_living_species$, 15 |
| b. $is_endurant$, 11 | m. is_plant , 15 |
| c. $is_perdurant$, 11 | n. is_animal , 16 |
| d. $is_discrete$, 11 | o. is_human , 16 |
| e. $is_continuous$, 11 | p. $has_components$, 17 |
| f. $is_physical_part$, 12 | q. $has_materials$, 17 |
| g. $is_living_species$, 12 | r. $is_artefact$, 17 |
| h. $is_structure$, 13 | s. $observe_endurant_sorts$, 18 |
| i. is_part , 14 | t. $has_concrete_type$, 21 |
| j. is_atomic , 15 | u. $has_mereology$, 26 |
| k. $is_composite$, 15 | v. $attribute_types$, 29 |

The analysis takes place in the context of a type definition “image”, $td: TypDef$, in the minds of the domain engineers.

is_entity

The is_entity predicate is meta-linguistic, that is, we cannot model it on the basis of the type systems given in Sect. 3.6. So we shall just have to accept that.

$is_endurant$

See analysis prompt definition 2 on Page 11 and Formula Item 158a on Page 112.

value

$is_endurant: Nm \times VAL \rightarrow TypDef \xrightarrow{\sim} \mathbf{Bool}$
 $is_endurant(_, val)(td) \equiv val \in \mathbf{dom} \text{ valtyp}(\text{typval}(td)); \mathbf{pre}: VAL \text{ is any value type}$

$is_discrete$

See analysis prompt definition 4 on Page 11 and Formula Item 159d on Page 112.

value

$is_discrete: NmVAL \rightarrow TypDef \xrightarrow{\sim} \mathbf{Bool}$
 $is_discrete(_, val)(td) \equiv (is_PaTyp | is_CoTyp)(td((\text{valtyp}(\text{typval}(td))))(val))$

is_part

See analysis prompt definition 9 on Page 14 and Formula Item 158a on Page 112.

value

$$\begin{aligned} \text{is_part}: \text{NmVAL} &\rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool} \\ \text{is_part}(_, \text{val})(\text{td}) &\equiv \text{is_PaTyp}(\text{td}(\text{valtyp}(\text{typval}(\text{td}))(\text{val}))) \end{aligned}$$
is_material [≡ is_continuous]

See analysis prompt definition 5 on Page 11 and Formula Item 158b on Page 112.

We remind the reader that $\text{is_continuous} \equiv \text{is_material}$.

value

$$\begin{aligned} \text{is_material}: \text{NmVAL} &\rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool} \\ \text{is_material}(_, \text{val})(\text{td}) &\equiv \text{is_MaTyp}(\text{td}(\text{valtyp}(\text{typval}(\text{td}))(\text{val}))) \end{aligned}$$
is_component

See analysis prompt definition 16 on Page 17 and Formula Item 158c on Page 112.

value

$$\begin{aligned} \text{is_component}: \text{NmVAL} &\rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool} \\ \text{is_component}(_, \text{val})(\text{td}) &\equiv \text{is_CoTyp}(\text{td}(\text{valtyp}(\text{typval}(\text{td}))(\text{val}))) \end{aligned}$$
is_atomic

See analysis prompt definition 10 on Page 15 and Formula Item 159b on Page 112.

value

$$\begin{aligned} \text{is_atomic}: \text{NmVAL} &\rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool} \\ \text{is_atomic}(_, \text{val})(\text{td}) &\equiv \text{is_AtPaTyp}(\text{td}(\text{valtyp}(\text{typval}(\text{td}))(\text{val}))) \end{aligned}$$
is_composite

See analysis prompt definition 11 on Page 15 and Formula Item 159c on Page 112.

value

$$\begin{aligned} \text{is_composite}: \text{NmVAL} &\rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool} \\ \text{is_composite}(_, \text{val})(\text{td}) &\equiv (\text{is_AbsCoPaTyp} | \text{is_ConCoPaTyp})(\text{td}(\text{valtyp}(\text{typval}(\text{td}))(\text{val}))) \end{aligned}$$
has_concrete_type

See analysis prompt definition 20 on Page 21 and Formula Item 161a on Page 113.

value

$$\begin{aligned} \text{has_concrete_type}: \text{NmVAL} &\rightarrow \text{TypDef} \xrightarrow{\sim} \mathbf{Bool} \\ \text{has_concrete_type}(_, \text{val})(\text{td}) &\equiv \text{is_ConCoPaTyp}(\text{td}(\text{valtyp}(\text{typval}(\text{td}))(\text{val}))) \end{aligned}$$

has_mereology

See analysis prompt definition 21 on Page 26 and Formula Item 156 on Page 112.

value

has_mereology: NmVAL \rightarrow TypDef $\xrightarrow{\sim}$ **Bool**
 has_mereology(_,val)(td) \equiv s_me(td((valtyp(typval(td)))(val))) \neq "nil"

has_materials

See analysis prompt definition 17 on Page 17 and Formula Item 160a on Page 113.

value

has_material: NmVAL \rightarrow TypDef $\xrightarrow{\sim}$ **Bool**
 has_material(_,val)(td) \equiv is_MNm(s_omkn(td((valtyp(typval(td)))(val))))
pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))

has_components

See analysis prompt definition 16 on Page 17 and Formula Item 160b on Page 113.

value

has_components: NmVAL \rightarrow TypDef $\xrightarrow{\sim}$ **Bool**
 has_components(_,val)(td) \equiv is_KNm(s_omkn(td((valtyp(typval(td)))(val))))
pre: is_AtPaTyp(td((valtyp(typval(td)))(val)))

3.8.3 The Description Prompts

These are the domain description prompts to be defined:

- | | |
|---------------------------------|-----------------------------------|
| [1] observe_endurant_sorts, 19 | [5] observe_unique_identifier, 25 |
| [2] observe_part_type, 21 | [6] observe_mereology, 27 |
| [3] observe_component_sorts, 22 | [7] observe_attributes, 29 |
| [4] observe_material_sorts, 24 | |

A Description State

In addition to the analysis state components α ps and ν ps there is now an additional, the description text state component.

212 Thus a global variable τ will hold the (so far) generated (in this case only) formal domain description text.

variable

212. $\tau := []$ **Text-set**

We shall explain the use of [...]s and the operations of \setminus and \oplus on the above variables in Sect. 3.4.3 on Page 114.

observe_part_sorts

See description prompt definition 1 on Page 19 and Formula Item 161c on Page 113.

value

```

observe_part_sorts: NmVAL → TypDef → Unit
observe_part_sorts(nm,val)(td) ≡
  let mkAbsCoPaTyp(⟦, {P1, P2, ..., Pn}⟩) = td((valtyp(typval(td)))(val)) in
    τ := τ ⊕ [ " type P1, P2, ..., Pn;
              value
                obs_part_P1: nm → P1
                obs_part_P2: nm → P2
                ...,
                obs_part_Pn: nm → Pn;
              proof obligation
                ℒ; " ]
    || vps := vps ⊕ ([P1, P2, ..., Pn] \ αps)
    || αps := αps ⊕ [P1, P2, ..., Pn]
  end
pre: is_AbsCoPaTyp(td((valtyp(typval(td)))(val)))

```

ℒ is a predicate expressing the disjointness of part sorts P₁, P₂, ..., P_n

observe_concrete_type

See description prompt definition 2 on Page 21 and Formula Item 161b on Page 113.

value

```

observe_concrete_type: NmVAL → TypDef → Unit
observe_concrete_type(nm,val)(td) ≡
  let mkConCoPaTyp(⟦, P⟩) = td((valtyp(typval(td)))(val)) in
    τ := τ ⊕ [ " type T = P-set ; value obs_part_T: nm → T; " ]
    || vps := vps ⊕ ([P] \ αps)
    || αps := αps ⊕ [P]
  end
pre: is_ConCoPaTyp(td((valtyp(typval(td)))(val)))

```

observe_unique_identifier

See description prompt definition 5 on Page 25 and Formula Item 159a on Page 112.

value

```

observe_unique_identifier: P → TypDef → Unit
observe_unique_identifier(nm,val)(td) ≡
  τ := τ ⊕ [ " type PI ; value uid_PI: nm → PI ; axiom ℒ; " ]

```

ℒ is a predicate expression over unique identifiers.

observe_mereology

See description prompt definition 6 on Page 27 and Formula Item 156a on Page 112.

value

```

observe_mereology: NmVAL → TypDef → Unit
observe_mereology(nm,val)(td) ≡
  τ := τ ⊕ [ " type MT =  $\mathcal{M}$ (PI1,PI2,...,PIn) ;
            value obs_mereo_P: nm → MT ;
            axiom  $\mathcal{M}\mathcal{E}$ ; " ]
pre: has_mereology(nm,val)(td) 30

```

$\mathcal{M}(PI1,PI2,\dots,PI_n)$ is a type expression over unique part identifiers. $\mathcal{M}\mathcal{E}$ is a predicate expression over unique part identifiers.

observe_part_attributes

See description prompt definition 7 on Page 29 and Formula Item 157 on Page 112.

value

```

observe_part_attributes: NmVAL → TypDef → Unit
observe_part_attributes(nm,val)(td) ≡
  let {A1,A2,...,Aa} = dom s_attrs(s_qs(val)) in
  τ := τ ⊕ [ " type A1, A2, ..., Aa
            value attr_A1: nm→Ai
            attr_A2: nm→A1
            ...
            attr_Aa: nm→Ai
            proof obligation [Disjointness of Attribute Types]
             $\mathcal{A}$ ; " ]
end

```

\mathcal{A} is a predicate over attribute types A_1, A_2, \dots, A_a .

observe_part_material_sort

See description prompt definition 4 on Page 23 and Formula Item 160a on Page 113.

value

```

observe_part_material_sort: NmVAL → TypDef → Unit
observe_part_material_sort(nm,val)(td) ≡
  let M = s_pns(td((valtyp(typval(td)))(val))) in
  τ := τ ⊕ [ " type M ; value obs_mat_sort_M: nm→M " ]
  || vps := vps ⊕ ([M] \ αps)
  || αps := αps ⊕ [M]
end
pre: is_AtPaVAL(val) ∧ is_MNm(s_pns(td((valtyp(typval(td)))(val))))

```

³⁰ See analysis prompt definition 21 on Page 26

observe_component_sort

See description prompt definition 3 on Page 22 and Formula Item 160b on Page 113.

value

```

observe_component_sort: NmVAL → TypDef → Unit
observe_component_sort(nm,val)(td) ≡
  let K = s_omkn(td((valtyp(typval(td)))(val))) in
  τ := τ ⊕ [ " type K ; value obs-comps: nm → K-set; " ]
  || vps := vps ⊕ ([K] \ αps)
  || αps := αps ⊕ [K]
end
pre is_AtPaTyp(td((valtyp(typval(td)))(val))) ∧ has_components(nm,val)

```

observe_material_part_sort

See description prompt definition 4 on Page 23 and Formula Item 158c on Page 112.

value

```

observe_material_part_sort: NmVAL → TypDef → Unit
observe_material_part_sort(nm,val)(td) ≡
  let P = s_pns(td((valtyp(typval(td)))(val))) in
  τ := τ ⊕ [ " type P ; value obs_part_P: nm → P " ]
  || vps := vps ⊕ ([P] \ αps)
  || αps := αps ⊕ [P]
end
pre is_MaTyp(td((valtyp(typval(td)))(val))) ∧ is_PNm(s_pns(td((valtyp(typval(td)))(val))))

```

3.8.4 Discussion of The Prompt Model

The prompt model of this section is formulated so as to reflect a “wavering”, of the domain engineer, between syntactic and semantic reflections. The syntactic reflections are represented by the syntactic arguments of the sort names, nm, and the type definitions, td. The semantic reflections are represented by the semantic argument of values, val. When we, in the various prompt definitions, use the expression $td((valtyp(typval(td)))(val))$ we mean to model that the domain analyser cum describer reflects semantically: “viewing”, as it were, the endurant. We could, as well, have written $td(nm)$ — reflecting a syntactic reference to the (emerging) type model in the mind of the domain engineer.

3.9 Conclusion

It is time to summarise, conclude and look forward.

3.9.1 What Has Been Achieved ?

Chapter 1 proposes a set of domain analysis & description prompts. Sections 3.4. and 3.8. proposed an operational semantics for the process of selecting and applying prompts, respectively a more abstract meaning of these prompts, the latter based on some notions of an “image” of perceived abstract types of syntactic and of semantic structures of the perceived domain. These notions were discussed in Sects. 3.5. and 3.6. To the best of our knowledge this is the first time a reasonably precise notion of ‘method’ with a similarly reasonably precise notion of a calculi of tools has been backed up formal definitions.

3.9.2 Are the Models Valid ?

Are the formal descriptions of the process of selecting and applying the analysis & description prompts, Sect. 3.4., and the meaning of these prompts, Sect. 3.8, modeling this process and these meanings realistically ? To that we can only answer the following: The process model is definitely modeling plausible processes. We discuss interpretations of the analysis & description order that this process model imposes in Sect. 3.4.3. There might be other orders, but the ones suggested in Sect. 3.4 can be said to be “orderly” and reflects empirical observations. The model of the meaning of prompts, Sect. 3.8, is more of an hypothesis. This model refers to “images” that the domain engineer is claimed to have in her mind. It must necessarily be a valid model, perhaps one of several valid models. We have speculated, over many years, over the existence of other models. But this is the most reasonable to us.

3.9.3 Future Work

We have hinted at possible ‘laws of description prompts’ in Sect. 3.4.3. Whether the process and prompt models (Sects. 3.4 and 3.8) are sufficient to express, let alone prove such laws is an open question. If the models are sufficient, then they certainly are valid.

To Every Manifest Domain Mereology a CSP Expression

We¹ give an abstract model of parts and part-hood relations, of Stanisław Leśniewski's *mereology* [104].

4.1 Introduction

Mereology applies to software application domains such as the financial service industry, railway systems, road transport systems, health care, oil pipelines, secure [IT] systems, etcetera. We relate this model to axiom systems for mereology, showing satisfiability, and show that for every mereology there corresponds a class of Communicating Sequential Processes [148], that is: a λ -expression.

4.1.1 Mereology

The term 'mereology' is accredited to the Polish mathematician, philosopher and logician Stanisław Leśniewski (1886–1939). In this contribution we shall be concerned with only certain aspects of mereology, namely those that appear most immediately relevant to domain science (a relatively new part of current computer science). Our knowledge of 'mereology' has been through studying, amongst others, [104].

"Mereology (from the Greek $\mu\epsilon\rho\omicron\varsigma$ 'part') is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole"². In this contribution we restrict 'parts' to be those that, firstly, are spatially distinguishable, then, secondly, while "being based" on such spatially distinguishable parts, are conceptually related. We use the term 'part' in a more general sense than in [70]. The relation: "being based", shall be made clear in this chapter. Accordingly two parts, p_x and p_y , (of a same "whole") are either "adjacent", or are "embedded within", one within the other, as loosely indicated in Fig. 4.1 on the following page. 'Adjacent' parts are direct parts of a same third part, p_z , i.e., p_x and p_y are "embedded within" p_z ; or one (p_x) or the other (p_y) or both (p_x and p_y) are parts of a same third part, p'_z "embedded within" p_z ; etcetera; as loosely indicated in Fig. 4.2 on the next page, or one is "embedded within" the other — etc. as loosely indicated in Fig. 1.2 on Page 5. Parts, whether 'adjacent' or 'embedded within', can share properties. For adjacent parts this sharing seems, in the literature, to be diagrammatically expressed by letting the part rectangles "intersect". Usually properties are not spatial hence 'intersection' seems confusing. We refer to Fig. 4.3 on the next page. Instead of depicting parts sharing properties as in Fig. 4.3 on the following page[Left], where shaded, dashed rounded-edge rectangles stands for 'sharing', we shall (eventually) show parts sharing properties as in Fig. 4.3 on the next page[Right] where $\bullet\text{---}\bullet$ connections connect those parts.

¹ This paper is a complete rewrite of [59].

² Achille Varzi: Mereology, <http://plato.stanford.edu/entries/mereology/> 2009 and [104].

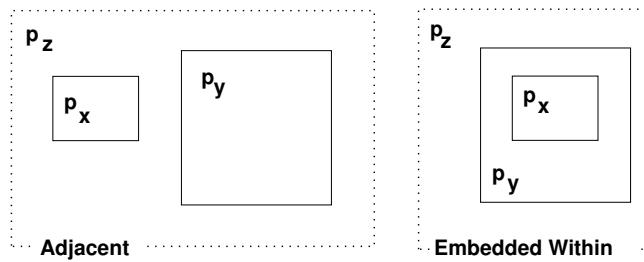


Fig. 4.1. Immediately 'Adjacent' and 'Embedded Within' Parts

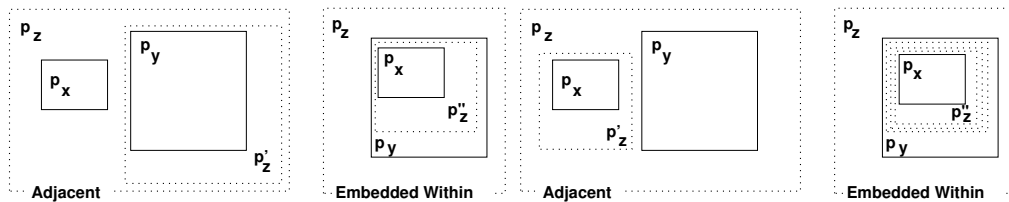


Fig. 4.2. Transitively 'Adjacent' and 'Embedded Within' Parts

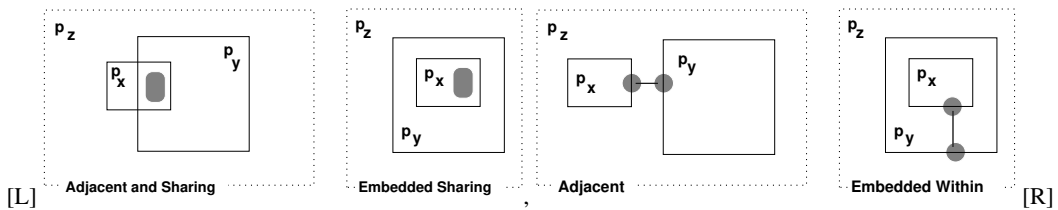


Fig. 4.3. Two models, [L,R], of parts sharing properties

4.1.2 From Domains via Requirements to Software

One reason for our interest in mereology is that we find that concept relevant to the modeling of domains. A derived reason is that we find the modeling of domains relevant to the development of software. Conventionally a first phase of software development is that of requirements engineering. To us domain engineering is (also) a prerequisite for requirements engineering, cf. Chapter 5. Thus to properly **design** Software we need to **understand** its or their **Requirements**; and to properly **prescribe** Requirements one must **understand** its **Domain**. To **argue** correctness of Software with respect to Requirements one must usually **make assumptions** about the **Domain**: $\mathbb{D}, \mathbb{S} \models \mathbb{R}$. Thus **description** of **Domains** become an indispensable part of **Software** development.

4.1.3 Domains: Science and Engineering

Domain Science is the study and knowledge of domains. **Domain Engineering** is the practice of “**walking the bridge**” from domain science to domain descriptions: to **create domain descriptions** on the background of scientific knowledge of domains, the specific domain “at hand”, or domains in general; and to **study domain descriptions** with a view to broaden and deepen scientific results about domain descriptions. This contribution is based on the engineering and study of many descriptions, of air traffic, banking, commerce (the consumer/retailer/wholesaler/producer supply chain), container lines, health care, logistics, pipelines, railway systems, secure [IT] systems, stock exchanges, etcetera.

4.1.4 Contributions of This Chapter

A general contribution of this chapter is that of providing elements of a domain science. Three specific contributions are those of (i) giving a model that satisfies published formal, axiomatic characterisations of mereology; (ii) showing that to every (such modeled) mereology there corresponds a CSP [148] program; and (iii) suggesting complementing **syntactic** and **semantic** theories of mereology.

4.1.5 Structure of Chapter

We briefly overview the structure of this contribution. First, in Sect. 4.2, **we loosely characterise how we look at mereologies: “what they are to us !”**. Then, in Sect. 4.3, **we give an abstract, model-oriented specification of a class of mereologies** in the form of composite parts and composite and atomic subparts and their possible connections. In preparation for Sect. 4.4 summarizes some of the part relations introduced by Leśniewski. The abstract model as well as the axiom system of Sect. 4.5 focuses on the **syntax of mereologies**. Following that, in Sect. 4.6, **we indicate how the model of Sect. 4.3 satisfies the axiom system of that Sect. 4.5**. In preparation for Sect. 4.7 we **present characterisations of attributes of parts, whether atomic or composite**. Finally Sect. 4.7 presents **a semantic model of mereologies**, one of a wide variety of such possible models. This one emphasizes the possibility of considering parts and subparts as processes and hence a mereology as a system of processes. Section 4.8 concludes with some remarks on what we have achieved.

4.2 Our Concept of Mereology

4.2.1 Informal Characterisation

Mereology, to us, is the study and knowledge about how physical and conceptual parts relate and what it means for a part to be related to another part: *being disjoint, being adjacent, being neighbours, being contained properly within, being properly overlapped with*, etcetera.

By physical parts we mean such spatial individuals which can be pointed to.

Examples: *a road net (consisting of street segments and street intersections); a street segment (between two intersections); a street intersection; a road (of sequentially neighbouring street segments of the same name); a vehicle; and a platoon (of sequentially neighbouring vehicles).*

By a conceptual part we mean an abstraction with no physical extent, which is either present or not.

Examples: *a bus timetable (not as a piece or booklet of paper, or as an electronic device, but) as an image in the minds of potential bus passengers; and routes of a pipeline, that is, neighbouring sequences of pipes, valves, pumps, forks and joins, for example referred to in discourse: “the gas flows through “such-and-such” a route”. The tricky thing here is that a route may be thought of as being both a concept or being a physical part — in which case one ought give them different names: a planned route and an actual road, for example.*

The mereological notion of subpart, that is: *contained within* can be illustrated by **examples:** *the intersections and street segments are subparts of the road net; vehicles are subparts of a platoon; and pipes, valves, pumps, forks and joins are subparts of pipelines.*

The mereological notion of adjacency can be illustrated by **examples.** We consider *the various controls of an air traffic system, cf. Fig. 4.4 on the following page, as well as its aircraft, as adjacent within the air traffic system; the pipes, valves, forks, joins and pumps of a pipeline, cf. Fig. 4.9 on Page 138, as adjacent within the pipeline system; two or more banks of a banking system, cf. Fig. 4.6 on Page 136, as being adjacent.*

The mereo-topological notion of neighbouring can be illustrated by **examples:** *Some adjacent pipes of a pipeline are neighbouring (connected) to other pipes or valves or pumps or forks or joins, etcetera; two immediately adjacent vehicles of a platoon are neighbouring.*

The mereological notion of proper overlap can be illustrated by **examples** some of which are of a general kind: *two routes of a pipelines may overlap; and two conceptual bus timetables may overlap with some, but not all bus line entries being the same; and some really reflect adjacency: two adjacent pipe overlap in their connection, a wall between two rooms overlap each of these rooms — that is, the rooms overlap each other “in the wall”.*

4.2.2 Six Examples

We shall, in Sect. 4.3, present a model that is claimed to abstract essential mereological properties of air traffic, buildings and their installations, machine assemblies, financial service industry, the oil industry and oil pipelines, and railway nets.

Air Traffic

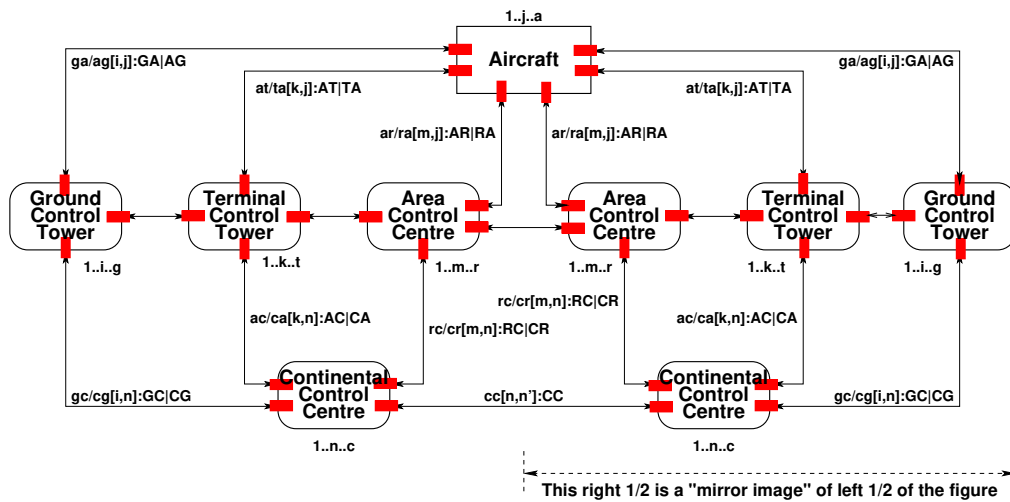


Fig. 4.4. A schematic air traffic system

Figure 4.4 shows nine adjacent (9) boxes and eighteen adjacent (18) lines. Boxes and lines are parts. The line parts “neighbours” the box parts they “connect”. Individually boxes and lines represent adjacent parts of the composite air traffic “whole”. The rounded corner boxes denote buildings. The sharp corner box denote aircraft. Lines denote radio telecommunication. The “overlap” between neighbouring line and box parts are indicated by “connectors”. Connectors are shown as small filled, narrow, either horizontal or vertical “filled” rectangle³ at both ends of the double-headed-arrows lines, overlapping both the line arrows and the boxes. The index ranges shown attached to, i.e., labeling each unit, shall indicate that there are a multiple of the “single” (thus representative) box or line unit shown. These index annotations are what makes the diagram of Fig. 4.4 schematic. Notice that the ‘box’ parts are fixed installations and that the double-headed arrows designate the ether where radio waves may propagate. We could, for example, assume that each such line is characterised by a combination of location and (possibly encrypted) radio communication frequency. That would allow us to consider all lines for not overlapping. And if they were overlapping, then that must have been a decision of the air traffic system.

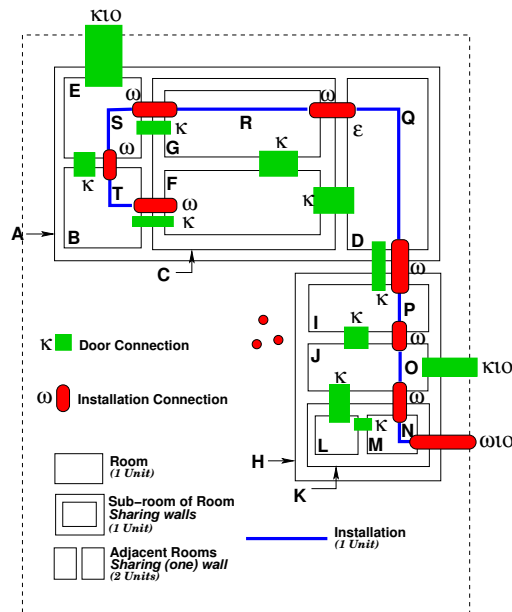


Fig. 4.5. A building plan with installation

Buildings

Figure 4.5 shows a building plan — as a composite part. The building consists of two buildings, A and H. The buildings A and H are neighbours, i.e., shares a common wall. Building A has rooms B, C, D and E, Building H has rooms I, J and K; Rooms L and M are within K. Rooms F and G are within C. The thick lines labeled N, O, P, Q, R, S, and T models either electric cabling, water supply, air conditioning, or some such “flow” of gases or liquids. Connection κ_{10} provides means of a connection between an environment, shown by dashed lines, and B or J, i.e. “models”, for example, a door. Connections κ provides “access” between neighbouring rooms. Note that ‘neighbouring’ is a transitive relation. Connection ω_{10} allows electricity (or water, or oil) to be conducted between an environment and a room. Connection ω allows electricity (or water, or oil) to be conducted through a wall. Etcetera. Thus “the whole” consists of A and H. Immediate subparts of A are B, C, D and E. Immediate subparts of C are G and F. Etcetera.

Financial Service Industry

Figure 4.6 on the next page is rather rough-sketchy! It shows seven (7) larger boxes [6 of which are shown by dashed lines], six [6] thin lined “distribution” boxes, and twelve (12) double-headed lines. Boxes and lines are parts. (We do not described what is meant by “distribution”.) Where double-headed lines touch upon (dashed) boxes we have connections. Six (6) of the boxes, the dashed line boxes, are composite parts, five (5) of them consisting of a variable number of atomic parts; five (5) are here shown as having three atomic parts each with bullets “between” them to designate “variability”. Clients, not shown, access the outermost (and hence the “innermost” boxes, but the latter is not shown) through connections, shown by bullets, ●.

Machine Assemblies

Figure 4.7 shows a machine assembly. Square boxes designate either composite or atomic parts. Black circles or ovals show connections. The full, i.e., the level 0, composite part consists of four immediate

³ There are 36 such rectangles in Fig. 4.4 on the facing page.

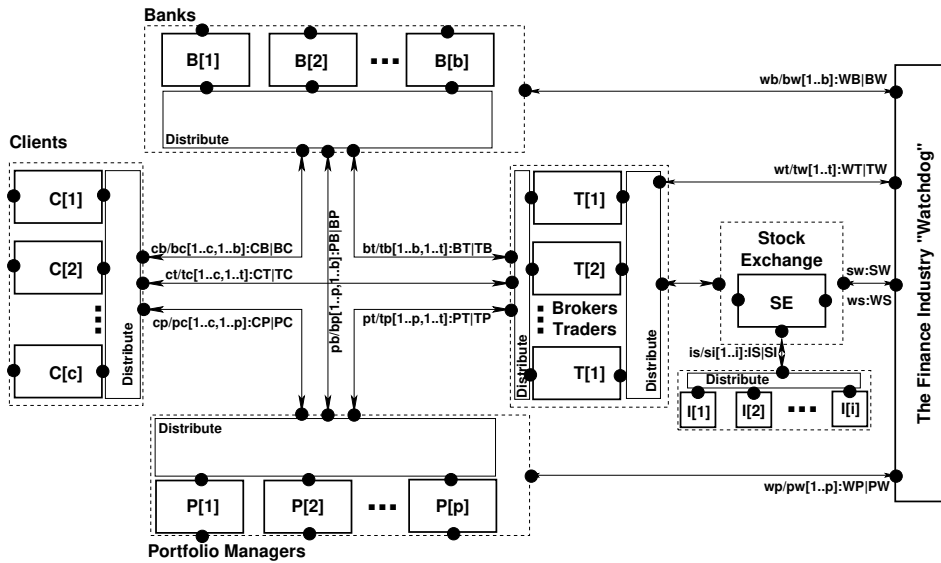


Fig. 4.6. A Financial Service Industry

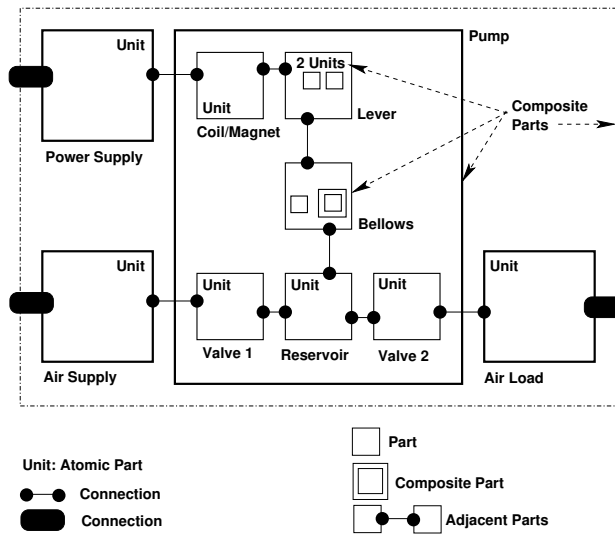


Fig. 4.7. An air pump, i.e., a physical mechanical system

parts and three internal and three external connections. The Pump is an assembly of six (6) immediate parts, five (5) internal connections and three (3) external connectors. Etcetera. Some connections afford “transmission” of electrical power. Other connections convey torque. Two connections convey input air, respectively output air.

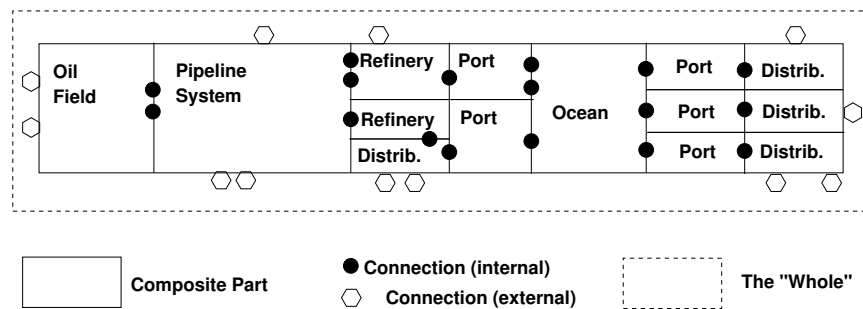


Fig. 4.8. A Schematic of an Oil Industry

Oil Industry

“The” Overall Assembly

Figure 4.8 shows a composite part consisting of fourteen (14) composite parts, left-to-right: one oil field, a crude oil pipeline system, two refineries and one, say, gasoline distribution network, two seaports, an ocean (with oil and ethanol tankers and their sea lanes), three (more) seaports, and three, say gasoline and ethanol distribution networks. Between all of the neighbouring composite parts there are connections, and from some of these composite parts there are connections (to an external environment). The crude oil pipeline system composite part will be concretised next.

A Concretised Composite Pipeline

Figure 4.9 on the following page shows a pipeline system. It consists of 32 atomic parts: fifteen (15) pipe units (shown as directed arrows and labeled $p1-p15$), four (4) input node units (shown as small circles, \circ , and labeled $ini-in\ell$), four (4) flow pump units (shown as small circles, \circ , and labeled $fpa-fpd$), five (5) valve units (shown as small circles, \circ , and labeled $vx-vw$), three (3) join units (shown as small circles, \circ , and labeled $jb-jc$), two (2) fork units (shown as small circles, \circ , and labeled $fb-fc$), one (1) combined join & fork unit (shown as small circles, \circ , and labeled $jafa$), and four (4) output node units (shown as small circles, \circ , and labeled $onp-ons$). In this example the routes through the pipeline system start with node units and end with node units, alternates between node units and pipe units, and are connected as shown by fully filled-out dark coloured disc connections. Input and output nodes have input, respectively output connections, one each, and shown as lighter coloured connections. In [56] we present a description of a class of abstracted pipeline systems.

Railway Nets

The left of Fig. 4.10 on the next page [L] diagrams four rail units, each with two, three or four connectors shown as narrow, somewhat “longish” rectangles. Multiple instances of these rail units can be assembled (i.e., composed) by their connectors as shown on Fig. 4.10 on the following page [L] into proper rail nets. The right of Fig. 4.10 on the next page [R] diagrams an example of a proper rail net. It is assembled from the kind of units shown in Fig. 4.10 [L]. In Fig. 4.10 [R] consider just the four dashed boxes: The dashed boxes are assembly units. Two designate stations, two designate lines (tracks) between stations. We refer to the caption four line text of Fig. 4.10 on the following page for more “statistics”. We could have chosen to show, instead, for each of the four “dangling” connectors, a composition of a connection, a special “end block” rail unit and a connector.

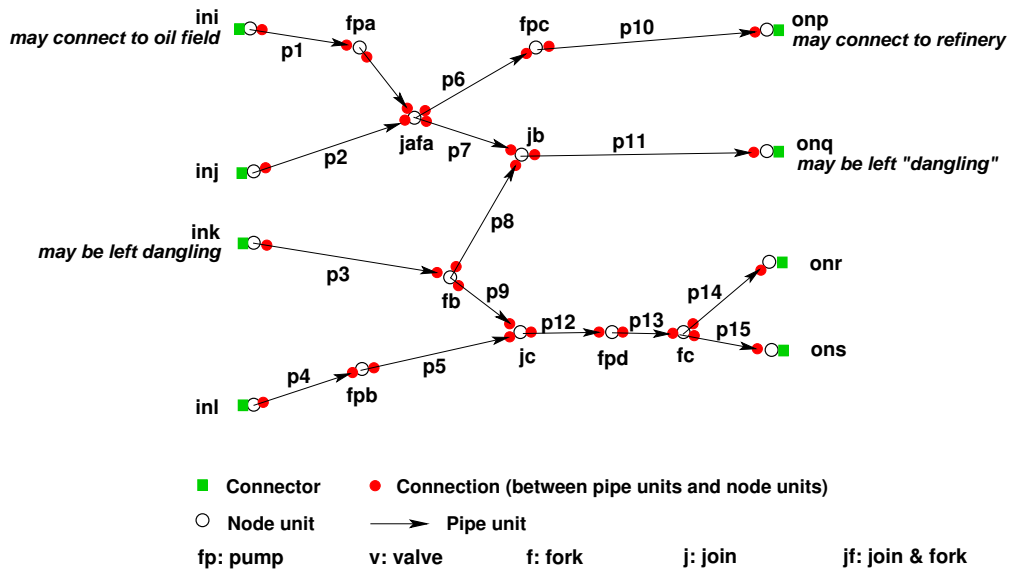


Fig. 4.9. A Pipeline System

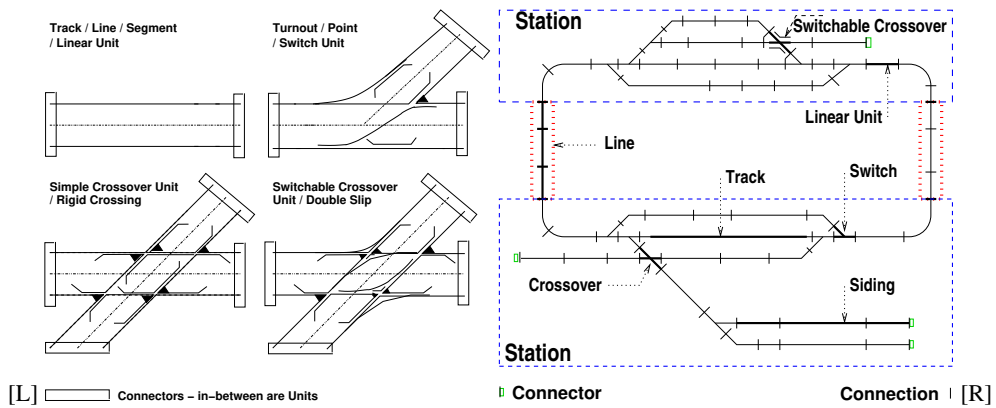


Fig. 4.10. To the left: Four rail units. To the right: A “model” railway net: An Assembly of four Assemblies: two stations and two lines. Lines here consist of linear rail units. Stations of all the kinds of units shown to the left. There are 66 connections and four “dangling” connectors

Discussion

We have brought these examples only to indicate the issues of a “whole” and atomic and composite parts, adjacency, within, neighbour and overlap relations, and the ideas of attributes and connections. We shall make the notion of ‘connection’ more precise in the next section.

4.3 An Abstract, Syntactic Model of Mereologies

4.3.1 Parts and Subparts

213 We distinguish between **atomic** and **composite parts**.

214 Atomic parts do not contain separately distinguishable parts.

215 Composite parts contain at least one separately distinguishable part.

type

213. $P ::= AP \mid CP^4$

214. $AP ::= mkAP(\dots)^5$

215. $CP ::= mkCP(\dots, s_sps:P\text{-set})^6$ **axiom** $\forall mkCP(_, ps):CP \cdot ps \neq \{\}$

It is the domain analyser who decides what constitutes “the whole”, that is, how parts relate to one another, what constitutes parts, and whether a part is atomic or composite. We refer to the proper parts of a composite part as subparts. Figure 4.11 illustrates composite and atomic parts. The *slanted sans serif* uppercase identifiers of Fig. 4.11 $A1, A2, A3, A4, A5, A6$ and $C1, C2, C3$ are meta-linguistic, that is, they stand for the parts they “decorate”; they are not identifiers of “our system”.

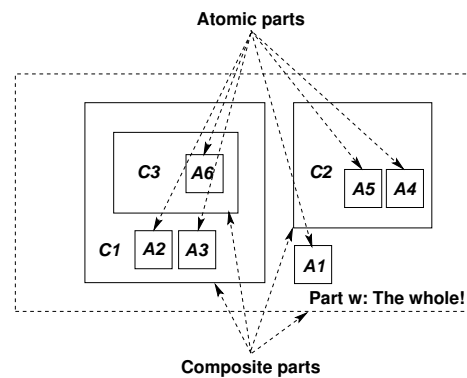


Fig. 4.11. Atomic and Composite Parts

4.3.2 No “Infinitely” Embedded Parts

The above syntax, Items 213–215, does not prevent composite parts, p , to contain composite parts, p' , “ad-infinitum”! But we do not wish such “recursively” contained parts!

216 To express the property that parts are finite we introduce a notion of *part derivation*.

217 The part derivation of an atomic part is the empty set.

218 The part derivation of a composite part, p , $mkC(\dots, ps)$ where \dots is left undefined, is the set ps of subparts of p .

⁴ In the RAISE [132] Specification Language, RSL [131], writing type definitions $X ::= Y|Z$ means that Y and Z are to be disjoint types. In Items 214.–215. the identifiers $mkAP$ and $mkCP$ are distinct, hence their types are disjoint.

⁵ $Y ::= mkY(\dots)$: y values (\dots) are marked with the “make constructor” mkY , cf. [177, 178].

⁶ In $Y ::= mkY(s_w:W, \dots)$ s_w is a “selector function” which when applied to an y , i.e., $s_w(y)$ identifies the W element, cf. [177, 178].

value

216. $\text{pt_der}: P \rightarrow \text{P-set}$
 217. $\text{pt_der}(\text{mkAP}(\dots)) \equiv \{\}$
 218. $\text{pt_der}(\text{mkCP}(\dots, \text{ps})) \equiv \text{ps}$

- 219 We can also express the part derivation, $\text{pt_der}(\text{ps})$ of a set, ps , of parts.
 220 If the set is empty then $\text{pt_der}(\{\})$ is the empty set, $\{\}$.
 221 Let $\text{mkA}(pq)$ be an element of ps , then $\text{pt_der}(\{\text{mkA}(pq)\} \cup \text{ps}')$ is ps' .
 222 Let $\text{mkC}(pq, \text{ps}')$ be an element of ps , then $\text{pt_der}(\text{ps}' \cup \text{ps})$ is ps' .

219. $\text{pt_der}: \text{P-set} \rightarrow \text{P-set}$
 220. $\text{pt_der}(\{\}) \equiv \{\}$
 221. $\text{pt_der}(\{\text{mkA}(\dots)\} \cup \text{ps}) \equiv \text{ps}$
 222. $\text{pt_der}(\{\text{mkC}(\dots, \text{ps}')\} \cup \text{ps}) \equiv \text{ps}' \cup \text{ps}$

- 223 Therefore, to express that a part is finite we postulate
 224 a natural number, n , such that a notion of iterated part set derivations lead to an empty set.
 225 An iterated part set derivation takes a set of parts and part set derive that set repeatedly, n times.
 226 If the result is an empty set, then part p was finite.

value

223. $\text{no_infinite_parts}: P \rightarrow \text{Bool}$
 224. $\text{no_infinite_parts}(p) \equiv$
 224. $\quad \exists n: \text{Nat} \cdot \text{it_pt_der}(\{p\})(n) = \{\}$
 225. $\text{it_pt_der}: \text{P-set} \rightarrow \text{Nat} \rightarrow \text{P-set}$
 226. $\text{it_pt_der}(\text{ps})(n) \equiv$
 226. $\quad \text{let } \text{ps}' = \text{pt_der}(\text{ps}) \text{ in}$
 226. $\quad \text{if } n=1 \text{ then } \text{ps}' \text{ else } \text{it_pt_der}(\text{ps}')(n-1) \text{ end end}$

4.3.3 Unique Identifications

Each physical part can be uniquely distinguished for example by an abstraction of its properties at a time of origin. In consequence we also endow conceptual parts with unique identifications.

- 227 In order to refer to specific parts we endow all parts, whether atomic or composite, with **unique identifications**.
 228 We postulate functions which observe these **unique identifications**, whether as parts in general or as atomic or composite parts in particular.
 229 such that any to parts which are distinct have **unique identifications**.

type

227. UI

value

228. $\text{uid_UI}: P \rightarrow \text{UI}$

axiom

229. $\forall p, p': P \cdot p \neq p' \Rightarrow \text{uid_UI}(p) \neq \text{uid_UI}(p')$

A model for uid_UI can be given. Presupposing subsequent material (on attributes and mereology) — “lumped” into part qualities, $pq: PQ$, we augment definitions of atomic and composite parts:

type
 214. AP :: mkA(s_pq:(s_uid:UI,...))
 215. CP :: mkC(s_pq:(s_uid:UI,...),s_sps:P-set)
value
 228. uid_UI(mkA((ui,...))) \equiv ui
 228. uid_UI(mkC((ui,...)),...) \equiv ui

Figure 4.12 illustrates the unique identifications of composite and atomic parts.

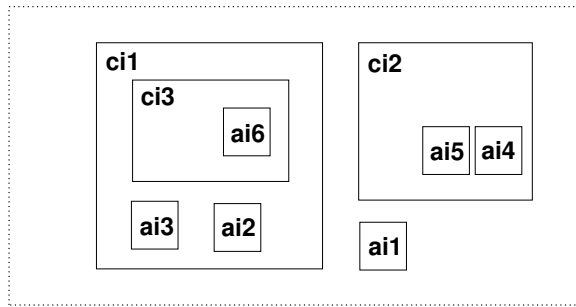


Fig. 4.12. ai_j : atomic part identifiers, ci_k : composite part identifiers

No two parts have the same unique identifier.

230 We define an auxiliary function, `no_pts_uis`, which applies to a[ny] part, p , and yields a pair: the number of subparts of the part argument, and the set of unique identifiers of parts within p .

231 `no_pts_uis` is defined in terms of yet an auxiliary function, `sum_no_pts_uis`.

value
 230. `no_pts_uis`: $P \rightarrow (\mathbf{Nat} \times \mathbf{UI-set}) \rightarrow (\mathbf{Nat} \times \mathbf{UI-set})$
 230. `no_pts_uis`(mkA(ui,...))(n,uis) \equiv (n+1,uis \cup {ui})
 230. `no_pts_uis`(mkC((ui,...),ps))(n,uis) \equiv
 230. **let** (n',uis') = `sum_no_pts_uis`(ps) **in**
 230. (n+n',uis \cup uis') **end**
 230. **pre**: `no_infinite_parts`(p)
 231. `sum_no_pts_uis`: $P\text{-set} \rightarrow (\mathbf{Nat} \times \mathbf{UI-set}) \rightarrow (\mathbf{Nat} \times \mathbf{UI-set})$
 231. `sum_no_pts_uis`(ps)(n,uis) \equiv
 231. **case** ps **of**
 231. {} \rightarrow (n,uis),
 231. {mkA(ui,...)}Ups' \rightarrow `sum_no_pts_uis`(ps')(n+1,uis \cup {ui}),
 231. {mkC((ui,...),ps')}Ups'' \rightarrow
 231. **let** (n'',uis'') = `sum_no_pts_uis`(ps')(1,{ui}) **in**
 231. `sum_no_pts_uis`(ps'')(n+n'',uis \cup uis'') **end**
 231. **end**
 231. **pre**: $\forall p:P \cdot p \in ps \Rightarrow \text{no_infinite_parts}(p)$

232 That no two parts have the same unique identifier can now be expressed by demanding that the number of parts equals the number of unique identifiers.

axiom
 232. $\forall p:P \cdot \text{let } (n,uis) = \text{no_pts_uis}(0,\{\}) \text{ in } n = \text{card } uis \text{ end}$

4.3.4 Attributes

Attribute Names and Values

- 233 Parts have sets of named attribute values, $\text{attrs}:\text{ATTRS}$.
 234 One can observe attributes from parts.
 235 Two distinct parts may share attributes:
 a For some (one or more) attribute name that is among the attribute names of both parts,
 b it is always the case that the corresponding attribute values are identical.

type

233. $\text{AN}_m, \text{AVAL}, \text{ATTRS} = \text{AN}_m \rightarrow_m \text{AVAL}$

value

234. $\text{attr_ATTRS}: P \rightarrow \text{ATTRS}$

235. $\text{share}: P \times P \rightarrow \mathbf{Bool}$

235. $\text{share}(p, p') \equiv$

235. $p \neq p' \wedge \sim \text{trans_adj}(p, p') \wedge$

235a. $\exists \text{anm}:\text{AN}_m \cdot \text{anm} \in \mathbf{dom} \text{attr_ATTRS}(p) \cap \mathbf{dom} \text{attr_ATTRS}(p') \Rightarrow$

235b. $\square (\text{attr_ATTRS}(p))(\text{anm}) = (\text{attr_ATTRS}(p'))(\text{anm})$

The function trans_adj is defined in Sect. 4.4.4 on Page 145.

Attribute Categories

One can suggest a hierarchy of part attribute categories: static or dynamic values — and within the dynamic value category: inert values or reactive values or active values — and within the dynamic active value category: autonomous values or biddable values or programmable values. By a **static attribute**, $a:A$, $\text{is_static_attribute}(a)$, we shall understand an attribute whose values are constants, i.e., cannot change. By a **dynamic attribute**, $a:A$, $\text{is_dynamic_attribute}(a)$, we shall understand an attribute whose values are variable, i.e., can change. By an **inert attribute**, $a:A$, $\text{is_inert_attribute}(a)$, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe properties of these new values. By a **reactive attribute**, $a:A$, $\text{is_reactive_attribute}(a)$, we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an **active attribute**, $a:A$, $\text{is_active_attribute}(a)$, we shall understand a dynamic attribute whose values change (also) of its own volition. By an **autonomous attribute**, $a:A$, $\text{is_autonomous_attribute}(a)$, we shall understand a dynamic active attribute whose values change value only “on their own volition”. The values of an autonomous attributes are a “law unto themselves and their surroundings”. By a **biddable attribute**, $a:A$, $\text{is_biddable_attribute}(a)$, (of a part) we shall understand a dynamic active attribute whose values are prescribed but may fail to be observed as such. By a **programmable attribute**, $a:A$, $\text{is_programmable_attribute}(a:A)$, we shall understand a dynamic active attribute whose values can be prescribed. By an **external attribute** we mean inert, reactive, active or autonomous attribute. By a **controllable attribute** we mean a biddable or programmable attribute. We define some auxiliary functions:

- 236 $\mathcal{S}_{\mathcal{A}}$ applies to $\text{attrs}:\text{ATTRS}$ and yields a grouping $(sa_1, sa_2, \dots, sa_{n_s})^7$, of **static** attribute values.
 237 $\mathcal{C}_{\mathcal{A}}$ applies to $\text{attrs}:\text{ATTRS}$ and yields a grouping $(ca_1, ca_2, \dots, ca_{n_c})^8$ of **controllable** attribute values.
 238 $\mathcal{E}_{\mathcal{A}}$ applies to $\text{attrs}:\text{ATTRS}$ and yields a set, $\{eA_1, eA_2, \dots, eA_{n_e}\}^9$ of **external** attribute names.

⁷ – where $\{sa_1, sa_2, \dots, sa_{n_s}\} \subseteq \mathbf{rng} \text{attrs}$

⁸ – where $\{ca_1, ca_2, \dots, ca_{n_c}\} \subseteq \mathbf{rng} \text{attrs}$

⁹ – where $\{eA_1, eA_2, \dots, eA_{n_e}\} \subseteq \mathbf{dom} \text{attrs}$

type 236. $\mathcal{I}_{sd}: \text{ATTRS} \rightarrow \text{SA}$
 SA, CA = AVAL* 237. $\mathcal{C}_{sd}: \text{ATTRS} \rightarrow \text{CA}$
 EA = ANm-st 238. $\mathcal{E}_{sd}: \text{ATTRS} \rightarrow \text{EA}$

value

The attribute names of static, controllable and external attributes do not overlap and together make up the attribute names of attrs.

4.3.5 Mereology

In order to illustrate other than the within and adjacency part relations we introduce the notion of mereology. Figure 4.13 illustrates a mereology between parts. A specific mereology-relation is, visually, a $\bullet\text{---}\bullet$ line that connects two distinct parts.

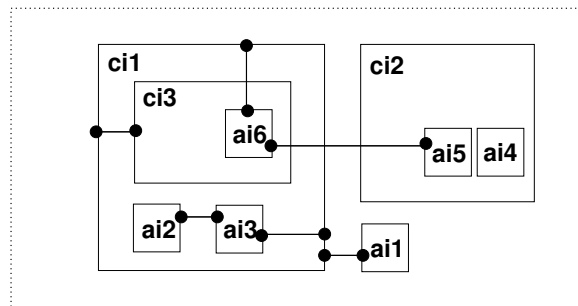


Fig. 4.13. Mereology: Relations between Parts

239 The mereology of a part is a set of unique identifiers of other parts.

type
 239. ME = UI-set

We may refer to the connectors by the two element sets of the unique identifiers of the parts they connect. For **example** with respect to Fig. 4.13:

- $\{ci_1, ci_3\}$,
- $\{ai_6, ci_1\}$,
- $\{ai_6, ai_5\}$ and
- $\{ai_2, ai_3\}$,
- $\{ai_3, ci_1\}$,
- $\{ai_1, ci_1\}$.

4.3.6 The Model

- 240 The “whole” is a part.
- 241 A part value has a part sort name and is either the value of an atomic part or of an abstract composite part.
- 242 An atomic part value has a part quality value.
- 243 An abstract composite part value has a part quality value and a set of at least of one or more part values.
- 244 A part quality value consists of a unique identifier, a mereology, and a set of one or more attribute named attribute values.

```

240 W = P
241 P = AP | CP
242 AP :: mkA(s_pq:PQ)
243 CP :: mkC(s_pq:PQ,s_ps:P-set)
244 PQ = UI × ME × (ANm  $\rightarrow$  AVAL)

```

We now assume that parts are not “recursively infinite”, and that all parts have unique identifiers

4.4 Some Part Relations

4.4.1 ‘Immediately Within’

245 One part, p , is said to be *immediately within*, $\text{imm_within}(p,p')$, another part, if p' is a composite part and p is observable in p' .

value

```

245. imm_within: P × P → Bool
245. imm_within(p,p') ≡
245.   case p' of
245.     (__,mkA(__,ps)) → p ∈ ps,
245.     (__,mkC(__,ps)) → p ∈ ps,
245.     _ → false
245.   end

```

4.4.2 ‘Transitive Within’

We can generalise the ‘immediate within’ property.

246 A part, p , is transitively within a part p' , $\text{trans_within}(p,p')$,
 a either if p , is immediately within p'
 b or
 c if there exists a (proper) composite part p'' of p' such that $\text{trans_within}(p'',p)$.

value

```

246. trans_wihin: P × P → Bool
246. trans_within(p,p') ≡
246a.   imm_within(p,p')
246b.   ∨
246c.   case p' of
246c.     (__,mkC(__,ps)) → p ∈ ps ∧
246c.       ∃ p'':P• p'' ∈ ps ∧ trans_within(p'',p),
246c.     _ → false
246.   end

```

4.4.3 ‘Adjacency’

247 Two parts, p,p' , are said to be *immediately adjacent*, $\text{imm_adj}(p,p')(c)$, to one another, in a composite part c , such that p and p' are distinct and observable in c .

value

247. $\text{imm_adj}: P \times P \rightarrow P \rightarrow \mathbf{Bool}$
 247. $\text{imm_adj}(p, p')(\text{mkA}(_, ps)) \equiv p \neq p' \wedge \{p, p'\} \subseteq ps$
 247. $\text{imm_adj}(p, p')(\text{mkC}(_, ps)) \equiv p \neq p' \wedge \{p, p'\} \subseteq ps$
 247. $\text{imm_adj}(p, p')(\text{mkA}(_)) \equiv \mathbf{false}$

4.4.4 Transitive ‘Adjacency’

We can generalise the immediate ‘adjacent’ property.

- 248 Two parts, p', p'' , of a composite part, p , are $\text{trans_adj}(p', p'')$ in p
 a either if $\text{imm_adj}(p', p'')(p)$,
 b or if there are two p''' and p'''' such that
 i p''' and p'''' are immediately adjacent parts of p and
 ii p is equal to p''' or p''' is properly within p and p' is equal to p'''' or p'''' is properly within p'

We leave the formalisation to the reader.

4.5 An Axiom System

Classical axiom systems for mereology focus on just one sort of “things”, namely \mathcal{P} arts. Leśniewski had in mind, when setting up his mereology to have it supplant set theory. So parts could be composite and consisting of other, the sub-parts — some of which would be atomic; just as sets could consist of elements which were sets — some of which would be empty.

4.5.1 Parts and Attributes

In our axiom system for mereology we shall avail ourselves of two sorts: \mathcal{P} arts, and \mathcal{A} ttributes.¹⁰

- **type** \mathcal{P}, \mathcal{A}

\mathcal{A} ttributes are associated with \mathcal{P} arts. We do not say very much about attributes: We think of attributes of parts to form possibly empty sets. So we postulate a primitive predicate, \in , relating \mathcal{P} arts and \mathcal{A} ttributes.

- $\in: \mathcal{A} \times \mathcal{P} \rightarrow \mathbf{Bool}$.

4.5.2 The Axioms

The axiom system to be developed in this section is a variant of that in [104]. We introduce the following relations between parts:

part_of:	$\mathbb{P}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
proper_part_of:	$\mathbb{PP}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
overlap:	$\mathbb{O}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
underlap:	$\mathbb{U}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
over_crossing:	$\mathbb{OX}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
under_crossing:	$\mathbb{UX}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
proper_overlap:	$\mathbb{PO}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146
proper_underlap:	$\mathbb{PU}: \mathcal{P} \times \mathcal{P} \rightarrow \mathbf{Bool}$	Page 146

¹⁰ Identifiers P and A stand for model-oriented types (parts and atomic parts), whereas identifiers \mathcal{P} and \mathcal{A} stand for property-oriented types (parts and attributes).

Let \mathbb{P} denote **part-hood**; p_x is part of p_y , is then expressed as $\mathbb{P}(p_x, p_y)$.¹¹ (4.1) Part p_x is part of itself (reflexivity). (4.2) If a part p_x is part p_y and, vice versa, part p_y is part of p_x , then $p_x = p_y$ (anti-symmetry). (4.3) If a part p_x is part of p_y and part p_y is part of p_z , then p_x is part of p_z (transitivity).

$$\forall p_x : \mathcal{S} \bullet \mathbb{P}(p_x, p_x) \quad (4.1)$$

$$\forall p_x, p_y : \mathcal{S} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_x)) \Rightarrow p_x = p_y \quad (4.2)$$

$$\forall p_x, p_y, p_z : \mathcal{S} \bullet (\mathbb{P}(p_x, p_y) \wedge \mathbb{P}(p_y, p_z)) \Rightarrow \mathbb{P}(p_x, p_z) \quad (4.3)$$

Let \mathbb{PP} denote **proper part-hood**. p_x is a proper part of p_y is then expressed as $\mathbb{PP}(p_x, p_y)$. \mathbb{PP} can be defined in terms of \mathbb{P} . $\mathbb{PP}(p_x, p_y)$ holds if p_x is part of p_y , but p_y is not part of p_x .

$$\mathbb{PP}(p_x, p_y) \triangleq \mathbb{P}(p_x, p_y) \wedge \neg \mathbb{P}(p_y, p_x) \quad (4.4)$$

Overlap, \mathbb{O} , expresses a relation between parts. Two parts are said to overlap if they have “something” in common. In classical mereology that ‘something’ is parts. To us parts are spatial entities and these cannot “overlap”. Instead they can ‘share’ attributes.

$$\mathbb{O}(p_x, p_y) \triangleq \exists a : \mathcal{A} \bullet a \in p_x \wedge a \in p_y \quad (4.5)$$

Underlap, \mathbb{U} , expresses a relation between parts. Two parts are said to underlap if there exists a part p_z of which p_x is a part and of which p_y is a part.

$$\mathbb{U}(p_x, p_y) \triangleq \exists p_z : \mathcal{S} \bullet \mathbb{P}(p_x, p_z) \wedge \mathbb{P}(p_y, p_z) \quad (4.6)$$

Think of the underlap p_z as an “umbrella” which both p_x and p_y are “under”.

Over-cross, \mathbb{OX} , p_x and p_y are said to over-cross if p_x and p_y overlap and p_x is not part of p_y .

$$\mathbb{OX}(p_x, p_y) \triangleq \mathbb{O}(p_x, p_y) \wedge \neg \mathbb{P}(p_x, p_y) \quad (4.7)$$

Under-cross, \mathbb{UX} , p_x and p_y are said to under cross if p_x and p_y underlap and p_y is not part of p_x .

$$\mathbb{UX}(p_x, p_y) \triangleq \mathbb{U}(p_x, p_y) \wedge \neg \mathbb{P}(p_y, p_x) \quad (4.8)$$

Proper Overlap, \mathbb{PO} , expresses a relation between parts. p_x and p_y are said to properly overlap if p_x and p_y over-cross and if p_y and p_x over-cross.

$$\mathbb{PO}(p_x, p_y) \triangleq \mathbb{OX}(p_x, p_y) \wedge \mathbb{OX}(p_y, p_x) \quad (4.9)$$

Proper Underlap, \mathbb{PU} , p_x and p_y are said to properly underlap if p_x and p_y under-cross and p_y and p_x under-cross.

$$\mathbb{PU}(p_x, p_y) \triangleq \mathbb{UX}(p_x, p_y) \wedge \mathbb{UX}(p_y, p_x) \quad (4.10)$$

4.6 Satisfaction

We shall sketch a proof that the *model* of Sect. 4.3, *satisfies*, i.e., is a model of, the *axioms* of Sect. 4.5.

4.6.1 Some Definitions

To that end we first define the notions of *interpretation*, *satisfiability*, *validity* and *model*. **Interpretation**: By an interpretation of a predicate we mean an assignment of a truth value to the predicate where the assignment may entail an assignment of values, in general, to the terms of the predicate. **Satisfiability**: By the satisfiability of a predicate we mean that the predicate is true for some interpretation. **Valid**: By the validity of a predicate we mean that the predicate is true for all interpretations. **Model**: By a model of a predicate we mean an interpretation for which the predicate holds.

¹¹ Our notation now is not RSL but a conventional first-order predicate logic notation.

4.6.2 A Proof Sketch

We assign

- 249 P as the meaning of \mathcal{P}
- 250 ATR as the meaning of \mathcal{A} ,
- 251 imm_within as the meaning of \mathbb{P} ,
- 252 trans_within as the meaning of \mathbb{PP} ,
- 253 \in : ATTR \times ATTRS-**set** \rightarrow **Bool** as the meaning of \in : $\mathcal{A} \times \mathcal{P} \rightarrow$ **Bool** and
- 254 sharing as the meaning of \mathbb{O} .

With the above assignments it is now easy to prove that the other axiom-operators U, PO, PU, OX and UX can be modeled by means of imm_within, within, ATTR \times ATTRS-**set** \rightarrow **Bool** and sharing.

4.7 A Semantic CSP Model of Mereology

The model of Sect. 4.3 can be said to be an abstract model-oriented definition of the syntax of mereology. Similarly the axiom system of Sect. 4.5 can be said to be an abstract property-oriented definition of the syntax of mereology. We show that to every mereology there corresponds a program of communicating sequential processes CSP. We assume that the reader has practical knowledge of Hoare's CSP [148].

4.7.1 Parts \simeq Processes

The model of mereology presented in Sect. 4.3 focused on (i) parts, (ii) unique identifiers and (iii) mereology. To parts we associate CSP processes. Part processes are indexed by the unique part identifiers. The mereology reveals the structure of CSP channels between CSP processes.

4.7.2 Channels

We define a general notion of a vector of channels. One vector element for each "pair" of distinct unique identifiers. Vector indices are set of two distinct unique identifiers.

- 255 Let w be the "whole" (i.e., a part).
- 256 Let uis be the set of all unique identifiers of the "whole".
- 257 Let M be the type of messages sent over channels.
- 258 Channels provide means for processes to synchronise and communicate.

value

- 255. w:P
- 256. uis = **let** ($_, uis'$)= $\text{no_prts_uis}(w)$ **in** uis' **end**

type

- 257. M

channel

- 258. $\{\text{ch}[\{ui, ui'\}]: M | ui, ui': U | ui \neq ui' \wedge \{ui, ui'\} \subseteq uis\}$

- 259 We also define channels for access to external attribute values.

Without loss of generality we do so for all possible parts and all possible attributes.

channel

- 259. $\{\text{ch}[ui, an]: \text{AVAL} | ui: U | ui \in uis, an: \text{ANm}\}$

4.7.3 Compilation

We now show how to compile “real-life, actual” parts into **RSL-Text**. That is, turning “semantics” into syntax !

value

$$\begin{aligned} \text{comp_P}: P &\rightarrow \mathbf{RSL\text{-}Text} \\ \text{comp_P}(\text{mkA}(ui,me,attrs)) &\equiv \text{“}\mathcal{M}_a(ui,me,attrs)\text{”} \\ \text{comp_P}(\text{mkC}((ui,me,attrs),\{p_1,p_2,\dots,p_n\})) &\equiv \\ &\text{“}\mathcal{M}_c(ui,me,attrs) \parallel \\ &\text{” comp_process}(p_1) \text{“}\parallel\text{” comp_process}(p_2) \text{“}\parallel\text{”} \dots \text{“}\parallel\text{” comp_process}(p_n) \end{aligned}$$

The so-called core process expressions \mathcal{M}_a and \mathcal{M}_c relate to atomic and composite parts. They are defined, schematically, below as just \mathcal{M} . The compilation expressions have two elements: (i) those embraced by double quotes: “...”, and (ii) those that invoke further compilations. The first texts, (i), shall be understood as **RSL-Texts**. The compilation invocations, (ii), as expending into **RSL-Texts**. We emphasize the distinction between ‘usages’ and ‘definitions’. The expressions between double quotes: “...” designate usages. We now show how some of these usages require “definitions”. These ‘definitions’ are not the result of ‘parts-to-processes’ compilations. They are shown here to indicate, to the domain engineers, what must be further described, beyond the ‘mere’ compilations.

value

$$\begin{aligned} \mathcal{M}: ui:UI \times me:ME \times attrs:ATTRS &\rightarrow ca:\mathcal{C}_{\mathcal{A}}(attrs) \rightarrow \mathbf{RSL\text{-}Text} \\ \mathcal{M}(ui,me,attrs)(ca) &\equiv \\ &\mathbf{let} (me',ca') = \mathcal{F}(ui,me,attrs)(ca) \mathbf{in} \mathcal{M}(ui,me',attrs)(ca') \mathbf{end} \\ \mathcal{F}: ui:UI \times me:ME \times attrs:ATTRS &\rightarrow ca:CA \rightarrow \\ &\mathbf{in_in_chs}(ui,attrs) \mathbf{in, out} \mathbf{in_out_chs}(ui,me) \rightarrow ME \times CA' \end{aligned}$$

Recall (Page 142) that $\mathcal{C}_{\mathcal{A}}(attrs)$ is a grouping, $(ca_1, ca_2, \dots, ca_{n_c})$, of controlled attribute values.

260 The $\mathbf{in_chs}$ function applies to a set of uniquely named attributes and yields some **RSL-Text**, in the form of **input** channel declarations, one for each external attribute.

260. $\mathbf{in_chs}: ui:UI \times attrs:ATTRS \rightarrow \mathbf{RSL\text{-}Text}$
 260. $\mathbf{in_chs}(ui,attrs) \equiv \mathbf{“in} \{ xch[ui,xa_i] \mid xa_i:ANm \cdot xa_i \in \mathcal{C}_{\mathcal{A}}(attrs) \} \mathbf{”}$

261 The $\mathbf{in_out_chs}$ function applies to a pair, a unique identifier and a mereology, and yields some **RSL-Text**, in the form of **input/output** channel declarations, one for each unique identifier in the mereology.

261. $\mathbf{in_out_chs}: ui:UI \times me:ME \rightarrow \mathbf{RSL\text{-}Text}$
 261. $\mathbf{in_out_chs}(ui,me) \equiv \mathbf{“in, out} \{ xch[ui,ui'] \mid ui:UI \cdot ui' \in me \} \mathbf{”}$

\mathcal{F} is an action: it returns a possibly updated mereology and possibly updated controlled attribute values. We present a rough sketch of \mathcal{F} . The \mathcal{F} action non-deterministically internal choice chooses between

- either [1,2,3,4]
 - ⊗ [1] accepting input from
 - ⊗ [4] a suitable (“offering”) part process,
 - ⊗ [2] optionally offering a reply;
 - ⊗ [3] leading to an updated state;
- or [3,4]
 - ⊗ [5] finding a suitable “order” (val)
 - ⊗ [8] to a suitable (“inquiring”) behaviour,
 - ⊗ [6] offering that value,
 - ⊗ [7] leading to an updated state;
- or [9] doing own work leading to a new state.

value

```

 $\mathcal{F}(ui, me, attrs)(ca) \equiv$ 
[1]   [] {let val=ch[{ui,ui'}]? in
[2]     (ch[{ui,ui'}]!in_reply(val,(ui,me,attrs))(ca)) ;
[3]     in_update(val,(ui,me,attrs))(ca) end
[4]   | ui':UI • ui' ∈ me}
[5]   [] [] {let val=await_reply(ui',me,attrs)(ca) in
[6]     ch[{ui,ui'}]!val ;
[7]     out_update(val,(ui,me,attrs))(ca) end
[8]   | ui':UI • ui' ∈ me}
[9]   [] (me,own_work(ui,attrs))(ca)

```

```

in_reply: VAL × (ui:UI × me:ME × attrs:ATTRS) → ca:CA →
          in in_chs(attrs) in, out in_out_chs(ui,me) → VAL
in_update: VAL × (ui:UI × me:ME × attrs:ATTRS) → ca:CA →
           in, out in_out_chs(ui,me) → ME × CA
await_reply: (ui:UI, me:ME) → ca:CA → in, out in_out_chs(ui,me:ME) → VAL
out_update: (VAL × (ui:UI × me:ME <> attrs:ATTRS)) → ca:CA →
           in, out in_out_chs(ui,me) → ME × CA
own_work: (ui:UI × attrs:ATTRS) → CA → in, out in_out_chs(ui,me) CA

```

The above definitions of channels and core functions \mathcal{M} and \mathcal{F} are not examples of what will be compiled but of what the domain engineer must, after careful analysis, “create”.

4.7.4 Discussion

General

A little more meaning has been added to the notions of parts and their mereology. The within and adjacent to relations between parts (composite and atomic) reflect a phenomenological world of geometry, and the mereological relation between parts reflect both physical and conceptual world understandings: physical world in that, for example, radio waves cross geometric “boundaries”, and conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric “boundaries”.

Specific

The notion of parts is far more general than that of Chapter 1. We have been able to treat Stanisław Leśniewski’s notion of mereology solely based on parts, that is, their semantic values, without introducing the notion of the syntax of parts. Our compilation functions are (thus) far more general than defined in Chapter 1.

4.8 Concluding Remarks

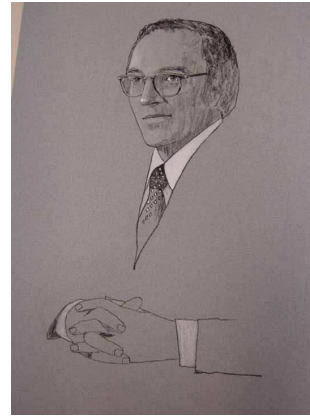
4.8.1 Relation to Other Work

The present contribution has been conceived in the following context.

My first awareness of the concept of ‘mereology’ was from listening to many presentations by **Douglas T. Ross** (1929–2007) at IFIP working group WG2.3 meetings over the years 1980–1999. In [228] Douglas T. Ross and John E. Ward report on the 1958–1967 MIT project for *computer-aided*

design (CAD) for numerically controlled production.¹² Pages 13–17 of [228] reflects on issues bordering to and behind the concerns of mereology. Ross’ thinking is clearly seen in the following text:

“... our consideration of fundamentals begins not with design or problem-solving or programming or even mathematics, but with philosophy (in the old-fashioned meaning of the word) – we begin by establishing a “world-view”. We have repeatedly emphasized that there is no way to bound or delimit the potential areas of application of our system, and that we must be prepared to cope with any conceivable problem. Whether the system will assist in any way in the solution of a given problem is quite another matter, . . . , but in order to have a firm and uniform foundation, we must have a uniform philosophical basis upon which to approach any given problem. This “world-view” must provide a working framework and methodology in terms of which any aspect of our awareness of the world may be viewed. It must be capable of expressing the utmost in reality, giving expression to unending layers of ever-finer and more concrete detail, but at the same time abstract chimerical¹³ visions bordering on unreality must fall within the same scheme. “Above all, the world-view itself must be concrete and workable, for it will form the basis for all involvement of the computer in the problem-solving process, as well as establishing a viewpoint for approaching the unknown human component of the problem-solving team.”



Douglas T. Ross 1927–2007.
Courtesy MIT Museum

Yes, indeed, the philosophical disciplines of ontology, epistemology and mereology, amongst others, ought be standard curricula items in the computer science and software engineering studies, or better: domain engineers cum software system designers ought be imbued by the wisdom of those disciplines as was Doug.

*“... in the summer of 1960 we coined the word plex to serve as a generic term for these philosophical ruminations. “Plex” derives from the word plexus, “An interwoven combination of parts in a structure”, (Webster). . . . The purpose of a ‘modeling plex’ is to represent completely and in its entirety a “thing”, whether it is concrete or abstract, physical or conceptual. A ‘modeling plex’ is a trinity with three primary aspects, all of which must be present. If any one is missing a complete representation or modeling is impossible. The three aspects of plex are **data, structure, and algorithm**. . . . ” which “. . . is concerned with the behavioral characteristics of the plex model – the interpretive rules for making meaningful the data and structural aspects of the plex, for assembling specific instances of the plex, and for interrelating the plex with other plexes and operators on plexes. Specification of the algorithmic aspect removes the ambiguity of meaning and interpretation of the data structure and provides a complete representation of the thing being modeled.”*

In the terminology of the current chapter a plex is a part (whether composite or atomic), the data are the properties (of that part), the structure is the mereology (of that part) and the algorithm is the process (for that part). Thus Ross was, perhaps, a first instigator (around 1960) of object-orientedness. A first, “top of the iceberg” account of the mereology-ideas that Doug had then can be found in the much later (1976) three page note [227]. Doug not only ‘invented’ CAD but was also the father of AED (Algol Extended for Design), the Automatically Programmed Tool (APT) language, SADT (Structured Analysis and Design

¹² Doug is said to have coined the term and the abbreviation CAD [226].

¹³ Chimerical: existing only as the product of unchecked imagination: fantastically visionary or improbable

Technique) and helped develop SADT into the IDEF0¹⁴ method for the Air Force’s Integrated Computer-Aided Manufacturing (ICAM) program’s IDEF suite of analysis and design methods. Douglas T. Ross went on for many years thereafter, to deepen and expand his ideas of relations between mereology and the programming language concept of type at the IFIP WG2.3 working group meetings. He did so in the, to some, enigmatic, but always fascinating style you find on Page 63 of [227].

In [169] **Henry S. Leonard** and **Henry Nelson Goodman**: *A Calculus of Individuals and Its Uses* present the American Pragmatist version of Leśniewski’s mereology. It is based on a single primitive: *discreet*. The idea of the calculus of individuals is, as in Leśniewski’s mereology, to avoid having to deal with the empty sets while relying on explicit reference to classes (or parts).

[104] **R. Casati** and **A. Varzi**: *Parts and Places: the structures of spatial representation* has been the major source for this chapter’s understanding of mereology. Our motivation was not the spatial or topological mereology, [236]. The present chapter does not utilize any of these concepts’ axiomatisation in [104, 236]. Still it is best to say that this chapter has benefited much from these publications.

Domain descriptions, besides mereological notions, also depend, in their successful form, on FCA: Formal Concept Analysis. Here a main inspiration has been drawn, since the mid 1990s, from **B. Ganter** and **R. Wille’s** *Formal Concept Analysis — Mathematical Foundations* [130].

The approach takes as input a matrix specifying a set of objects and the properties thereof, called attributes, and finds both all the “natural” clusters of attributes and all the “natural” clusters of objects in the input data, where a “natural” object cluster is the set of all objects that share a common subset of attributes, and a “natural” property cluster is the set of all attributes shared by one of the natural object clusters. Natural property clusters correspond one-for-one with natural object clusters, and a concept is a pair containing both a natural property cluster and its corresponding natural object cluster. The family of these concepts obeys the mathematical axioms defining a lattice, a Galois connection).

Thus the choice of adjacent and embedded (‘within’) parts and their connections is determined after serious formal concept analysis.

4.8.2 What Has Been Achieved ?

We have given a model-oriented specification of mereology. We have indicated that the model satisfies a widely known axiom system for mereology. We have suggested that (perhaps most) work on mereology amounts to syntactic studies. So we have suggested one of a large number of possible, schematic semantics of mereology. And we have shown that to every mereology there corresponds a set of communicating sequential process (CSP).

¹⁴ IDEF0: Icam DEfinition for Function Modeling: <https://en.wikipedia.org/wiki/IDEF0>

A Requirements Engineering Method

From Domain Descriptions to Requirements Prescriptions

Chapter 1 introduces a method for analysing and describing manifest domains. In this chapter we show how to systematically, but, of course, not automatically, “derive” requirements prescriptions from domain descriptions.

5.1 Introduction

We survey preliminary issues.

5.1.1 The Triptych Dogma of Software Development

We see software development progressing as follows: *Before one can design software one must have a firm grasp of the requirements. Before one can prescribe requirements one must have a reasonably firm grasp of the domain.* Software engineering, to us, therefore include these three phases: *domain engineering, requirements engineering and software design.*

5.1.2 Software As Mathematical Objects

Our base view is that *computer programs* are *mathematical objects*. That is, the text that makes up a computer program can be reasoned about. This view entails that computer program specifications can be reasoned about. And that the *requirements prescriptions* upon which these specifications are based can be reasoned about. This base view entails, therefore, that specifications, whether *software design specifications*, or *requirements prescriptions*, or *domain descriptions*, must [also] be *formal specifications*. This is in contrast to considering *software design specifications* being artifacts of sociological, or even of psychological “nature”.

5.1.3 The Contribution of Chapter

We claim that the present chapter contributes to our understanding and practice of *software engineering* as follows: (1) it shows how the new phase of engineering, domain engineering, as introduced in [70], forms a prerequisite for requirements engineering; (2) it endows the “classical” form of requirements engineering with a structured set of development stages and steps: (a) first a domain requirements stage, (b) to be followed by an interface requirements stages, and (c) to be concluded by a machine requirements stage; (3) it further structures and gives a reasonably precise contents to the stage of domain requirements: (i) first a projection step, (ii) then an instantiation step, (iii) then a determination step, (iv) then an extension step, and (v) finally a fitting step — with these five steps possibly being iterated; and (4) it also structures

and gives a reasonably precise contents to the stage of interface requirements based on a notion of shared entities, Each of the steps (i–v) open for the possibility of *simplifications*. Steps (a–c) and (i–v), we claim, are new. They reflect a serious contribution, we claim, to a logical structuring of the field of requirements engineering and its very many otherwise seemingly diverse concerns.

5.1.4 Some Comments

This chapter is, perhaps, unusual in the following respects: (i) It is a methodology chapter, hence there are no “neat” theories about development, no succinctly expressed propositions, lemmas nor theorems, and hence no proofs¹. (ii) As a consequence the chapter is borne by many, and by extensive examples. (iii) The examples of this chapter are all focused on a generic road transport net. (iv) To reasonably fully exemplify the requirements approach, illustrating how our method copes with a seeming complexity of interrelated method aspects, the full example of this chapter embodies very many description and prescription elements: hundreds of concepts (types, axioms, functions). (v) This methodology chapter covers a “grand” area of software engineering: Many textbooks and papers are written on *Requirements Engineering*. We postulate, in contrast to all such books (and papers), that *requirements engineering* should be founded on *domain engineering*. Hence we must, somehow, show that our approach relates to major elements of what the *Requirements Engineering* books put forward. (vi) As a result, this chapter is long.

5.1.5 Structure of Chapter

The structure of the chapter is as follows: Section 5.2 provides a fair-sized, hence realistic example. Sections 5.3–5.5 covers our approach to requirements development. Section 5.3 overviews the issue of ‘requirements’; relates our approach (i.e., Sects. 5.4–5.5) to *systems, user and external equipment* and *functional requirements*; and Sect. 5.3 also introduces the concepts of the *machine* to be requirements prescribed, the *domain*, the *interface* and the *machine requirements*. Section 5.4 covers the *domain requirements* stages of *projection* (Sect. 5.4.1), *instantiation* (Sect. 5.4.2), *determination* (Sect. 5.4.3), *extension* (Sect. 5.4.4) and *fitting* (Sect. 5.4.5). Section 5.5 covers key features of *interface requirements*: *shared phenomena* (Sect. 5.5.1), *shared endurants* (Sect. 5.5.1) and *shared actions, shared events and shared behaviours* (Sect. 5.5.1). Section 5.5.1 further introduces the notion of *derived requirements*. Section 5.7 concludes the chapter.

5.2 An Example Domain: Transport

In order to exemplify the various stages and steps of requirements development we first bring a domain description example.² The example follows the steps of an idealised domain description. First we describe the endurants, then we describe the perdurants. Endurant description initially focus on the composite and atomic parts. Then on their “internal” qualities: unique identifications, mereologies, and attributes. The descriptions alternate between enumerated, i.e., labeled narrative sentences and correspondingly “numbered” formalisations. The narrative labels cum formula numbers will be referred to, frequently in the various steps of domain requirements development.

5.2.1 Endurants

Since we have chosen a manifest domain, that is, a domain whose endurants can be pointed at, seen, touched, we shall follow the analysis & description process as outlined in [70] and formalised in [61]. That

¹ — where these proofs would be about the development theories. The example development of requirements do imply properties, but formulation and proof of these do not constitute new contributions — so are left out.

² The example of this section is that of the “running example” of Chapter 1.

is, we first identify, analyse and describe (manifest) parts, composite and atomic, abstract (Sect. 5.2.2) or concrete (Sect. 5.2.2). Then we identify, analyse and describe their unique identifiers (Sect. 5.2.2), mereologies (Sect. 5.2.2), and attributes (Sects. 5.2.2–5.2.2).

The example fragments will be presented in a small type-font.

5.2.2 Domain, Net, Fleet and Monitor

The root domain, Δ , is that of a composite traffic system (262a.) with a road net, (262b.) with a fleet of vehicles and (262c.) of whose individual position on the road net we can speak, that is, monitor.³

- 262 We analyse the traffic system into
- a a composite road net,
 - b a composite fleet (of vehicles), and
 - c an atomic monitor.

type

262 Δ
 262a N
 262b F
 262c M

value

262a **obs_part_N**: $\Delta \rightarrow N$
 262b **obs_part_F**: $\Delta \rightarrow F$
 262c **obs_part_M**: $\Delta \rightarrow M$

Applying `observe_endurant_sorts` [70, Sect. ddp:observe-endurant-sort] to a net, $n:N$, yields the following.

- 263 The road net consists of two composite parts,
- a an aggregation of hubs and
 - b an aggregation of links.

type

263a HA
 263b LA

value

263a **obs_part_HA**: $N \rightarrow HA$
 263b **obs_part_LA**: $N \rightarrow LA$

Hubs and Links

Applying `observe_part_type` [70, Sect. ddp:observe-part-type] to hub and link aggregates yields the following.

- 264 Hub aggregates are sets of hubs.
 265 Link aggregates are sets of links.
 266 Fleets are set of vehicles.

type

264 H, HS = **H-set**
 265 L, LS = **L-set**
 266 V, VS = **V-set**

³ The monitor can be thought of, i.e., conceptualised. It is not necessarily a physically manifest phenomenon.

value264 **obs_part_HS**: HA \rightarrow HS265 **obs_part_LS**: LA \rightarrow LS266 **obs_part_VS**: F \rightarrow VS

267 We introduce some auxiliary functions.

a links extracts the links of a network.

b hubs extracts the hubs of a network.

value267a links: $\Delta \rightarrow$ L-set267a links(δ) \equiv **obs_part_LS**(**obs_part_LA**(**obs_part_N**(δ)))267b hubs: $\Delta \rightarrow$ H-set267b hubs(δ) \equiv **obs_part_HS**(**obs_part_HA**(**obs_part_N**(δ)))**Unique Identifiers**

Applying `observe_unique_identifier` [70, Sect. ddp:observe-unique-identifier] to the observed parts yields the following.

268 Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all

a have unique identifiers

b such that all such are distinct, and

c with corresponding observers.

type

268a NI, HAI, LAI, HI, LI, FI, VI, MI

value268c **uid_NI**: N \rightarrow NI268c **uid_HAI**: HA \rightarrow HAI268c **uid_LAI**: LA \rightarrow LAI268c **uid_HI**: H \rightarrow HI268c **uid_LI**: L \rightarrow LI268c **uid_FI**: F \rightarrow FI268c **uid_VI**: V \rightarrow VI268c **uid_MI**: M \rightarrow MI**axiom**268b $NI \cap HAI = \emptyset$, $NI \cap LAI = \emptyset$, $NI \cap HI = \emptyset$, etc.

where axiom 268b. is expressed semi-formally, in mathematics. We introduce some auxiliary functions:

269 `xtr_lis` extracts all link identifiers of a traffic system.270 `xtr_his` extracts all hub identifiers of a traffic system.271 Given an appropriate link identifier and a net `get_link` ‘retrieves’ the designated link.272 Given an appropriate hub identifier and a net `get_hub` ‘retrieves’ the designated hub.**value**269 `xtr_lis`: $\Delta \rightarrow$ LI-set269 `xtr_lis`(δ) \equiv 269 **let** `ls` = links(δ) **in** {**uid_LI**(`l`) | `l` \in `ls`} **end**270 `xtr_his`: $\Delta \rightarrow$ HI-set270 `xtr_his`(δ) \equiv

```

270   let hs = hubs( $\delta$ ) in {uid_Hl(h)|h:H•k  $\in$  hs} end
271   get_link: Ll  $\rightarrow$   $\Delta \rightsquigarrow$  L
271   get_link(li)( $\delta$ )  $\equiv$ 
271     let ls = links( $\delta$ ) in
271     let l:L • l  $\in$  ls  $\wedge$  li=uid_Ll(l) in l end end
271     pre: li  $\in$  xtr_lis( $\delta$ )
272   get_hub: Hl  $\rightarrow$   $\Delta \rightsquigarrow$  H
272   get_hub(hi)( $\delta$ )  $\equiv$ 
272     let hs = hubs( $\delta$ ) in
272     let h:H • h  $\in$  hs  $\wedge$  hi=uid_Hl(h) in h end end
272     pre: hi  $\in$  xtr_his( $\delta$ )

```

Mereology

We cover the mereologies of all part sorts introduced so far. We decide that nets, hub aggregates, link aggregates and fleets have no mereologies of interest. Applying `observe_mereology` [70, Sect. ddp:observe_mereology] to hubs, links, vehicles and the monitor yields the following.

- 273 Hub mereologies reflect that they are connected to zero, one or more links.
- 274 Link mereologies reflect that they are connected to exactly two distinct hubs.
- 275 Vehicle mereologies reflect that they are connected to the monitor.
- 276 The monitor mereology reflects that it is connected to all vehicles.
- 277 For all hubs of any net it must be the case that their mereology designates links of that net.
- 278 For all links of any net it must be the case that their mereologies designates hubs of that net.
- 279 For all transport domains it must be the case that
 - a the mereology of vehicles of that system designates the monitor of that system, and that
 - b the mereology of the monitor of that system designates vehicles of that system.

```

value
273 obs_mereo_H: H  $\rightarrow$  Ll-set
274 obs_mereo_L: L  $\rightarrow$  Hl-set
axiom
274  $\forall$  l:L•card obs_mereo_L(l)=2
value
275 obs_mereo_V: V  $\rightarrow$  Ml
276 obs_mereo_M: M  $\rightarrow$  Vl-set
axiom
277  $\forall$   $\delta$ : $\Delta$ , hs:HS•hs=hubs( $\delta$ ), ls:LS•ls=links( $\delta$ ) •
277    $\forall$  h:H•h  $\in$  hs•obs_mereo_H(h) $\subseteq$ xtr_lis( $\delta$ )  $\wedge$ 
278    $\forall$  l:L•l  $\in$  ls•obs_mereo_L(l) $\subseteq$ xtr_his( $\delta$ )  $\wedge$ 
279a   let f:F•f=obs_part_F( $\delta$ )  $\Rightarrow$ 
279a     let m:M•m=obs_part_M( $\delta$ ),
279a     vs:VS•vs=obs_part_VS(f) in
279a      $\forall$  v:V•v  $\in$  vs $\Rightarrow$ uid_V(v)  $\in$  obs_mereo_M(m)
279b      $\wedge$  obs_mereo_M(m) = {uid_V(v)|v:V•v  $\in$  vs}
279b   end end

```

Attributes, I

We may not have shown all of the attributes mentioned below — so consider them informally introduced !

- **Hubs:** *locations*⁴ are considered static, *hub states* and *hub state spaces* are considered programmable;
- **Links:** *lengths* and *locations* are considered static, *link states* and *link state spaces* are considered programmable;
- **Vehicles:** *manufacturer name*, *engine type* (whether diesel, gasoline or electric) and *engine power* (kW/horse power) are considered static; *velocity* and *acceleration* may be considered reactive (i.e., a function of gas pedal position, etc.), *global position* (informed via a GNSS: Global Navigation Satellite System) and *local position* (calculated from a global position) are considered biddable

Applying `observe_attributes` [70, Sect. ddp:observe-attributes] to hubs, links, vehicles and the monitor yields the following.

First hubs.

280 Hubs

- a have geodetic locations, GeoH ,
- b have *hub states* which are sets of pairs of identifiers of links connected to the hub⁵,
- c and have *hub state spaces* which are sets of hub states⁶.

281 For every net,

- a link identifiers of a hub state must designate links of that net.
- b Every hub state of a net must be in the hub state space of that hub.

282 We introduce an auxiliary function: `xtr_lis` extracts all link identifiers of a hub state.

type

280a GeoH

280b $\text{H}\Sigma = (\text{LI} \times \text{LI})\text{-set}$

280c $\text{H}\Omega = \text{H}\Sigma\text{-set}$

value

280a `attr_GeoH`: $\text{H} \rightarrow \text{GeoH}$

280b `attr_HΣ`: $\text{H} \rightarrow \text{H}\Sigma$

280c `attr_HΩ`: $\text{H} \rightarrow \text{H}\Omega$

axiom

281 $\forall \delta:\Delta \cdot \text{let } \text{hs} = \text{hubs}(\delta) \text{ in}$

281 $\forall h:\text{H} \cdot h \in \text{hs} \cdot$

281a $\text{xtr_lis}(h) \subseteq \text{xtr_lis}(\delta)$

281b $\wedge \text{attr_}\Sigma(h) \in \text{attr_}\Omega(h)$

281 **end**

value

282 `xtr_lis`: $\text{H} \rightarrow \text{LI}\text{-set}$

282 $\text{xtr_lis}(h) \equiv \{li \mid li:\text{LI}, (li', li''):\text{LI} \times \text{LI} \cdot (li', li'') \in \text{attr_H}\Sigma(h) \wedge li \in \{li', li''\}\}$

Then links.

283 Links have lengths.

284 Links have geodetic location.

285 Links have states and state spaces:

⁴ By location we mean a geodetic position.

⁵ A hub state “signals” which input-to-output link connections are open for traffic.

⁶ A hub state space indicates which hub states a hub may attain over time.

- a States modeled here as pairs, (h', h'') , of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h'' .) A link state can thus have 0, 1, 2, 3 or 4 such pairs.
- b State spaces are the set of all the link states that a link may enjoy.

type

283 LEN
 284 GeoL
 285a $L\Sigma = (HI \times HI)\text{-set}$
 285b $L\Omega = L\Sigma\text{-set}$

value

283 **attr_LEN**: $L \rightarrow \text{LEN}$
 284 **attr_GeoL**: $L \rightarrow \text{GeoL}$
 285a **attr_LΣ**: $L \rightarrow L\Sigma$
 285b **attr_LΩ**: $L \rightarrow L\Omega$

axiom

285 $\forall n:N \cdot \text{let } ls = \text{xtr_links}(n), hs = \text{xtr_hubs}(n) \text{ in}$
 285 $\forall l:L \cdot l \in ls \Rightarrow$
 285a $\text{let } l\sigma = \text{attr_L}\Sigma(l) \text{ in}$
 285a $0 \leq \text{card } l\sigma \leq 4$
 285a $\wedge \forall (h', h''):(HI \times HI) \cdot (h', h'') \in l\sigma \Rightarrow \{h', h''\} = \text{obs_mereo_L}(l)$
 285b $\wedge \text{attr_L}\Sigma(l) \in \text{attr_L}\Omega(l)$
 285 **end end**

Then vehicles.

- 286 Every vehicle of a traffic system has a position which is either ‘on a link’ or ‘at a hub’.
- a An ‘on a link’ position has four elements: a unique link identifier which must designate a link of that traffic system and a pair of unique hub identifiers which must be those of the mereology of that link.
- b The ‘on a link’ position real is the fraction, thus properly between 0 (zero) and 1 (one) of the length from the first identified hub “down the link” to the second identifier hub.
- c An ‘at a hub’ position has three elements: a unique hub identifier and a pair of unique link identifiers — which must be in the hub state.

type

286 $VPos = \text{onL} \mid \text{atH}$
 286a $\text{onL} :: LI HI HI R$
 286b $R = \text{Real} \quad \text{axiom } \forall r:R \cdot 0 \leq r \leq 1$
 286c $\text{atH} :: HI LI LI$

value

286 **attr_VPos**: $V \rightarrow VPos$

axiom

286a $\forall n:N, \text{onL}(li, fhi, thi, r):VPos \cdot$
 286a $\exists l:L \cdot l \in \text{obs_part_LS}(\text{obs_part_N}(n)) \Rightarrow li = \text{uid_L}(l) \wedge \{fhi, thi\} = \text{obs_mereo_L}(l),$
 286c $\forall n:N, \text{atH}(hi, fli, tli):VPos \cdot$
 286c $\exists h:H \cdot h \in \text{obs_part_HS}(\text{obs_part_N}(n)) \Rightarrow hi = \text{uid_H}(h) \wedge (fli, tli) \in \text{attr_L}\Sigma(h)$

287 We introduce an auxiliary function `distribute`.

- a `distribute` takes a net and a set of vehicles and
- b generates a map from vehicles to distinct vehicle positions on the net.

- c We sketch a “formal” `distribute` function, but, for simplicity we omit the technical details that secures distinctness — and leave that to an axiom !
- 288 We define two auxiliary functions:
- a `xtr_links` extracts all links of a net and
 - b `xtr_hub` extracts all hubs of a net.

type

287b `MAP = VI \rightarrow_m VPos`

axiom

287b $\forall \text{map:MAP} \cdot \text{card dom map} = \text{card rng map}$

value

287 `distribute: VS \rightarrow N \rightarrow MAP`

287 `distribute(vs)(n) \equiv`

287a `let (hs,ls) = (xtr_hubs(n),xtr_links(n)) in`

287a `let vps = {onL(uid_(l),fhi,thi,r) | l:L·l \in ls \wedge {fhi,thi} \subseteq obs_mereo_L(l) \wedge 0 \leq r \leq 1}`

287a `U {atH(uid_H(h),fli,tli) | h:H·h \in hs \wedge {fli,tli} \subseteq obs_mereo_H(h)} in`

287b `[uid_V(v) \mapsto vp | v:V, vp:VPos·v \in vs \wedge vp \in vps] end`

287 `end`

288a `xtr_links: N \rightarrow L-set`

288a `xtr_links(n) \equiv obs_part_LS(obs_part_LA(n))`

288b `xtr_hubs: N \rightarrow H-set`

288a `xtr_hubs(n) \equiv obs_part_H(obs_part_HA $_{\Delta}$ (n))`

And finally monitors. We consider only one monitor attribute.

- 289 The monitor has a vehicle traffic attribute.
- a For every vehicle of the road transport system the vehicle traffic attribute records a possibly empty list of time marked vehicle positions.
 - b These vehicle positions are alternate sequences of ‘on link’ and ‘at hub’ positions
 - i such that any sub-sequence of ‘on link’ positions record the same link identifier, the same pair of ‘to’ and ‘from’ hub identifiers and increasing fractions,
 - ii such that any sub-segment of ‘at hub’ positions are identical,
 - iii such that vehicle transition from a link to a hub is commensurate with the link and hub mereologies, and
 - iv such that vehicle transition from a hub to a link is commensurate with the hub and link mereologies.

type

289 `Traffic = VI \rightarrow_m (T \times VPos)*`

value

289 `attr_Traffic: M \rightarrow Traffic`

axiom

289b $\forall \delta:\Delta \cdot$

289b `let m = obs_part_M(δ) in`

289b `let tf = attr_Traffic(m) in`

289b `dom tf \subseteq xtr_vis(δ) \wedge`

289b $\forall vi:VI \cdot vi \in \text{dom tf} \cdot$

289b `let tr = tf(vi) in`

289b $\forall i,i+1:\text{Nat} \cdot \{i,i+1\} \subseteq \text{dom tr} \cdot$

289b `let (t,vp)=tr(i),(t',vp')=tr(i+1) in`

289b $t < t'$

289(b)i $\wedge \text{case (vp,vp')} \text{ of}$

```

289(b)i      (onL(li,fhi,thi,r),onL(li',fhi',thi',r'))
289(b)i      → li=li'∧fhi=fhi'∧thi=thi'∧r≤r' ∧ li ∈ xtr_lis(δ) ∧ {fhi,thi} = obs_mereo_L(get_link(li)(δ)),
289(b)ii     (atH(hi,fli,tli),atH(hi',fli',tli'))
289(b)ii     → hi=hi'∧fli=fli'∧tli=tli' ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
289(b)iii    (onL(li,fhi,thi,1),atH(hi,fli,tli))
289(b)iii    → li=fli∧thi=hi ∧ {li,tli} ⊆ xtr_lis(δ) ∧ {fhi,thi} = obs_mereo_L(get_link(li)(δ))
289(b)iii    ∧ hi ∈ xtr_his(δ) ∧ (fli,tli) ∈ obs_mereo_H(get_hub(hi)(δ)),
289(b)iv     (atH(hi,fli,tli),onL(li',fhi',thi',0))
289(b)iv     → etcetera,
289b        _ → false
289b        end end end end end

```

5.2.3 Perdurants

Our presentation of example perdurants is not as systematic as that of example endurants. Give the simple basis of endurants covered above there is now a huge variety of perdurants, so we just select one example from each of the three classes of perdurants (as outline in [70]): a simple hub insertion *action* (Sect. 5.2.3), a simple link disappearance *event* (Sect. 5.2.3) and a not quite so simple *behaviour*, that of road traffic (Sect. 5.2.3).

Hub Insertion Action

- 290 Initially inserted hubs, h , are characterised
- a by their unique identifier which not one of any hub in the net, n , into which the hub is being inserted,
 - b by a mereology, $\{\}$, of zero link identifiers, and
 - c by — whatever — attributes, *attrs*, are needed.
- 291 The result of such a hub insertion is a net, n' ,
- a whose links are those of n , and
 - b whose hubs are those of n augmented with h .

value

```

290 insert_hub: H → N → N
291 insert_hub(h)(n) as n'
290a  pre: uid_H(h) ∉ xtr_his(n)
290b    ∧ obs_mereo_H= {}
290c    ∧ ...
291a  post: obs_part_Ls(n) = obs_part_Ls(n')
291b    ∧ obs_part_Hs(n) ∪ {h} = obs_part_Hs(n')

```

Link Disappearance Event

We formalise aspects of the link disappearance event:

- 292 The result net, $n':N'$, is not well-formed.
- 293 For a link to disappear there must be at least one link in the net;
- 294 and such a link may disappear such that
- 295 it together with the resulting net makes up for the “original” net.

value

```

292 link_diss_event: N × N' × Bool
292 link_diss_event(n,n') as tf
293   pre: obs_part_Ls(obs_part_LS(n)) ≠ {}
294   post: ∃ l:L·l ∈ obs_part_Ls(obs_part_LS(n)) ⇒
295         l ∉ obs_part_Ls(obs_part_LS(n'))
295         ∧ n' ∪ {l} = obs_part_Ls(obs_part_LS(n))

```

Road Traffic

The analysis & description of the road traffic behaviour is composed (i) from the description of the global values of nets, links and hubs, vehicles, monitor, a clock, and an initial distribution, *map*, of vehicles, “across” the net; (ii) from the description of channels between vehicles and the monitor; (iii) from the description of behaviour signatures, that is, those of the overall road traffic system, the vehicles, and the monitor; and (iv) from the description of the individual behaviours, that is, the overall road traffic system, *rts*, the individual vehicles, *veh*, and the monitor, *mon*.

Global Values:

There is given some globally observable parts.

```

296 besides the domain, δ:Δ,
297 a net, n:N,
298 a set of vehicles, vs:V-set,
299 a monitor, m:M, and
300 a clock, clock, behaviour.
301 From the net and vehicles we generate an initial distribution of positions of vehicles.

```

The $n:N$, $vs:V\text{-set}$ and $m:M$ are observable from any road traffic system domain δ .

value

```

296 δ:Δ
297 n:N = obs_part_N(δ),
297 ls:L-set=links(δ),hs:H-set=hubs(δ),
297 lis:LI-set=xtr_lis(δ),his:HI-set=xtr_his(δ)
298 va:VS=obs_part_VS(obs_part_F(δ)),
298 vs:Vs-set=obs_part_Vs(va),
298 vis:VI-set = {uid_VI(v)|v:V·v ∈ vs},
299 m:obs_part_M(δ),
299 mi=uid_MI(m),
299 ma:attributes(m)
300 clock: T → out {clk_ch[vi|vi:VI·vi ∈ vis]} Unit
301 vm:MAP·vpos_map = distribute(vs)(n);

```

Channels:

```

302 We additionally declare a set of vehicle-to-monitor-channels indexed
    a by the unique identifiers of vehicles
    b and the (single) monitor identifier.7
    and communicating vehicle positions.

```

⁷ Technically speaking: we could omit the monitor identifier.

channel

302 $\{v_m_ch[vi,mi] \mid vi:VI \cdot vi \in vis\}:VPos$

Behaviour Signatures:

- 303 The road traffic system behaviour, *rts*, takes no arguments (hence the first **Unit**)⁸; and “behaves”, that is, continues forever (hence the last **Unit**).
- 304 The vehicle behaviour
- a is indexed by the unique identifier, $uid_V(v):VI$,
 - b the vehicle mereology, in this case the single monitor identifier $mi:MI$,
 - c the vehicle attributes, **obs_attr** $bs(v)$
 - d and — factoring out one of the vehicle attributes — the current vehicle position.
 - e The vehicle behaviour offers communication to the monitor behaviour (on channel $vm_ch[vi]$); and behaves “forever”.
- 305 The monitor behaviour takes
- a the monitor identifier,
 - b the monitor mereology,
 - c the monitor attributes,
 - d and — factoring out one of the vehicle attributes — the discrete road traffic, $drtf:dRTF$, being repeatedly “updated” as the result of **input** communications from (all) vehicles;
 - e the behaviour otherwise behaves forever.

value

- 303 $rts: \mathbf{Unit} \rightarrow \mathbf{Unit}$
- 304 $veh_{vi:VI}: mi:MI \rightarrow vp:VPos \rightarrow \mathbf{out} \ vm_ch[vi,mi] \ \mathbf{Unit}$
- 305 $mon_{mi:MI}: vis:VI\text{-set} \rightarrow RTF \rightarrow \mathbf{in} \ \{v_m_ch[vi,mi] \mid vi:VI \cdot vi \in vis\}, clk_ch \ \mathbf{Unit}$

The Road Traffic System Behaviour:

- 306 Thus we shall consider our **road traffic system**, *rts*, as
- a the concurrent behaviour of a number of vehicles and, to “observe”, or, as we shall call it, to monitor their movements,
 - b the monitor behaviour.

value

- 306 $rts() =$
- 306a $\parallel \{veh_{uid_VI(v)}(mi)(vm(uid_VI(v))) \mid v:V \cdot v \in vs\}$
- 306b $\parallel mon_{mi}(vis)([vi \mapsto \langle \rangle \mid vi:VI \cdot vi \in vis])$

where, wrt, the monitor, we dispense with the mereology and the attribute state arguments and instead just have a monitor traffic argument which records the discrete road traffic, MAP, initially set to “empty” traces ($\langle \rangle$, of so far “no road traffic”!).

In order for the monitor behaviour to assess the vehicle positions these vehicles communicate their positions to the monitor via a vehicle to monitor channel. In order for the monitor to time-stamp these positions it must be able to “read” a clock.

- 307 We describe here an abstraction of the vehicle behaviour **at** a Hub (*hi*).
- a Either the vehicle remains at that hub informing the monitor of its position,
 - b or, internally non-deterministically,

⁸ The **Unit** designator is an RSL technicality.

- i moves onto a link, tli, whose “next” hub, identified by thi, is obtained from the mereology of the link identified by tli;
- ii informs the monitor, on channel $vm[vi,mi]$, that it is now at the very beginning (0) of the link identified by tli, whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning of that link,
- c or, again internally non-deterministically, the vehicle “disappears — off the radar” !

```

307 vehvi(mi)(vp:atH(hi,fli,tli)) ≡
307a   v_m_ch[vi,mi]!vp ; vehvi(mi)(vp)
307b   □
307(b)i   let {hi',thi}=obs_mereo_L(get_link(tli)(n)) in
307(b)i   assert: hi'=hi
307(b)ii  v_m_ch[vi,mi]!onL(tli,hi,thi,0) ;
307(b)ii  vehvi(mi)(onL(tli,hi,thi,0)) end
307c   □ stop

```

- 308 We describe here an abstraction of the vehicle behaviour **on** a Link (ii). Either
- a the vehicle remains at that link position informing the monitor of its position,
 - b or, internally non-deterministically, if the vehicle’s position on the link has not yet reached the hub,
 - i then the vehicle moves an arbitrary increment ℓ_ε (less than or equal to the distance to the hub) along the link informing the monitor of this, or
 - ii else,
 - 1 while obtaining a “next link” from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
 - 2 the vehicle informs the monitor that it is now at the hub identified by thi, whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
 - c or, internally non-deterministically, the vehicle “disappears — off the radar” !

```

308 vehvi(mi)(vp:onL(li,fhi,thi,r)) ≡
308a   v_m_ch[vi,mi]!vp ; vehvi(mi,va)(vp)
308b   □ if r + ℓε ≤ 1
308(b)i   then
308(b)i   v_m_ch[vi,mi]!onL(li,fhi,thi,r+ℓε) ;
308(b)i   vehvi(mi)(onL(li,fhi,thi,r+ℓε))
308(b)ii  else
308(b)ii1  let li':LI·li' ∈ obs_mereo_H(get_hub(thi)(n)) in
308(b)ii2  v_m_ch[vi,mi]!atH(li,thi,li');
308(b)ii2  vehvi(mi)(atH(li,thi,li')) end end
308c   □ stop

```

The Monitor Behaviour

- 309 The monitor behaviour evolves around
- a the monitor identifier,
 - b the monitor mereology,
 - c and the attributes, ma:ATTR
 - d — where we have factored out as a separate arguments — a table of traces of time-stamped vehicle positions,
 - e while accepting messages
 - i about time
 - ii and about vehicle positions
 - f and otherwise progressing “in[de]finitely”.

- 310 Either the monitor “does own work”
 311 or, internally non-deterministically accepts messages from vehicles.
- a A vehicle position message, vp , may arrive from the vehicle identified by vi .
 - b That message is appended to that vehicle’s movement trace – prefixed by time (obtained from the time channel),
 - c whereupon the monitor resumes its behaviour —
 - d where the communicating vehicles range over all identified vehicles.

```

309 monmi(vis)(trf) ≡
310     monmi(vis)(trf)
311     □
311a   □ {let tvp = (clk_ch?, v_m_ch[vi,mi]?) in
311b     let trf' = trf † [vi ↦ trf(vi) ^ <tvp>] in
311c     monmi(vis)(trf')
311d     end end | vi:V1 • vi ∈ vis}

```

We are about to complete a long, i.e., a 6.3 page example (!). We can now comment on the full example: The domain, $\delta : \Delta$ is a manifest part. The road net, $n : N$ is also a manifest part. The fleet, $f : F$, of vehicles, $vs : VS$, likewise, is a manifest part. But the monitor, $m : M$, is a concept. One does not have to think of it as a manifest “observer”. The vehicles are on — or off — the road (i.e., links and hubs). We know that from a few observations and generalise to all vehicles. They either move or stand still. We also, similarly, know that. Vehicles move. Yes, we know that. Based on all these repeated observations and generalisations we introduce the concept of vehicle traffic. Unless positioned high above a road net — and with good binoculars — a single person cannot really observe the traffic. There are simply too many links, hubs, vehicles, vehicle positions and times. Thus we conclude that, even in a richly manifest domain, we can also “speak of”, that is, describe concepts over manifest phenomena, including time !

5.2.4 Domain Facets

The example of this section, i.e., Sect. 5.2, focuses on the *domain facet* [45, 2008] of (i) *intrinsic*. It does not reflect the other *domain facets*: (ii) domain support technologies, (iii) domain rules, regulations & scripts, (iv) organisation & management, and (v) human behaviour. The requirements examples, i.e., the rest of this chapter, thus builds only on the *domain intrinsic*. This means that we shall not be able to cover principles, technique and tools for the prescription of such important requirements that handle failures of support technology or humans. We shall, however point out where we think such, for example, fault tolerance requirements prescriptions “fit in” and refer to relevant publications for their handling.

5.3 Requirements

This and the next three sections, Sects. 5.4.–5.5., are the main sections of this chapter. Section 5.4. is the most detailed and systematic section. It covers the *domain requirements* operations of *projection*, *instantiation*, *determination*, *extension* and, less detailed, *fitting*. Section 5.5. surveys the *interface requirements* issues of *shared phenomena*: *shared endurants*, *shared actions*, *shared events* and *shared behaviour*, and “completes” the exemplification of the detailed *domain extension* of our requirements into a *road pricing system*. Section 5.5. also covers the notion of *derived requirements*.

5.3.1 The Three Phases of Requirements Engineering

There are, as we see it, three kinds of design assumptions and requirements: (i) *domain requirements*, (ii) *interface requirements* and (iii) *machine requirements*. (i) **Domain requirements** are those requirements

which can be expressed solely using terms of the domain ■ (ii) **Interface requirements** are those requirements which can be expressed only using technical terms of both the domain and the machine ■ (iii) **Machine requirements** are those requirements which, in principle, can be expressed solely using terms of the machine ■

Definition 26 Verification Paradigm: Some preliminary designations: let \mathcal{D} designate the the domain description; let \mathcal{R} designate the requirements prescription, and let \mathcal{S} designate the system design. Now $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ shall be read: it must be verified that the \mathcal{S} system design satisfies the \mathcal{R} requirements prescription in the context of the \mathcal{D} domain description ■

The “in the context of \mathcal{D} ...” term means that proofs of \mathcal{S} software design correctness with respect to \mathcal{R} requirements will often have to refer to \mathcal{D} domain requirements assumptions. We refer to [136, Gunter, Jackson and Zave, 2000] for an analysis of a varieties of forms in which \models relate to variants of \mathcal{D} , \mathcal{R} and \mathcal{S} .

5.3.2 Order of Presentation of Requirements Prescriptions

The *domain requirements development* stage — as we shall see — can be sub-staged into: *projection, instantiation, determination, extension and fitting*. The *interface requirements development* stage — can be sub-staged into *shared: enduring, action, event and behaviour* developments, where “sharedness” pertains to phenomena shared between, i.e., “present” in, both the domain (concretely, manifestly) and the machine (abstractly, conceptually). These development stages need not be pursued in the order of the three stages and their sub-stages. We emphasize that one thing is the stages and steps of development, as for example these: projection, instantiation, determination, extension, fitting, shared durants, shared actions, shared events, shared behaviours, etcetera, another thing is the requirements prescription that results from these development stages and steps. The further software development, after and on the basis of the requirements prescription starts only when all stages and steps of the requirements prescription have been fully developed. The domain engineer is now free to rearrange the final prescription, irrespective of the order in which the various sections were developed, in such a way as to give a most pleasing, pedagogic and cohesive reading (i.e., presentation). From such a requirements prescription one can therefore not necessarily see in which order the various sections of the prescription were developed.

5.3.3 Design Requirements and Design Assumptions

A crucial distinction is between *design requirements* and *design assumptions*. The **design requirements** are those requirements for which the system designer **has to** implement hardware or software in order satisfy system user expectations ■ The **design assumptions** are those requirements for which the system designer **does not** have to implement hardware or software, but whose properties the designed hardware, respectively software relies on for proper functioning ■

Example 5.1. . Road Pricing System — Design Requirements: The design requirements for the road pricing calculator of this chapter are for the design (ii) of that part of the vehicle software which interfaces the GNSS receiver and the road pricing calculator (cf. Items 390–393), (iii) of that part of the toll-gate software which interfaces the toll-gate and the road pricing calculator (cf. Items 398–400) and (i) of the road pricing calculator (cf. Items 429–442) ■

Example 5.2. . Road Pricing System — Design Assumptions: The design assumptions for the road pricing calculator include: (i) that *vehicles* behave as prescribed in Items 389–393, (ii) that the GNSS regularly offers vehicles correct information as to their global position (cf. Item 390), (iii) that *toll-gates* behave as prescribed in Items 395–400, and (iv) that the *road net* is formed and well-formed as defined in Examples 5.7–5.9 ■

Example 5.3. . **Toll-Gate System — Design Requirements:** The design requirements for the toll-gate system of this chapter are for the design of software for the toll-gate and its interfaces to the road pricing system, i.e., Items 394–395 ■

Example 5.4. . **Toll-Gate System — Design Assumptions:** The design assumptions for the toll-gate system include (i) that the vehicles behave as per Items 389–393, and (ii) that the road pricing calculator behave as per Items 429–442 ■

5.3.4 Derived Requirements

In building up the domain, interface and machine requirements a number of machine concepts are introduced. These machine concepts enable the expression of additional requirements. It is these we refer to as derived requirements. Techniques and tools espoused in such classical publications as [116, 158, 264, 168, 254] can in those cases be used to advantage.

5.4 Domain Requirements

Domain requirements primarily express the assumptions that a design must rely upon in order that that design can be verified. Although domain requirements firstly express assumptions it appears that the software designer is well-advised in also implementing, as data structures and procedures, the endurants, respectively perdurants expressed in the domain requirements prescriptions. Whereas domain endurants are “real-life” phenomena they are now, in domain requirements prescriptions, abstract concepts (to be represented by a machine).

Definition 27 Domain Requirements Prescription: A **domain requirements prescription** is that subset of the requirements prescription whose technical terms are defined in a domain description ■

To determine a relevant subset all we need is collaboration with requirements, cum domain stake-holders. Experimental evidence, in the form of example developments of requirements prescriptions from domain descriptions, appears to show that one can formulate techniques for such developments around a few domain-description-to-requirements-prescription operations. We suggest these: *projection*, *instantiation*, *determination*, *extension* and *fitting*. In Sect. 5.3.2 we mentioned that the order in which one performs these domain-description-to-domain-requirements-prescription operations is not necessarily the order in which we have listed them here, but, with notable exceptions, one is well-served in starting out requirements development by following this order.

5.4.1 Domain Projection

Definition 28 Domain Projection: By a **domain projection** is meant a *subset of the domain description, one which projects out all those endurants: parts, materials and components, as well as perdurants: actions, events and behaviours that the stake-holders do not wish represented or relied upon by the machine* ■

The resulting document is a *partial domain requirements prescription*. In determining an appropriate subset the requirements engineer must secure that the final “projection prescription” is complete and consistent — that is, that there are no “dangling references”, i.e., that all entities and their internal properties that are referred to are all properly defined.

Domain Projection — Narrative

We now start on a series of examples that illustrate domain requirements development.

Example 5.5. . **Domain Requirements. Projection: A Narrative Sketch:** We require that the road pricing system shall [at most] relate to the following domain entities – and only to these⁹: the net, its links and hubs, and their properties (unique identifiers, mereologies and some attributes), the vehicles, as endurants, and the general vehicle behaviours, as perdurants. We treat projection together with a concept of *simplification*. The example simplifications are vehicle positions and, related to the simpler vehicle position, vehicle behaviours. To prescribe and formalise this we copy the domain description. From that domain description we remove all mention of the hub insertion action, the link disappearance event, and the monitor ■

As a result we obtain $\Delta_{\mathcal{D}}$, the projected version of the domain requirements prescription¹⁰.

Domain Projection — Formalisation

The requirements prescription hinges, crucially, not only on a systematic narrative of all the projected, instantiated, determinated, extended and fitted specifications, but also on their formalisation. In the formal domain projection example we, regrettably, omit the narrative texts. In bringing the formal texts we keep the item numbering from Sect. 5.2, where you can find the associated narrative texts.

Example 5.6. . **Domain Requirements — Projection: Main Sorts**

type

262 $\Delta_{\mathcal{D}}$

262a $N_{\mathcal{D}}$

262b $F_{\mathcal{D}}$

value

262a **obs_part_N** $_{\mathcal{D}}$: $\Delta_{\mathcal{D}} \rightarrow N_{\mathcal{D}}$

262b **obs_part_F** $_{\mathcal{D}}$: $\Delta_{\mathcal{D}} \rightarrow F_{\mathcal{D}}$

type

263a $HA_{\mathcal{D}}$

263b $LA_{\mathcal{D}}$

value

263a **obs_part_HA**: $N_{\mathcal{D}} \rightarrow HA$

263b **obs_part_LA**: $N_{\mathcal{D}} \rightarrow LA$

Concrete Types

type

264 $H_{\mathcal{D}}, HS_{\mathcal{D}} = H_{\mathcal{D}}\text{-set}$

265 $L_{\mathcal{D}}, LS_{\mathcal{D}} = L_{\mathcal{D}}\text{-set}$

266 $V_{\mathcal{D}}, VS_{\mathcal{D}} = V_{\mathcal{D}}\text{-set}$

value

264 **obs_part_HS** $_{\mathcal{D}}$: $HA_{\mathcal{D}} \rightarrow HS_{\mathcal{D}}$

265 **obs_part_LS** $_{\mathcal{D}}$: $LA_{\mathcal{D}} \rightarrow LS_{\mathcal{D}}$

266 **obs_part_VS** $_{\mathcal{D}}$: $F_{\mathcal{D}} \rightarrow VS_{\mathcal{D}}$

267a links: $\Delta_{\mathcal{D}} \rightarrow L\text{-set}$

267a links($\delta_{\mathcal{D}}$) \equiv **obs_part_LS** $_{\mathcal{D}}$ (**obs_part_LA** $_{\mathcal{D}}$ ($\delta_{\mathcal{D}}$))

267b hubs: $\Delta_{\mathcal{D}} \rightarrow H\text{-set}$

267b hubs($\delta_{\mathcal{D}}$) \equiv **obs_part_HS** $_{\mathcal{D}}$ (**obs_part_HA** $_{\mathcal{D}}$ ($\delta_{\mathcal{D}}$))

⁹ By ‘relate to ... these’ we mean that the required system does not rely on domain phenomena that have been “projected away”.

¹⁰ Restrictions of the net to the toll road nets, hinted at earlier, will follow in the next domain requirements steps.

Unique Identifiers**type**268a HI, LI, VI, MI **value**268c **uid_HI**: $H_{\mathcal{D}} \rightarrow HI$ 268c **uid_LI**: $L_{\mathcal{D}} \rightarrow LI$ 268c **uid_VI**: $V_{\mathcal{D}} \rightarrow VI$ 268c **uid_MI**: $M_{\mathcal{D}} \rightarrow MI$ **axiom**268b $HI \cap LI = \emptyset, HI \cap VI = \emptyset, HI \cap MI = \emptyset,$ 268b $LI \cap VI = \emptyset, LI \cap MI = \emptyset, VI \cap MI = \emptyset$ **Mereology****value**273 **obs_mereo_H** $_{\mathcal{D}}$: $H_{\mathcal{D}} \rightarrow LI\text{-set}$ 274 **obs_mereo_L** $_{\mathcal{D}}$: $L_{\mathcal{D}} \rightarrow HI\text{-set}$ 274 **axiom** $\forall l:L_{\mathcal{D}} \cdot \text{card obs_mereo_L}_{\mathcal{D}}(l)=2$ 275 **obs_mereo_V** $_{\mathcal{D}}$: $V_{\mathcal{D}} \rightarrow MI$ 276 **obs_mereo_M** $_{\mathcal{D}}$: $M_{\mathcal{D}} \rightarrow VI\text{-set}$ **axiom**277 $\forall \delta_{\mathcal{D}}:\Delta_{\mathcal{D}}, \text{hs}:HS \cdot \text{hs} = \text{hubs}(\delta), \text{ls}:LS \cdot \text{ls} = \text{links}(\delta_{\mathcal{D}}) \Rightarrow$ 277 $\forall h:H_{\mathcal{D}} \cdot h \in \text{hs} \Rightarrow \text{obs_mereo_H}_{\mathcal{D}}(h) \subseteq \text{xtr_his}(\delta_{\mathcal{D}}) \wedge$ 278 $\forall l:L_{\mathcal{D}} \cdot l \in \text{ls} \cdot \text{obs_mereo_L}_{\mathcal{D}}(l) \subseteq \text{xtr_lis}(\delta_{\mathcal{D}}) \wedge$ 279a **let** $f:F_{\mathcal{D}} \cdot f = \text{obs_part_F}_{\mathcal{D}}(\delta_{\mathcal{D}}) \Rightarrow \text{vs}:VS_{\mathcal{D}} \cdot \text{vs} = \text{obs_part_VS}_{\mathcal{D}}(f)$ **in**279a $\forall v:V_{\mathcal{D}} \cdot v \in \text{vs} \Rightarrow \text{uid_V}_{\mathcal{D}}(v) \in \text{obs_mereo_M}_{\mathcal{D}}(m)$ 279b $\wedge \text{obs_mereo_M}_{\mathcal{D}}(m) = \{\text{uid_V}_{\mathcal{D}}(v) \mid v:V_{\mathcal{D}} \in \text{vs}\}$ 279b **end****Attributes:** We project attributes of hubs, links and vehicles. First **hubs**:**type**280a GeoH 280b $H\Sigma_{\mathcal{D}} = (LI \times LI)\text{-set}$ 280c $H\Omega_{\mathcal{D}} = H\Sigma_{\mathcal{D}}\text{-set}$ **value**280b **attr_H** $\Sigma_{\mathcal{D}}$: $H_{\mathcal{D}} \rightarrow H\Sigma_{\mathcal{D}}$ 280c **attr_H** $\Omega_{\mathcal{D}}$: $H_{\mathcal{D}} \rightarrow H\Omega_{\mathcal{D}}$ **axiom**281 $\forall \delta_{\mathcal{D}}:\Delta_{\mathcal{D}},$ 281 **let** $\text{hs} = \text{hubs}(\delta_{\mathcal{D}})$ **in**281 $\forall h:H_{\mathcal{D}} \cdot h \in \text{hs} \cdot$ 281a $\text{xtr_lis}(h) \subseteq \text{xtr_lis}(\delta_{\mathcal{D}})$ 281b $\wedge \text{attr_}\Sigma_{\mathcal{D}}(h) \in \text{attr_}\Omega_{\mathcal{D}}(h)$ 281 **end**Then **links**:**type**284 GeoL 285a $L\Sigma_{\mathcal{D}} = (HI \times HI)\text{-set}$ 285b $L\Omega_{\mathcal{D}} = L\Sigma_{\mathcal{D}}\text{-set}$ **value**

284 **attr_GeoL**: $L \rightarrow \text{GeoL}$
 285a **attr_LΣ_∅**: $L_{\emptyset} \rightarrow L\Sigma_{\emptyset}$
 285b **attr_LΩ_∅**: $L_{\emptyset} \rightarrow L\Omega_{\emptyset}$

axiom

285a– 285b on Page 161.

Finally **vehicles**: For ‘road pricing’ we need vehicle positions. But, for “technical reasons”, we must abstain from the detailed description given in Items 286–286c¹¹ We therefore *simplify* vehicle positions.

312 A simplified vehicle position designates

- a either a link
- b or a hub,

type

312 $\text{SVPos} = \text{SonL} \mid \text{SatH}$

312a $\text{SonL} :: \text{LI}$

312b $\text{SatH} :: \text{HI}$

axiom

286a' $\forall n:N, \text{SonL}(li):\text{SVPos} \cdot \exists l:L \cdot l \in \text{obs_part_LS}(\text{obs_part_N}(n)) \Rightarrow li = \text{uid_L}(l)$

286c' $\forall n:N, \text{SatH}(hi):\text{SVPos} \cdot \exists h:H \cdot h \in \text{obs_part_HS}(\text{obs_part_N}(n)) \Rightarrow hi = \text{uid_H}(h)$

Global Values**value**

296 $\delta_{\emptyset} : \Delta_{\emptyset}$,

297 $n:N_{\emptyset} = \text{obs_part_N}_{\emptyset}(\delta_{\emptyset})$,

297 $ls:L_{\emptyset}\text{-set} = \text{links}(\delta_{\emptyset})$,

297 $hs:H_{\emptyset}\text{-set} = \text{hubs}(\delta_{\emptyset})$,

297 $lis:LI\text{-set} = \text{xtr_lis}(\delta_{\emptyset})$,

297 $his:HI\text{-set} = \text{xtr_his}(\delta_{\emptyset})$

Behaviour Signatures: We omit the monitor behaviour.

313 We leave the vehicle behaviours’ attribute argument undefined.

type

313 ATTR

value

303 $\text{trs}_{\emptyset} : \text{Unit} \rightarrow \text{Unit}$

304 $\text{veh}_{\emptyset} : \text{VI} \times \text{MI} \times \text{ATTR} \rightarrow \dots \text{Unit}$

The System Behaviour: We omit the monitor behaviour.

value

306a $\text{trs}_{\emptyset}() = \parallel \{ \text{veh}_{\emptyset}(\text{uid_VI}(v), \text{obs_mereo_V}(v), _) \mid v:V_{\emptyset} \cdot v \in \text{vs} \}$

The Vehicle Behaviour: Given the simplification of vehicle positions we *simplify* the vehicle behaviour given in Items 307–308

307' $\text{veh}_{vi}(mi)(vp:\text{SatH}(hi)) \equiv$

307a' $\quad v_m_ch[vi,mi]!\text{SatH}(hi) ; \text{veh}_{vi}(mi)(\text{SatH}(hi))$

307(b)i' $\quad \square \text{ let } li:L \cdot li \in \text{obs_mereo_H}(\text{get_hub}(hi)(n)) \text{ in}$

307(b)ii' $\quad \quad v_m_ch[vi,mi]!\text{SonL}(li) ; \text{veh}_{vi}(mi)(\text{SonL}(li)) \text{ end}$

307c' $\quad \square \text{ stop}$

¹¹ The ‘technical reasons’ are that we assume that the *GNSS* cannot provide us with direction of vehicle movement and therefore we cannot, using only the *GNSS* provide the details of ‘offset’ along a link (*onL*) nor the “from/to link” at a hub (*atH*).


```

308' vehvi(mi)(vp:SonL(li)) ≡
308a'   v_m_ch[vi,mi]!SonL(li) ; vehvi(mi)(SonL(li))
308(b)ii1'  [] let hi:Hl·hi ∈ obs_mereo_L(get_link(li)(n)) in
308(b)ii2'   v_m_ch[vi,mi]!SatH(hi) ; vehvi(mi)(atH(hi)) end
308c'      [] stop

```

We can simplify Items 307'–308c' further.

```

314 vehvi(mi)(vp) ≡
315   v_m_ch[vi,mi]!vp ; vehvi(mi)(vp)
316   [] case vp of
316     SatH(hi) →
317       let li:L·li ∈ obs_mereo_H(get_hub(hi)(n)) in
318       v_m_ch[vi,mi]!SonL(li) ; vehvi(mi)(SonL(li)) end,
316     SonL(li) →
319       let hi:Hl·hi ∈ obs_mereo_L(get_link(li)(n)) in
320       v_m_ch[vi,mi]!SatH(hi) ; vehvi(mi)(atH(hi)) end end
321   [] stop

```

314 This line coalesces Items 307' and 308'.

315 Coalescing Items 307a' and 308'.

316 Captures the distinct parameters of Items 307' and 308'.

317 Item 307(b)i'.

318 Item 307(b)ii'.

319 Item 308(b)ii1'.

320 Item 308(b)ii2'.

321 Coalescing Items 307c' and 308c'.

The above vehicle behaviour definition will be transformed (i.e., further “refined”) in Sect. 5.5.1’s Example 5.15; cf. Items 389–393 on Page 186 ■

Discussion

Domain projection can also be achieved by developing a “completely new” domain description — typically on the basis of one or more existing domain description(s) — where that “new” description now takes the rôle of being the project domain requirements.

5.4.2 Domain Instantiation

Definition 29 Domain Instantiation: By **domain instantiation** we mean a **refinement** of the *partial domain requirements prescription* (resulting from the projection step) in which the refinements aim at rendering the *endurants: parts, materials and components*, as well as the *perdurants: actions, events and behaviours* of the domain requirements prescription more concrete, more specific ■ Instantiations usually render these concepts less general.

Properties that hold of the projected domain shall also hold of the (therefrom) instantiated domain.

Refinement of endurants can be expressed (i) either in the form of concrete types, (ii) or of further “delineating” axioms over sorts, (iii) or of a combination of concretisation and axioms. We shall exemplify the third possibility. Example 5.7 express requirements that the road net (on which the road-pricing system is to be based) must satisfy. Refinement of perdurants will not be illustrated (other than the simplification of the *vehicle* projected behaviour).

Domain Instantiation

Example 5.7. . **Domain Requirements. Instantiation Road Net:** We now require that there is, as before, a road net, $n_{\mathcal{G}}:N_{\mathcal{G}}$, which can be understood as consisting of two, “connected sub-nets”. A toll-road net, $trn_{\mathcal{G}}:TRN_{\mathcal{G}}$, cf. Fig. 5.1 on the next page, and an ordinary road net, $n_{\mathcal{O}}$. The two are connected as follows: The toll-road net, $trn_{\mathcal{G}}$, borders some toll-road plazas, in Fig. 5.1 on the following page shown by white filled circles (i.e., hubs). These toll-road plaza hubs are proper hubs of the ‘ordinary’ road net, $n'_{\mathcal{O}}$.

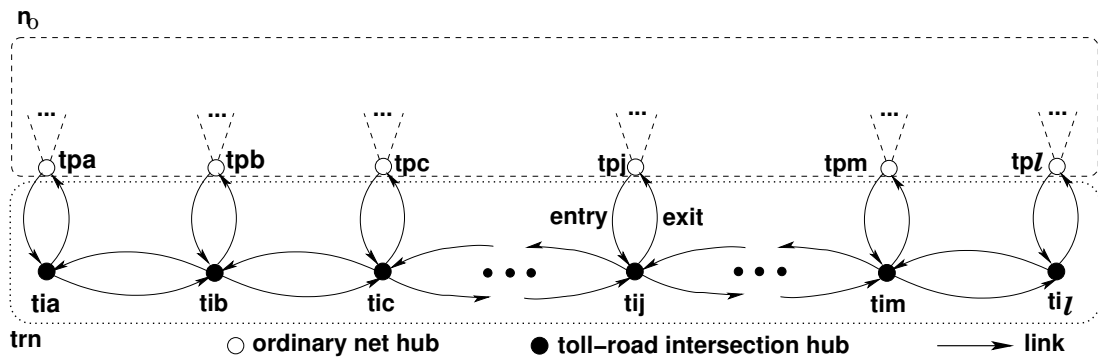


Fig. 5.1. A simple, linear toll-road net trn . tp_j : toll plaza j , ti_j : toll road intersection j .
 Upper dashed sub-figure hint at an ordinary road net n_o .
 Lower dotted sub-figure hint at a toll-road net trn .
 Dash-dotted (---) "V"-images above tp_j s hint at links to remaining “parts” of n_o .

- 322 The instantiated domain, $\delta_{\mathcal{G}}:\Delta_{\mathcal{G}}$ has just the net, $n_{\mathcal{G}}:N_{\mathcal{G}}$ being instantiated.
- 323 The road net consists of two “sub-nets”
 - a an “ordinary” road net, $n_o:N_{\mathcal{O}}$ and
 - b a toll-road net proper, $trn:TRN_{\mathcal{G}}$ —
 - c “connected” by an interface $hil:HIL$:
 - i That interface consists of a number of toll-road plazas (i.e., hubs), modeled as a list of hub identifiers, $hil:HI^*$.
 - ii The toll-road plaza interface to the toll-road net, $trn:TRN_{\mathcal{G}}$ ¹², has each plaza, $hil[i]$, connected to a pair of toll-road links: an entry and an exit link: $(l_e:L, l_x:L)$.
 - iii The toll-road plaza interface to the ‘ordinary’ net, $n_o:N_{\mathcal{O}}$, has each plaza, i.e., the hub designated by the hub identifier $hil[i]$, connected to one or more ordinary net links, $\{l_{i_1}, l_{i_2}, \dots, l_{i_k}\}$.
- 323b The toll-road net, $trn:TRN_{\mathcal{G}}$, consists of three collections (modeled as lists) of links and hubs:
 - i a list of pairs of toll-road entry/exit links: $\langle (l_{e_1}, l_{x_1}), \dots, (l_{e_\ell}, l_{x_\ell}) \rangle$,
 - ii a list of toll-road intersection hubs: $\langle h_{i_1}, h_{i_2}, \dots, h_{i_\ell} \rangle$, and
 - iii a list of pairs of main toll-road (“up” and “down”) links: $\langle (ml_{i_{1u}}, ml_{i_{1d}}), (m_{i_{2u}}, m_{i_{2d}}), \dots, (m_{i_{\ell u}}, m_{i_{\ell d}}) \rangle$.
- d The three lists have commensurate lengths (ℓ).

ℓ is the number of toll plazas, hence also the number of toll-road intersection hubs and therefore a number one larger than the number of pairs of main toll-road (“up” and “down”) links

¹² We (sometimes) omit the subscript \mathcal{G} when it should be clear from the context what we mean.

type

```

322  $\Delta_{\mathcal{G}}$ 
323  $N_{\mathcal{G}} = N_{\mathcal{G}'}$   $\times$  HIL  $\times$  TRN
323a  $N_{\mathcal{G}'}$ 
323b  $TRN_{\mathcal{G}} = (L \times L)^* \times H^* \times (L \times L)^*$ 
323c HIL =  $H^*$ 

```

axiom

```

323d  $\forall n_{\mathcal{G}}:N_{\mathcal{G}} \bullet$ 
323d   let  $(n_{\Delta}, hil, (exll, hl, lll)) = n_{\mathcal{G}}$  in
323d   len  $hil = \mathbf{len}$   $exll = \mathbf{len}$   $hl = \mathbf{len}$   $lll + 1$ 
323d   end

```

We have named the “ordinary” net sort (primed) $N_{\mathcal{G}'}$. It is “almost” like (unprimed) $N_{\mathcal{G}}$ — except that the interface hubs are also connected to the toll-road net entry and exit links.

The partial concretisation of the net sorts, $N_{\mathcal{G}'}$, into $N_{\mathcal{G}}$ requires some additional well-formedness conditions to be satisfied.

324 The toll-road intersection hubs all¹³ have distinct identifiers.

```

324  $wf\_dist\_toll\_road\_isect\_hub\_ids: H^* \rightarrow \mathbf{Bool}$ 
324  $wf\_dist\_toll\_road\_isect\_hub\_ids(hl) \equiv \mathbf{len}$   $hl = \mathbf{card}$   $xtr\_his(hl)$ 

```

325 The toll-road links all have distinct identifiers.

```

325  $wf\_dist\_toll\_road\_u\_d\_link\_ids: (L \times L)^* \rightarrow \mathbf{Bool}$ 
325  $wf\_dist\_toll\_road\_u\_d\_link\_ids(lll) \equiv 2 \times \mathbf{len}$   $lll = \mathbf{card}$   $xtr\_lis(lll)$ 

```

326 The toll-road entry/exit links all have distinct identifiers.

```

326  $wf\_dist\_e\_x\_link\_ids: (L \times L)^* \rightarrow \mathbf{Bool}$ 
326  $wf\_dist\_e\_x\_link\_ids(exll) \equiv 2 \times \mathbf{len}$   $exll = \mathbf{card}$   $xtr\_lis(exll)$ 

```

327 Proper net links must not designate toll-road intersection hubs.

```

327  $wf\_isoltd\_toll\_road\_isect\_hubs: H^* \times H^* \rightarrow N_{\mathcal{G}} \rightarrow \mathbf{Bool}$ 
327  $wf\_isoltd\_toll\_road\_isect\_hubs(hil, hl)(n_{\mathcal{G}}) \equiv$ 
327   let  $ls = xtr\_links(n_{\mathcal{G}})$  in
327   let  $his = \cup \{ \mathbf{obs\_mereo\_L}(l) \mid l \in ls \}$  in
327    $his \cap xtr\_his(hl) = \{ \}$  end end

```

328 The plaza hub identifiers must designate hubs of the ‘ordinary’ net.

```

328  $wf\_p\_hubs\_pt\_of\_ord\_net: H^* \rightarrow N'_{\Delta} \rightarrow \mathbf{Bool}$ 
328  $wf\_p\_hubs\_pt\_of\_ord\_net(hil)(n'_{\Delta}) \equiv \mathbf{elems}$   $hil \subseteq xtr\_his(n'_{\Delta})$ 

```

329 The plaza hub mereologies must each,
 a besides identifying at least one hub of the ordinary net,
 b also identify the two entry/exit links with which they are supposed to be connected.

¹³ A ‘must’ can be inserted in front of all ‘all’s,

```

329 wf_p_hub_interf:  $N'_\Delta \rightarrow \mathbf{Bool}$ 
329 wf_p_hub_interf( $n_o, hil, (exll, \_, \_)$ )  $\equiv$ 
329    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{inds} \text{ exll} \Rightarrow$ 
329     let  $h = \text{get\_H}(hil(i))(n'_\Delta)$  in
329     let  $lis = \mathbf{obs\_mereo\_H}(h)$  in
329     let  $lis' = lis \setminus \text{xtr\_lis}(n')$  in
329      $lis' = \text{xtr\_lis}(exll(i))$  end end end

```

330 The mereology of each toll-road intersection hub must identify
 a the entry/exit links
 b and exactly the toll-road ‘up’ and ‘down’ links
 c with which they are supposed to be connected.

```

330 wf_toll_road_isect_hub_iface:  $N_{\mathcal{J}} \rightarrow \mathbf{Bool}$ 
330 wf_toll_road_isect_hub_iface( $\_, \_, (exll, hl, lll)$ )  $\equiv$ 
330    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{inds} \text{ hl} \Rightarrow$ 
330      $\mathbf{obs\_mereo\_H}(hl(i)) =$ 
330a      $\text{xtr\_lis}(exll(i)) \cup$ 
330     case  $i$  of
330b        $1 \rightarrow \text{xtr\_lis}(lll(1)),$ 
330b       len  $hl \rightarrow \text{xtr\_lis}(lll(\mathbf{len} \text{ hl} - 1))$ 
330b        $\_ \rightarrow \text{xtr\_lis}(lll(i)) \cup \text{xtr\_lis}(lll(i - 1))$ 
330     end

```

331 The mereology of the entry/exit links must identify exactly the
 a interface hubs and the
 b toll-road intersection hubs
 c with which they are supposed to be connected.

```

331 wf_exll:  $(L \times L)^* \times Hl^* \times H^* \rightarrow \mathbf{Bool}$ 
331 wf_exll( $exll, hil, hl$ )  $\equiv$ 
331    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{len} \text{ exll}$ 
331     let  $(hi, (el, xl), h) = (hil(i), exll(i), hl(i))$  in
331      $\mathbf{obs\_mereo\_L}(el) = \mathbf{obs\_mereo\_L}(xl)$ 
331      $= \{hi\} \cup \{\mathbf{uid\_H}(h)\}$  end
331   pre:  $\mathbf{len} \text{ eell} = \mathbf{len} \text{ hil} = \mathbf{len} \text{ hl}$ 

```

332 The mereology of the toll-road ‘up’ and ‘down’ links must
 a identify exactly the toll-road intersection hubs
 b with which they are supposed to be connected.

```

332 wf_u_d_links:  $(L \times L)^* \times H^* \rightarrow \mathbf{Bool}$ 
332 wf_u_d_links( $lll, hl$ )  $\equiv$ 
332    $\forall i: \mathbf{Nat} \cdot i \in \mathbf{inds} \text{ lll} \Rightarrow$ 
332     let  $(ul, dl) = lll(i)$  in
332      $\mathbf{obs\_mereo\_L}(ul) = \mathbf{obs\_mereo\_L}(dl) =$ 
332a      $\mathbf{uid\_H}(hl(i)) \cup \mathbf{uid\_H}(hl(i + 1))$  end
332   pre:  $\mathbf{len} \text{ lll} = \mathbf{len} \text{ hl} + 1$ 

```

We have used some additional auxiliary functions:

```

xtr_his: H* → Hl-set
xtr_his(hl) ≡ {uid_Hl(h) | h: H • h ∈ elems hl}
xtr_lis: (L × L) → Ll-set
xtr_lis(l', l'') ≡ {uid_Ll(l')} ∪ {uid_Ll(l'')}
xtr_lis: (L × L)* → Ll-set
xtr_lis(III) ≡
  ∪ {xtr_lis(l', l'') | (l', l''): (L × L) • (l', l'') ∈ elems III}

```

333 The well-formedness of instantiated nets is now the conjunction of the individual well-formedness predicates above.

```

333 wf_instantiated_net: Nℒ → Bool
333 wf_instantiated_net(n'Δ, hil, (exll, hl, III))
324   wf_dist_toll_road_isect_hub_ids(hl)
325   ∧ wf_dist_toll_road_u_d_link_ids(III)
326   ∧ wf_dist_e_e_link_ids(exll)
327   ∧ wf_isolated_toll_road_isect_hubs(hil, hl)(n')
328   ∧ wf_p_hubs_pt_of_ord_net(hil)(n')
329   ∧ wf_p_hub_interf(n'Δ, hil, (exll, _, _))
330   ∧ wf_toll_road_isect_hub_iface(_, _, (exll, hl, III))
331   ∧ wf_exll(exll, hil, hl)
332   ∧ wf_u_d_links(III, hl)

```

Domain Instantiation — Abstraction

Example 5.8. . **Domain Requirements. Instantiation Road Net, Abstraction:** Domain instantiation has refined an abstract definition of net sorts, $n_{\mathcal{D}}:N_{\mathcal{D}}$, into a partially concrete definition of nets, $n_{\mathcal{L}}:N_{\mathcal{L}}$. We need to show the refinement relation:

- $\text{abstraction}(n_{\mathcal{L}}) = n_{\mathcal{D}}$.

value

```

334 abstraction: Nℒ → Nℒ
335 abstraction(n'Δ, hil, (exll, hl, III)) ≡
336   let nℒ:Nℒ •
336     let hs = obs_part_HSℒ(obs_part_HAℒ(n'ℒ)),
336         ls = obs_part_LSℒ(obs_part_LAℒ(n'ℒ)),
336         ths = elems hl,
336         eells = xtr_links(exll), IIIs = xtr_links(III) in
337     hsUths=obs_part_HSℒ(obs_part_HAℒ(nℒ))
338     ∧ lsUeellsUlls=obs_part_LSℒ(obs_part_LAℒ(nℒ))
339   nℒ end end

```

334 The abstraction function takes a concrete net, $n_{\mathcal{L}}:N_{\mathcal{L}}$, and yields an abstract net, $n_{\mathcal{D}}:N_{\mathcal{D}}$.

335 The abstraction function doubly decomposes its argument into constituent lists and sub-lists.

336 There is postulated an abstract net, $n_{\mathcal{D}}:N_{\mathcal{D}}$, such that

337 the hubs of the concrete net and toll-road equals those of the abstract net, and

338 the links of the concrete net and toll-road equals those of the abstract net.

339 And that abstract net, $n_{\mathcal{D}}:N_{\mathcal{D}}$, is postulated to be an abstraction of the concrete net.

Discussion

Domain descriptions, such as illustrated in [70, *Manifest Domains: Analysis & Description*] and in this chapter, model families of concrete, i.e., specifically occurring domains. Domain instantiation, as exemplified in this section (i.e., Sect. 5.4.2), “narrow down” these families. Domain instantiation, such as it is defined, cf. Definition 29 on Page 173, allows the requirements engineer to instantiate to a concrete instance of a very specific domain, that, for example, of the toll-road between *Bolzano Nord* and *Trento Sud* in Italy (i.e., $n=7$)¹⁴.

5.4.3 Domain Determination

Definition 30 Determination: By **domain determination** we mean a refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the *endurants: parts, materials and components*, as well as the *perdurants: functions, events and behaviours* of the partial domain requirements prescription less non-determinate, more determinate ■

Determinations usually render these concepts less general. That is, the value space of endurants that are made more determinate is “smaller”, contains fewer values, as compared to the endurants before determination has been “applied”.

Domain Determination: Example

We show an example of ‘domain determination’. It is expressed sôlely in terms of axioms over the concrete toll-road net type.

Example 5.9. . Domain Requirements. Determination Toll-roads: We focus only on the toll-road net. We single out only two ‘determinations’:

All Toll-road Links are One-way Links

340 *The entry/exit and toll-road links*

- a are always all one way links,
- b as indicated by the arrows of Fig. 5.1 on Page 174,
- c such that each pair allows traffic in opposite directions.

```

340 opposite_traffics: (L×L)* × (L×L)* → Bool
340 opposite_traffics(exll,lll) ≡
340   ∀ (lt,lf):(L×L) · (lt,lf) ∈ elems exll∩lll ⇒
340a   let (ltσ,lfσ) = (attr_LΣ(lt),attr_LΣ(lf)) in
340a'.   attr_LΩ(lt)={ltσ}∧attr_LΩ(lf)={lfσ}
340a''.  ∧ card ltσ = 1 = card lfσ
340   ∧ let ({(hi,hi')},{(hi'',hi''')}) = (ltσ,lfσ) in
340c   hi=hi''' ∧ hi'=hi''
340   end end

```

Predicates 340a'. and 340a''. express the same property.

All Toll-road Hubs are Free-flow

341 *The hub state spaces* are singleton sets of the toll-road hub states which always allow exactly these (and only these) crossings:
 a from entry links back to the paired exit links,

¹⁴ Here we disregard the fact that this toll-road does not start/end in neither *Bolzano Nord* nor *Trento Sud*.

b from *entry* links to emanating *toll-road links*,
 c from incident *toll-road links* to exit *links*, and
 d from incident *toll-road link* to emanating *toll-road links*.

```

341 free_flow_toll_road_hubs: (L×L)* × (L×L)* → Bool
341 free_flow_toll_road_hubs(exl, ll) ≡
341   ∀ i: Nat·i ∈ inds hl ⇒
341     attr_HΣ(hl(i)) =
341a     hσ_ex_ls(exl(i))
341b     ∪ hσ_et_ls(exl(i), (i, ll))
341c     ∪ hσ_tx_ls(exl(i), (i, ll))
341d     ∪ hσ_tt_ls(i, ll)

```

341a: from *entry* links back to the paired exit *links*:

```

341a hσ_ex_ls: (L×L) → LΣ
341a hσ_ex_ls(e, x) ≡ {(uid_Ll(e), uid_Ll(x))}

```

341b: from *entry* links to emanating *toll-road links*:

```

341b hσ_et_ls: (L×L) × (Nat × (em: L × in: L)*) → LΣ
341b hσ_et_ls((e, _), (i, ll)) ≡
341b   case i of
341b     2      → {(uid_Ll(e), uid_Ll(em(ll(1))))},
341b     len ll+1 → {(uid_Ll(e), uid_Ll(em(ll(len ll))))},
341b     —      → {(uid_Ll(e), uid_Ll(em(ll(i-1))))},
341b             (uid_Ll(e), uid_Ll(em(ll(i))))}
341b   end

```

The *em* and *in* in the toll-road link list $(em: L \times in: L)^*$ designate selectors for *emanating*, respectively *incident* links.

341c: from incident *toll-road links* to exit *links*:

```

341c hσ_tx_ls: (L×L) × (Nat × (em: L × in: L)*) → LΣ
341c hσ_tx_ls((_, x), (i, ll)) ≡
341c   case i of
341c     2      → {(uid_Ll(in(ll(1))), uid_Ll(x))},
341c     len ll+1 → {(uid_Ll(in(ll(len ll))), uid_Ll(x))},
341c     —      → {(uid_Ll(in(ll(i-1))), uid_Ll(x))},
341c             (uid_Ll(in(ll(i))), uid_Ll(x))}
341c   end

```

341d: from incident *toll-road link* to emanating *toll-road links*:

```

341d hσ_tt_ls: Nat × (em: L × in: L)* → LΣ
341d hσ_tt_ls(i, ll) ≡
341d   case i of
341d     2      → {(uid_Ll(in(ll(1))), uid_Ll(em(ll(1))))},
341d     len ll+1 → {(uid_Ll(in(ll(len ll))), uid_Ll(em(ll(len ll))))},
341d     —      → {(uid_Ll(in(ll(i-1))), uid_Ll(em(ll(i-1))))},
341d             (uid_Ll(in(ll(i))), uid_Ll(em(ll(i))))}
341d   end

```

The example above illustrated ‘domain determination’ with respect to *endurants*. Typically “*endurant determination*” is expressed in terms of axioms that limit state spaces — where “*endurant instantiation*” typically “*limited*” the mereology of *endurants*: how parts are related to one another. We shall not exemplify domain determination with respect to *perdurants*.

Discussion

The borderline between instantiation and determination is fuzzy. Whether, as an example, fixing the number of toll-road intersection hubs to a constant value, e.g., $n=7$, is instantiation or determination, is really a matter of choice !

5.4.4 Domain Extension

Definition 31 Extension: By **domain extension** we understand *the introduction of endurants (see Sect. 5.4.4) and perdurants (see Sect. 5.5.2) that were not feasible in the original domain, but for which, with computing and communication, and with new, emerging technologies, for example, sensors, actuators and satellites, there is the possibility of feasible implementations, hence the requirements, that what is introduced becomes part of the unfolding requirements prescription* ■

Endurant Extensions

Definition 32 Endurant Extension: By an **endurant extension** we understand the introduction of one or more endurants into the projected, instantiated and determined domain $\mathcal{D}_{\mathcal{R}}$ resulting in domain $\mathcal{D}_{\mathcal{R}'}$, such that these form a *conservative extension* of the theory, $\mathcal{T}_{\mathcal{D}_{\mathcal{R}}}$ denoted by the domain requirements $\mathcal{D}_{\mathcal{R}}$ (i.e., “before” the extension), that is: every theorem of $\mathcal{T}_{\mathcal{D}_{\mathcal{R}'}}$ is still a theorem of $\mathcal{T}_{\mathcal{D}_{\mathcal{R}}}$.

Usually domain extensions involve one or more of the already introduced sorts. In Example 5.10 we introduce (i.e., “extend”) vehicles with GPSS-like sensors, and introduce toll-gates with entry sensors, vehicle identification sensors, gate actuators and exit sensors. Finally road pricing calculators are introduced.

Example 5.10. . Domain Requirements — Endurant Extension: We present the extensions in several steps. Some of them will be developed in this section. Development of the remaining will be deferred to Sect. 5.5.1. The reason for this deferment is that those last steps are examples of *interface requirements*. The initial extension-development steps are: [a] vehicle extension, [b] sort and unique identifiers of road price calculators, [c] vehicle to road pricing calculator channel, [d] sorts and dynamic attributes of toll-gates, [e] road pricing calculator attributes, [f] “total” system state, and [g] the overall system behaviour. This decomposition establishes system interfaces in “small, easy steps”.

[a] Vehicle Extension:

342 There is a domain, $\delta_{\mathcal{E}}:\Delta_{\mathcal{E}}$, which contains
 343 a fleet, $f_{\mathcal{E}}:F_{\mathcal{E}}$, that is,
 344 a set, $vs_{\mathcal{E}}:VS_{\mathcal{E}}$, of
 345 extended vehicles, $v_{\mathcal{E}}:V_{\mathcal{E}}$ — their extension amounting to
 346 a dynamic reactive attribute, whose value, $ti_gpos:TiGpos$, at any time, reflects that vehicle’s
time-stamped global position.¹⁵
 347 The vehicle’s GNSS receiver calculates, loc_pos , its local position, $lpos:LPos$, based on these signals.
 348 Vehicles access these *external attributes* via the *external attribute* channel, $attr_TiGPos_ch$.

¹⁵ We refer to literature on GNSS, *global navigation satellite systems*. The simple vehicle position, $vp:SVPos$, is determined from three to four time-stamped signals received from a like number of GNSS satellites [121].


```

type
342  $\Delta_{\mathcal{E}}$ 
343  $F_{\mathcal{E}}$ 
344  $VS_{\mathcal{E}} = V_{\mathcal{E}}\text{-set}$ 
345  $V_{\mathcal{E}}$ 
346  $TiGPos = \mathbb{T} \times GPos$ 
347  $GPos, LPos$ 
value
342  $\delta_{\mathcal{E}}: \Delta_{\mathcal{E}}$ 
343 obs_part $_F_{\mathcal{E}}: \Delta_{\mathcal{E}} \rightarrow F_{\mathcal{E}}$ 
343  $f = \mathbf{obs\_part}_F_{\mathcal{E}}(\delta_{\mathcal{E}})$ 
344 obs_part $_VS_{\mathcal{E}}: F_{\mathcal{E}} \rightarrow VS_{\mathcal{E}}$ 
344  $vs = \mathbf{obs\_part}_VS_{\mathcal{E}}(f)$ 
344  $vis = \mathbf{xtr\_vis}(vs)$ 
346  $\mathbf{attr\_TiGPos\_ch}[vi]?$ 
347  $\mathbf{loc\_pos}: GPos \rightarrow LPos$ 
channel
347  $\{\mathbf{attr\_TiGPos\_ch}[vi] \mid vi: VI \cdot vi \in vis\}: TiGPos$ 

```

We define two auxiliary functions,

349 $\mathbf{xtr_vs}$, which given a domain, or a fleet, extracts its set of vehicles, and
 350 $\mathbf{xtr_vis}$ which given a set of vehicles generates their unique identifiers.

```

value
349  $\mathbf{xtr\_vs}: (\Delta_{\mathcal{E}} \mid F_{\mathcal{E}} \mid VS_{\mathcal{E}}) \rightarrow V_{\mathcal{E}}\text{-set}$ 
349  $\mathbf{xtr\_vs}(\mathit{arg}) \equiv$ 
349  $\mathbf{is\_}\Delta_{\mathcal{E}}(\mathit{arg}) \rightarrow \mathbf{obs\_part}_VS_{\mathcal{E}}(\mathbf{obs\_part}_F_{\mathcal{E}}(\mathit{arg})),$ 
349  $\mathbf{is\_}F_{\mathcal{E}}(\mathit{arg}) \rightarrow \mathbf{obs\_part}_VS_{\mathcal{E}}(\mathit{arg}),$ 
349  $\mathbf{is\_}VS_{\mathcal{E}}(\mathit{arg}) \rightarrow \mathit{arg}$ 
350  $\mathbf{xtr\_vis}: (\Delta_{\mathcal{E}} \mid F_{\mathcal{E}} \mid VS_{\mathcal{E}}) \rightarrow VI\text{-set}$ 
350  $\mathbf{xtr\_vis}(\mathit{arg}) \equiv \{\mathbf{uid\_}VI(v) \mid v \in \mathbf{xtr\_vs}(\mathit{arg})\}$ 

```

[b] Road Pricing Calculator: Basic Sort and Unique Identifier:

351 The domain $\delta_{\mathcal{E}}: \Delta_{\mathcal{E}}$, also contains a pricing calculator, $c: C_{\delta_{\mathcal{E}}}$, with unique identifier $ci: CI$.

```

type
351  $C, CI$ 
value
351 obs_part $_C: \Delta_{\mathcal{E}} \rightarrow C$ 
351 uid $_CI: C \rightarrow CI$ 
351  $c = \mathbf{obs\_part}_C(\delta_{\mathcal{E}})$ 
351  $ci = \mathbf{uid\_}CI(c)$ 

```

[c] Vehicle to Road Pricing Calculator Channel:

352 Vehicles can, on their own volition, offer the timed local position, $\mathbf{viti_lpos}: VITiLPos$
 353 to the pricing calculator, $c: C_{\mathcal{E}}$ along a vehicles-to-calculator channel, $\mathbf{v_c_ch}$.

```

type
352  $VITiLPos = VI \times (\mathbb{T} \times LPos)$ 
channel
353  $\{\mathbf{v\_c\_ch}[vi, ci] \mid vi: VI, ci: CI \cdot vi \in vis \wedge ci = \mathbf{uid\_}C(c)\}: VITiLPos$ 

```

[d] Toll-gate Sorts and Dynamic Types:

We extend the domain with toll-gates for vehicles entering and exiting the toll-road entry and exit links. Figure 5.2 illustrates the idea of gates.

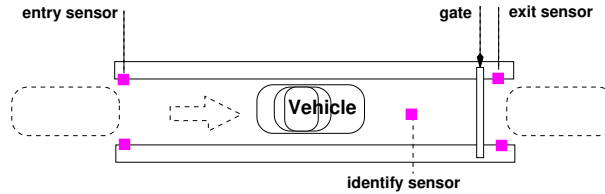


Fig. 5.2. A toll plaza gate

Figure 5.2 is intended to illustrate a vehicle entering (or exiting) a toll-road arrival link. The toll-gate is equipped with three sensors: an arrival sensor, a vehicle identification sensor and an departure sensor. The arrival sensor serves to prepare the vehicle identification sensor. The departure sensor serves to prepare the gate for closing when a vehicle has passed. The vehicle identify sensor identifies the vehicle and “delivers” a pair: the current time and the vehicle identifier. Once the vehicle identification sensor has identified a vehicle the gate opens and a message is sent to the road pricing calculator as to the passing vehicle’s identity and the identity of the link associated with the toll-gate (see Items 370- 371 on the facing page).

354 The domain contains the extended net, $n:N_{\mathcal{E}}$,

355 with the net extension amounting to the toll-road net, $TRN_{\mathcal{E}}$, that is, the instantiated toll-road net, $trn:TRN_{\mathcal{G}}$, is extended, into $trn:TRN_{\mathcal{E}}$, with *entry*, $eg:EG$, and *exit*, $xg:XG$, toll-gates.

From entry- and exit-gates we can observe

356 their unique identifier and

357 their mereology: pairs of entry-, respectively exit link and calculator unique identifiers; further

358 a pair of gate entry and exit sensors modeled as *external attribute* channels, $(ges:ES, gls:XS)$, and

359 a time-stamped vehicle identity sensor modeled as *external attribute* channels.

type

354 $N_{\mathcal{E}}$

355 $TRN_{\mathcal{E}} = (EG \times XG)^* \times TRN_{\mathcal{G}}$

356 GI

value

354 **obs_part** $_N_{\mathcal{E}}$: $\Delta_{\mathcal{E}} \rightarrow N_{\mathcal{E}}$

355 **obs_part** $_TRN_{\mathcal{E}}$: $N_{\mathcal{E}} \rightarrow TRN_{\mathcal{E}}$

356 **uid** $_G$: $(EG|XG) \rightarrow GI$

357 **obs_mereo** $_G$: $(EG|XG) \rightarrow (LI \times CI)$

355 $trn:TRN_{\mathcal{E}} = \mathbf{obs_part_TRN}_{\mathcal{E}}(\delta_{\mathcal{E}})$

channel

358 $\{attr_entry_ch[gi]|gi:GI \times tr_eGlds(trn)\}$ "enter"

358 $\{attr_exit_ch[gi]|gi:GI \times tr_xGlds(trn)\}$ "exit"

359 $\{attr_identity_ch[gi]|gi:GI \times tr_Glds(trn)\}$ TIVI

type

359 $TIVI = \mathbb{T} \times VI$

We define some **auxiliary functions** over toll-road nets, $trn:TRN_{\mathcal{E}}$:

360 $xtr_eG\ell$ extracts the *list* of entry gates,
 361 $xtr_xG\ell$ extracts the *list* of exit gates,
 362 xtr_eGlds extracts the set of entry gate identifiers,
 363 xtr_xGlds extracts the set of exit gate identifiers,
 364 xtr_Gs extracts the set of all gates, and
 365 xtr_Glds extracts the set of all gate identifiers.

value

360 $xtr_eG\ell: TRN_{\mathcal{E}} \rightarrow EG^*$
 360 $xtr_eG\ell(pgl, _) \equiv \{eg | (eg, xg): (EG, XG) \bullet (eg, xg) \in \mathbf{elems} \ pgl\}$
 361 $xtr_xG\ell: TRN_{\mathcal{E}} \rightarrow XG^*$
 361 $xtr_xG\ell(pgl, _) \equiv \{xg | (eg, xg): (EG, XG) \bullet (eg, xg) \in \mathbf{elems} \ pgl\}$
 362 $xtr_eGlds: TRN_{\mathcal{E}} \rightarrow \mathbf{GI-set}$
 362 $xtr_eGlds(pgl, _) \equiv \{\mathbf{uid_GI}(g) | g: EG \bullet g \in xtr_eGs(pgl, _)\}$
 363 $xtr_xGlds: TRN_{\mathcal{E}} \rightarrow \mathbf{GI-set}$
 363 $xtr_xGlds(pgl, _) \equiv \{\mathbf{uid_GI}(g) | g: EG \bullet g \in xtr_xGs(pgl, _)\}$
 364 $xtr_Gs: TRN_{\mathcal{E}} \rightarrow \mathbf{G-set}$
 364 $xtr_Gs(pgl, _) \equiv xtr_eGs(pgl, _) \cup xtr_xGs(pgl, _)$
 365 $xtr_Glds: TRN_{\mathcal{E}} \rightarrow \mathbf{GI-set}$
 365 $xtr_Glds(pgl, _) \equiv xtr_eGlds(pgl, _) \cup xtr_xGlds(pgl, _)$

366 A **well-formedness condition** expresses

- a that there are as many entry end exit gate pairs as there are toll-plazas,
- b that all gates are uniquely identified, and
- c that each entry [exit] gate is paired with an entry [exit] link and has that link's unique identifier as one element of its mereology, the other elements being the calculator identifier and the vehicle identifiers.

The well-formedness relies on awareness of

367 the unique identifier, $ci:CI$, of the road pricing calculator, $c:C$, and
 368 the unique identifiers, $vis:VI\text{-set}$, of the fleet vehicles.

axiom

366 $\forall n: \mathbb{N}_{\mathcal{R}_3}, trn: TRN_{\mathcal{R}_3} \bullet$
 366 **let** $(exgl, (exl, hl, lll)) = \mathbf{obs_part_TRN}_{\mathcal{R}_3}(n)$ **in**
 366a **len** $exgl = \mathbf{len} \ exl = \mathbf{len} \ hl = \mathbf{len} \ lll + 1$
 366b \wedge **card** $xtr_Glds(exgl) = 2 * \mathbf{len} \ exgl$
 366c $\wedge \forall i: \mathbf{Nat} \bullet i \in \mathbf{inds} \ exgl \bullet$
 366c **let** $((eg, xg), (el, xl)) = (exgl(i), exl(i))$ **in**
 366c **obs_mereo_G**(eg) = $(\mathbf{uid_U}(el), ci, vis)$
 366c \wedge **obs_mereo_G**(xg) = $(\mathbf{uid_U}(xl), ci, vis)$
 366 **end end**

[e] Toll-gate to Calculator Channels:

369 We distinguish between *entry* and *exit* gates.

370 Toll road entry and exit gates offers the road pricing calculator a pair: whether it is an entry or an exit gates, and pair of the passing vehicle's identity and the time-stamped identity of the link associated with the toll-gate

371 to the road pricing calculator via a (gate to calculator) channel.

```

type
369 EE = "entry"|"exit"
370 EEViTiLI = EE × (VI × (T × SonL))
channel
371 {g_c.ch[gi,ci]|gi:GI•gi ∈ gis}:EETiViLI

```

[f] Road Pricing Calculator Attributes:

372 The road pricing attributes include a programmable traffic map, $\text{trm}:\text{TRM}$, which, for each vehicle inside the toll-road net, records a chronologically ordered list of each vehicle's timed position, (τ, lpos) , and

373 a static (total) road location function, $\text{vplf}:\text{VPLF}$. The vehicle *position location function*, $\text{vplf}:\text{VPLF}$, which, given a local position, $\text{lpos}:\text{LPos}$, yields *either* the simple vehicle position, $\text{svpos}:\text{SVPos}$, designated by the GNSS-provided position, or yields the response that the provided position is off the toll-road net. The $\text{vplf}:\text{VPLF}$ function is constructed, construct_vplf ,

374 from awareness, of a geodetic road map, GRM , of the topology of the extended net, $n_{\mathcal{E}}:\mathbb{N}_{\mathcal{E}}$, including the mereology and the geodetic attributes of links and hubs.

```

type
372 TRM = VI  $\rightarrow_{\text{h}}$  (T × SVPos)*
373 VPLF = GRM → LPos → (SVPos | "off_N")
374 GRM
value
372 attr_TRM:  $C_{\mathcal{E}} \rightarrow \text{TRM}$ 
373 attr_VPLF:  $C_{\mathcal{E}} \rightarrow \text{VPLF}$ 

```

The geodetic road map maps geodetic locations into hub and link identifiers.

284 Geodetic link locations represent the set of point locations of a link.
 280a Geodetic hub locations represent the set of point locations of a hub.
 375 A geodetic road map maps geodetic link locations into link identifiers and geodetic hub locations into hub identifiers.
 376 We sketch the construction, geo_GRM , of geodetic road maps.

```

type
375 GRM = (GeoL  $\rightarrow_{\text{h}}$  LI) ∪ (GeoH  $\rightarrow_{\text{h}}$  HI)
value
376 geo_GRM:  $\mathbb{N} \rightarrow \text{GRM}$ 
376 geo_GRM(n) ≡
376 let ls = xtr_links(n), hs = xtr_hubs(n) in
376 [attr_GeoL(l)  $\mapsto$  uid_LI(l)|l:L•l ∈ ls]
376 ∪
376 [attr_GeoH(h)  $\mapsto$  uid_HI(h)|h:H•h ∈ hs] end

```

377 The $\text{vplf}:\text{VPLF}$ function obtains a simple vehicle position, svpos , from a geodetic road map, $\text{grm}:\text{GRM}$, and a local position, lpos :

```

value
377 obtain_SVPos:  $\text{GRM} \rightarrow \text{LPos} \rightarrow \text{SVPos}$ 
377 obtain_SVPos(grm)(lpos) as svpos
377 post: case svpos of
377     SatH(hi) → within(lpos, grm(hi)),
377     SonL(li) → within(lpos, grm(li)),
377     "off_N" → true end

```

where *within* is a predicate which holds if its first argument, a local position calculated from a GNSS-generated global position, falls within the point set representation of the geodetic locations of a link or a hub. The design of the *obtain_SVPos* represents an interesting challenge.

[g] “Total” System State:

Global values:

378 There is a given domain, $\delta_{\mathcal{E}}:\Delta_{\mathcal{E}}$;
 379 there is the net, $n_{\mathcal{E}}:N_{\mathcal{E}}$, of that domain;
 380 there is toll-road net, $trn_{\mathcal{E}}:TRN_{\mathcal{E}}$, of that net;
 381 there is a set, $egs_{\mathcal{E}}:EG\text{-set}$, of entry gates;
 382 there is a set, $xgs_{\mathcal{E}}:XG\text{-set}$, of exit gates;
 383 there is a set, $gis_{\mathcal{E}}:GI\text{-set}$, of gate identifiers;
 384 there is a set, $vs_{\mathcal{E}}:V_{\mathcal{E}}\text{-set}$, of vehicles;
 385 there is a set, $vis_{\mathcal{E}}:VI\text{-set}$, of vehicle identifiers;
 386 there is the road-pricing calculator, $c_{\mathcal{E}}:C_{\mathcal{E}}$ and
 387 there is its unique identifier, $ci_{\mathcal{E}}:CI$.

value

378 $\delta_{\mathcal{E}}:\Delta_{\mathcal{E}}$
 379 $n_{\mathcal{E}}:N_{\mathcal{E}} = \mathbf{obs_part_}N_{\mathcal{E}}(\delta_{\mathcal{E}})$
 380 $trn_{\mathcal{E}}:TRN_{\mathcal{E}} = \mathbf{obs_part_}TRN_{\mathcal{E}}(n_{\mathcal{E}})$
 381 $egs_{\mathcal{E}}:EG\text{-set} = \mathbf{xtr_}egs(trn_{\mathcal{E}})$
 382 $xgs_{\mathcal{E}}:XG\text{-set} = \mathbf{xtr_}xgs(trn_{\mathcal{E}})$
 383 $gis_{\mathcal{E}}:XG\text{-set} = \mathbf{xtr_}gis(trn_{\mathcal{E}})$
 384 $vs_{\mathcal{E}}:V_{\mathcal{E}}\text{-set} = \mathbf{obs_part_}VS(\mathbf{obs_part_}F_{\mathcal{E}}(\delta_{\mathcal{E}}))$
 385 $vis_{\mathcal{E}}:VI\text{-set} = \{\mathbf{uid_}VI(v_{\mathcal{E}}) \mid v_{\mathcal{E}}:V_{\mathcal{E}} \bullet v_{\mathcal{E}} \in vs_{\mathcal{E}}\}$
 386 $c_{\mathcal{E}}:C_{\mathcal{E}} = \mathbf{obs_part_}C_{\mathcal{E}}(\delta_{\mathcal{E}})$
 387 $ci_{\mathcal{E}}:CI_{\mathcal{E}} = \mathbf{uid_}CI(c_{\mathcal{E}})$

In the following we shall omit the cumbersome \mathcal{E} subscripts.

[h] “Total” System Behaviour:

The signature and definition of the system behaviour is sketched as are the signatures of the vehicle, toll-gate and road pricing calculator. We shall model the behaviour of the road pricing system as follows: we shall not model behaviours nets, hubs and links; thus we shall model only the behaviour of vehicles, *veh*, the behaviour of toll-gates, *gate*, and the behaviour of the road-pricing calculator, *calc*. The behaviours of vehicles and toll-gates are presented here. But the behaviour of the road-pricing calculator is “deferred” till Sect. 5.5.1 since it reflects an interface requirements.

388 The road pricing system behaviour, *sys*, is expressed as
 a the parallel, \parallel , (distributed) composition of the behaviours of all vehicles,
 b with the parallel composition of the parallel (likewise distributed) composition of the behaviours of all entry gates,
 c with the parallel composition of the parallel (likewise distributed) composition of the behaviours of all exit gates,
 d with the parallel composition of the behaviour of the road-pricing calculator,

value

388 $\mathbf{sys}: \mathbf{Unit} \rightarrow \mathbf{Unit}$
 388 $\mathbf{sys}() \equiv$
 388a $\parallel \{\mathbf{veh}_{\mathbf{uid_}V(v)}(\mathbf{obs_mereo_}V(v)) \mid v:V \bullet v \in vs\}$

```

388b  || || {gateuidEG(eg)(obs_mereo_G(eg),"entry")|eg:EG•eg ∈ egs}
388c  || || {gateuidXG(xg)(obs_mereo_G(xg),"exit")|xg:XG•xg ∈ xgs}
388d  ||  calcuidLC(vis,gis)(rlf)(trm)

389  vehvi: (ci:CI×gis:GI-set) → in attr_TiGPos[vi] out v_c_ch[vi,ci] Unit
395  gategi: (ci:CI×VI-set×LI)×ee:EE →
395      in attr_entry_ch[gi,ci],attr_id_ch[gi,ci],attr_exit_ch[gi,ci]
395      out attr_barrier_ch[gi],g_c_ch[gi,ci] Unit
429  calcci: (vis:VI-set×gis:GI-set)×VPLF→TRM→
429      in {v_c_ch[vi,ci]|vi:VI•vi ∈ vis},{g_c_ch[gi,ci]|gi:GI•gi ∈ gis} Unit

```

We consider "entry" or "exit" to be a static attribute of toll-gates. The behaviour signatures were determined as per the techniques presented in [70, Sect. 4.1.1 and 4.5.2].

Vehicle Behaviour: We refer to the vehicle behaviour, in the domain, described in Sect. 5.2's The Road Traffic System Behaviour Items 307 and Items 308, Page 166 and, projected, Page 172.

389 Instead of moving around by explicitly expressed internal non-determinism¹⁶ vehicles move around by unstated internal non-determinism and instead receive their current position from the global positioning subsystem.

390 At each moment the vehicle receives its time-stamped global position, $(\tau, gpos):TiGPos$,

391 from which it calculates the local position, $lpos:VPos$

392 which it then communicates, with its vehicle identification, $(vi, (\tau, lpos))$, to the road pricing subsystem

393 whereupon it resumes its vehicle behaviour.

value

```

389  vehvi: (ci:CI×gis:GI-set) →
389      in attr_TiGPos_ch[vi] out v_c_ch[vi,ci] Unit
389  vehvi(ci,gis) ≡
390      let  $(\tau, gpos) = attr\_TiGPos\_ch[vi]?$  in
391      let  $lpos = loc\_pos(gpos)$  in
392      v_c_ch[vi,ci] ! (vi, ( $\tau, lpos$ )) ;
393      vehvi(ci,gis) end end
389      pre vi ∈ vis

```

The *vehicle* signature has $attr_TiGPos_ch[vi]$ model an external vehicle attribute and $v_c_ch[vi,ci]$ the *embedded attribute sharing* [70, Sect. 4.1.1 and 4.5.2] between vehicles (their position) and the price calculator's road map. The above behaviour represents an assumption about the behaviour of vehicles. If we were to design software for the monitoring and control of vehicles then the above vehicle behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers' behaviour when entering, passing and exiting toll-gates, about the proper function of the GNSS equipment, and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of *domain facets* such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [45].

Gate Behaviour: The entry and the exit gates have "vehicle enter", "vehicle exit" and "timed vehicle identification" sensors. The following assumption can now be made: during the time interval between a gate's vehicle "entry" sensor having first sensed a vehicle entering that gate and that gate's "exit" sensor having last sensed that vehicle leaving that gate that gate's vehicle time and "identify" sensor registers the time when the vehicle is entering the gate and that vehicle's unique identification. We sketch the toll-gate behaviour:

¹⁶ We refer to Items 307b, 307c on Page 165 and 308b, 308(b)ii, 308c on Page 166

394 We parameterise the toll-gate behaviour as either an entry or an exit gate.
 395 Toll-gates operate autonomously and cyclically.
 396 The `attr_enter_ch` event “triggers” the behaviour specified in formula line Item 397–399 starting with a “Raise” barrier action.
 397 The time-of-passing and the identity of the passing vehicle is sensed by `attr_passing_ch` channel events.
 398 Then the road pricing calculator is informed of time-of-passing and of the vehicle identity `vi` and the link `li` associated with the gate – and with a “Lower” barrier action.
 399 And finally, after that vehicle has left the entry or exit gate the barrier is again “Lower”ed and
 400 that toll-gate’s behaviour is resumed.

type

394 `EE = "enter" | "exit"`

value

```

395 gategi: (ci:CI×VI-set×LI)×ee:EE →
395   in attr_enter_ch[gi],attr_passing_ch[gi],attr_leave_ch[gi]
395   out attr_barrier_ch[gi],g_c_ch[gi,ci] Unit
395 gategi((ci,vis,li),ee) ≡
396   attr_enter_ch[gi] ? ; attr_barrier_ch[gi] ! "Lower"
397   let (τ,vi) = attr_passing_ch[gi] ? in assert vi ∈ vis
398   (attr_barrier_ch[gi] ! "Raise"
398   || g_c_ch[gi,ci] ! (ee,(vi,(τ,SonL(li)))));
399   attr_leave_ch[gi] ? ; attr_barrier_ch[gi] ! "Lower"
400   gategi((ci,vis,li),ee)
395   end
395   pre li ∈ lis

```

The *gate* signature’s `attr_enter_ch[gi]`, `attr_passing_ch[gi]`, `attr_barrier_ch[gi]` and `attr_leave_ch[gi]` model respective *external attributes* [70, Sect. 4.1.1 and 4.5.2] (the `attr_barrier_ch[gi]` models reactive (i.e., output) attribute), while `g_c_ch[gi,ci]` models the *embedded attribute sharing* between gates (their identification of vehicle positions) and the calculator road map. The above behaviour represents an assumption about the behaviour of toll-gates. If we were to design software for the monitoring and control of toll-gates then the above gate behaviour would have to be refined in order to serve as a proper interface requirements. The refinement would include handling concerns about the drivers’ behaviour when entering, passing and exiting toll-gates, about the proper function of the entry, passing and exit sensors, about the proper function of the gate barrier (opening and closing), and about the safe communication with the road price calculator. The above concerns would already have been addressed in a model of *domain facets* such as *human behaviour*, *technology support*, proper tele-communications *scripts*, etcetera. We refer to [45] ■

We shall define the *calculator* behaviour in Sect. 5.5.1 on Page 193. The reason for this deferral is that it exemplifies *interface requirements*.

Discussion

The requirements assumptions expressed in the specifications of the vehicle and gate behaviours assume that these behave in an orderly fashion. But they seldom do! The `attr_TiGPos_ch` sensor may fail. And so may the `attr_enter_ch`, `attr_passing_ch`, and `attr_leave_ch` sensors and the `attr_barrier_ch` actuator. These attributes represent *support technology* facets. They can fail. To secure fault tolerance one must prescribe very carefully what counter-measures are to be taken and/or the safety assumptions. We refer to [264, 162, 192]. They cover three alternative approaches to the handling of fault tolerance. Either of the approaches can be made to fit with our approach. First one can pursue our approach to where we stand now. Then we join the approaches of either of [264, 162, 192]. [162] likewise decompose the requirements prescription as is suggested here.

5.4.5 Requirements Fitting

Often a domain being described “fits” onto, is “adjacent” to, “interacts” in some areas with, another domain: *transportation with logistics, health-care with insurance, banking with securities trading and/or insurance*, and so on. The issue of requirements fitting arises when two or more software development projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments.

We thus assume that there are n domain requirements developments, $d_{r_1}, d_{r_2}, \dots, d_{r_n}$, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description.

Definition 33 Requirements Fitting: By **requirements fitting** we mean a *harmonisation* of $n > 1$ domain requirements that have overlapping (shared) not always consistent parts and which results in n *partial domain requirements*, $p_{d_{r_1}}, p_{d_{r_2}}, \dots, p_{d_{r_n}}$, and m *shared domain requirements*, $s_{d_{r_1}}, s_{d_{r_2}}, \dots, s_{d_{r_m}}$, that “fit into” two or more of the partial domain requirements ■ The above definition pertains to the result of ‘fitting’. The next definition pertains to the act, or process, of ‘fitting’.

Definition 34 Requirements Harmonisation: By **requirements harmonisation** we mean a number of alternative and/or co-ordinated prescription actions, one set for each of the domain requirements actions: *Projection, Instantiation, Determination* and *Extension*. They are – we assume n separate software product requirements: *Projection*: If the n product requirements do not have the same projections, then identify a common projection which they all share, and refer to it as the *common projection*. Then develop, for each of the n product requirements, if required, a *specific projection* of the common one. Let there be m such specific projections, $m \leq n$. *Instantiation*: First instantiate the common projection, if any instantiation is needed. Then for each of the m specific projections instantiate these, if required. *Determination*: Likewise, if required, “perform” “determination” of the possibly instantiated common projection, and, similarly, if required, “perform” “determination” of the up to m possibly instantiated projections. *Extension*: Finally “perform extension” likewise: First, if required, of the common projection (etc.), then, if required, on the up m specific projections (etc.). These harmonization developments may possibly interact and may need to be iterated ■

By a **partial domain requirements** we mean a domain requirements which is short of (that is, is missing) some prescription parts: text and formula ■ By a **shared domain requirements** we mean a domain requirements ■ By **requirements fitting** m *shared domain requirements* texts, $sdrs$, into n *partial domain requirements* we mean that there is for each *partial domain requirements*, pdr_i , an identified, non-empty subset of $sdrs$ (could be all of $sdrs$), $ssdrs_i$, such that textually conjoining $ssdrs_i$ to pdr_i , i.e., $ssdrs_i \oplus pdr_i$ can be claimed to yield the “original” d_{r_i} , that is, $\mathcal{M}(ssdrs_i \oplus pdr_i) \subseteq \mathcal{M}(d_{r_i})$, where \mathcal{M} is a suitable meaning function over prescriptions ■

5.4.6 Discussion

Facet-oriented Fittings: An altogether different way of looking at domain requirements may be achieved when also considering domain facets — not covered in neither the example of Sect. 5.2 nor in this section (i.e., Sect. 5.4) nor in the following two sections. We refer to [45].

Example 5.11. . Domain Requirements — Fitting: Example 5.10 hints at three possible sets of interface requirements: (i) for a road pricing [sub-]system, as will be illustrated in Sect. 5.5.1; (ii) for a vehicle monitoring and control [sub-]system, and (iii) for a toll-gate monitoring and control [sub-]system. The vehicle monitoring and control [sub-]system would focus on implementing the vehicle behaviour, see Items 389- 393 on Page 186. The toll-gate monitoring and control [sub-]system would focus on implementing the calculator behaviour, see Items 395- 400 on Page 187. The fitting amounts

to (a) making precise the narrative and formal texts specific to each of the three (i–iii) separate sub-system requirements are kept separate; (b) ensuring that meaning-wise shared texts that have different names for meaning-wise identical entities have these names renamed appropriately; (c) that these texts are subject to commensurate and ameliorated further requirements development; etcetera ■

5.5 Interface and Derived Requirements

We remind the reader that **interface requirements** can be expressed only using terms from both the domain and the machine ■ Users are not part of the machine. So no reference can be made to users, such as “*the system must be user friendly*”, and the like!¹⁷ By **interface requirements** we [also] mean *requirements prescriptions which refines and extends the domain requirements by considering those requirements of the domain requirements whose endurants (parts, materials) and perdurants (actions, events and behaviours) are “shared” between the domain and the machine (being requirements prescribed)* ■ The two *interface requirements* definitions above go hand-in-hand, i.e., complement one-another.

By **derived requirements** we mean *requirements prescriptions which are expressed in terms of the machine concepts and facilities introduced by the emerging requirements* ■

5.5.1 Interface Requirements

Shared Phenomena

By **sharing** we mean (a) that *some or all properties* of an **endurant** is represented both in the domain and “inside” the machine, and that their machine representation must at suitable times reflect their state in the domain; and/or (b) that an **action** requires a sequence of several “on-line” interactions between the machine (being requirements prescribed) and the domain, usually a person or another machine; and/or (c) that an **event** arises either in the domain, that is, in the environment of the machine, or in the machine, and need be communicated to the machine, respectively to the environment; and/or (d) that a **behaviour** is manifested both by actions and events of the domain and by actions and events of the machine ■ So a systematic reading of the domain requirements shall result in an identification of all shared endurants, parts, materials and components; and perdurants actions, events and behaviours. Each such shared phenomenon shall then be individually dealt with: **endurant sharing** shall lead to interface requirements for data initialisation and refreshment as well as for access to endurant attributes; **action sharing** shall lead to interface requirements for interactive dialogues between the machine and its environment; **event sharing** shall lead to interface requirements for how such event are communicated between the environment of the machine and the machine; and **behaviour sharing** shall lead to interface requirements for action and event dialogues between the machine and its environment.

Environment–Machine Interface:

Domain requirements extension, Sect. 5.4.4, usually introduce new endurants into (i.e., ‘extend’ the) domain. Some of these endurants may become elements of the domain requirements. Others are to be projected “away”. Those that are let into the domain requirements either have their endurants represented, somehow, also in the machine, or have (some of) their properties, usually some attributes, accessed by the machine. Similarly for perdurants. Usually the machine representation of shared perdurants access (some of) their properties, usually some attributes. The interface requirements must spell out which domain extensions are shared. Thus domain extensions may necessitate a review of domain projection, instantiations

¹⁷ So how do we cope with the statement: “*the system must be user friendly*”? We refer to Sect. 5.5.3 on Page 197 for a discussion of this issue.

and determination. In general, there may be several of the projection–eliminated parts (etc.) whose dynamic attributes need be accessed in the usual way, i.e., by means of `attr_XYZ_ch` channel communications (where XYZ is a projection–eliminated part attribute).

Example 5.12. . **Interface Requirements — Projected Extensions:** We refer to Fig. 5.2 on Page 182. We do not represent the GNSS system in the machine: only its “effect”: the ability to record global positions by accessing the GNSS attribute (channel):

channel

```
348 {attr_TiGPos_ch[vi]|vi:VI•vi ∈ xtr_VIs(vs)}: TiGPos
```

And we do not really represent the gate nor its sensors and actuator in the machine. But we do give an idealised description of the gate behaviour, see Items 395–400. Instead we represent their dynamic gate attributes:

- (358) the vehicle entry sensors (leftmost ■s),
- (358) the vehicle identity sensor (center ■), and
- (359) the vehicle exit sensors (rightmost ■s)

by channels — we refer to Example 5.10 (Sect. 5.5.1, Page 182):

channel

```
358 {attr_entry_ch[gi]|gi:GI•xtr_eGlds(trn)} "enter"
358 {attr_exit_ch[gi]|gi:GI•xtr_xGlds(trn)} "exit"
359 {attr_identity_ch[gi]|gi:GI•xtr_Glds(trn)} TIVI ■
```

Shared Endurants

Example 5.13. . **Interface Requirements. Shared Endurants:** The main shared endurants are the vehicles, the net (hubs, links, toll-gates) and the price calculator. As domain endurants hubs and links undergo changes, all the time, with respect to the values of several attributes: *length*, *geodetic information*, *names*, *wear and tear* (where-ever applicable), *last/next scheduled maintenance* (where-ever applicable), *state* and *state space*, and many others. Similarly for vehicles: their position, velocity and acceleration, and many other attributes. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modeling effort it must be secured that “standard” operations on nets, hubs and links can be supported by the chosen relational database system

Data Initialisation:

In general, one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes names and their types, and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

Example 5.14. . **Interface Requirements. Shared Endurant Initialisation:** The domain is that of the road net, $n:N$. By ‘shared road net initialisation’ we mean the “ab initio” establishment, “from scratch”, of a data base recording the properties of all links, $l:L$, and hubs, $h:H$, their unique identifications, `uid_L(l)` and `uid_H(h)`, their mereologies, `obs_mereo_L(l)` and `obs_mereo_H(h)`, the initial values of all their static and programmable attributes and the access values, that is, channel designations for all other attribute categories.

401 There are r_l and r_h “recorders” recording link, respectively hub properties – with each recorder having a unique identity.

402 Each recorder is charged with the recording of a set of links or a set of hubs according to some partitioning of all such.
 403 The recorders inform a central data base, `net_db`, of their recordings $(ri, hol, (u_j, m_j, attrs_j))$ where
 404 ri is the identity of the recorder,
 405 hol is either a `hub` or a `link` literal,
 406 $u_j = \mathbf{uid_L}(l)$ or $\mathbf{uid_H}(h)$ for some link or hub,
 407 $m_j = \mathbf{obs_mereo_L}(l)$ or $\mathbf{obs_mereo_H}(h)$ for that link or hub and
 408 $attrs_j$ are *attributes* for that link or hub — where *attributes* is a function which “records” all respective static and dynamic attributes (left undefined).

type401 `RI`**value**401 `rl, rh: NAT axiom rl > 0 ∧ rh > 0`**type**403 `M = RI × "link" × LNK | RI × "hub" × HUB`403 `LNK = LI × HI-set × LATTRS`403 `HUB = HI × LI-set × HATTRS`**value**402 `partitioning: L-set → Nat → (L-set)*`402 `| H-set → Nat → (H-set)*`402 `partitioning(s)(r) as sl`402 `post: len sl = r ∧ ∪ elems sl = s`402 `∧ ∀ si, sj: (L-set | H-set) •`402 `si ≠ {} ∧ sj ≠ {} ∧ {si, sj} ⊆ elems ss ⇒ si ∩ sj = {}`409 The $r_l + r_h$ recorder behaviours interact with the one `net_db` behaviour**channel**409 `r_db: RI × (LNK | HUB)`**value**409 `link_rec: RI → L-set → out r_db Unit`409 `hub_rec: RI → H-set → out r_db Unit`409 `net_db: Unit → in r_db Unit`410 The data base behaviour, `net_db`, offers to receive messages from the link and hub recorders.411 The data base behaviour, `net_db`, deposits these messages in respective variables.412 Initially there is a net, $n : N$,

413 from which is observed its links and hubs.

414 These sets are partitioned into r_l , respectively r_h length lists of non-empty links and hubs.415 The ab-initio data initialisation behaviour, `ab_initio_data`, is then the parallel composition of link recorder, hub recorder and data base behaviours with link and hub recorder being allotted appropriate link, respectively hub sets.

416 We construct, for technical reasons, as the reader will soon see, disjoint lists of link, respectively hub recorder identities.

value410 `net_db:`**variable**

```

411 lnk_db: (RI×LNK)-set
411 hub_db: (RI×HUB)-set
value
412 n:N
413 ls:L-set = obs_LS(obs_LS(n))
413 hs:H-set = obs_HS(obs_HS(n))
414 lsl:(L-set)* = partitioning(ls)(rl)
414 lhl:(H-set)* = partitioning(hs)(rh)
416 rill:RI* axiom len rill = rl = card elems rill
416 rihl:RI* axiom len rihl = rh = card elems rihl
415 ab_initio_data: Unit → Unit
415 ab_initio_data() ≡
415   || {lnk_rec(rill[i])(lsl[i])|i:Nat·1≤i≤rl} ||
415   || {hub_rec(rihl[i])(lhl[i])|i:Nat·1≤i≤rh}
415   || net_db()

```

417 The link and the hub recorders are near-identical behaviours.

418 They both revolve around an imperatively stated **for all ... do ... end**. The selected link (or hub) is inspected and the “data” for the data base is prepared from

419 the unique identifier,

420 the mereology, and

421 the attributes.

422 These “data” are sent, as a message, prefixed the senders identity, to the data base behaviour.

423 We presently leave the ... unexplained.

```

value
409 link_rec: RI → L-set → Unit
417 link_rec(ri,ls) ≡
418   for ∀ l:L·l ∈ ls do uid_L(l)
419     let lnk = (uid_L(l),
420               obs_mereo_L(l),
421               attributes(l)) in
422     rdb ! (ri,"link",lnk);
423   ... end
418 end

```

```

409 hub_rec: RI × H-set → Unit
417 hub_rec(ri,hs) ≡
418   for ∀ h:H·h ∈ hs do uid_H(h)
419     let hub = (uid_L(h),
420               obs_mereo_H(h),
421               attributes(h)) in
422     rdb ! (ri,"hub",hub);
423   ... end
418 end

```

424 The net_db data base behaviour revolves around a seemingly “never-ending” cyclic process.

425 Each cycle “starts” with acceptance of some,

426 either link or hub data.

427 If link data then it is deposited in the link data base,

428 if hub data then it is deposited in the hub data base.

```

value
424 net_db() ≡
425   let (ri,hol,data) = r_db ? in
426   case hol of
427     "link" → ... ; lnk_db := lnk_db ∪ (ri,data),
428     "hub"  → ... ; hub_db := hub_db ∪ (ri,data)
426   end end ;
424'  ... ;
424   net_db()

```

The above model is an idealisation. It assumes that the link and hub data represent a well-formed net. Included in this well-formedness are the following issues: (a) that all link or hub identifiers are communicated exactly once, (b) that all mereologies refer to defined parts, and (c) that all attribute values lie within an appropriate value range. If we were to cope with possible recording errors then we could, for example, extend the model as follows: (i) when a link or a hub recorder has completed its recording then it increments an initially zero counter (say at formula Item 423); (ii) before the net data base recycles it tests whether all recording sessions has ended and then proceeds to check the data base for well-formedness issues (a–b–c) (say at formula Item 424') ■

The above example illustrates the ‘interface’ phenomenon: In the formulas, for example, we show both manifest domain entities, viz., n, l, h etc., and abstract (required) software objects, viz., $(ui, me, attrs)$.

Data Refreshment:

One must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for selecting the updating of net, of hub or of link attribute names and their types and, for example, two for the respective update of hub and link attribute values. Interaction-prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions.

Shared Perdurants

We can expect that for every part in the domain that is shared with the machine and for which there is a corresponding behaviour of the domain there might be a corresponding process of the machine. If a projected, instantiated, ‘determined’ and possibly extended domain part is dynamic, then it is definitely a candidate for being shared and having an associated machine process. We now illustrate the concept of shared perdurants via the domain requirements extension example of Sect. 5.4.4, i.e. Example 5.10 Pages 180–187.

Example 5.15. . **Interface Requirements — Shared Behaviours: Road Pricing Calculator Behaviour:**

429 The road-pricing calculator alternates between offering to accept communication from
 430 either any vehicle
 431 or any toll-gate.

```

429 calc: ci:CI × (vis:VI-set × gis:GI-set) → RLF → TRM →
430   in {v_c_ch[ci,vi] | vi:VI • vi ∈ vis},
431   {g_c_ch[ci,gi] | gi:GI • gi ∈ gis} Unit
429 calc(ci,(vis,gis))(rlf)(trm) ≡
430   react_to_vehicles(ci,(vis,gis))(rlf)(trm)
429   □
431   react_to_gates(ci,(vis,gis))(rlf)(trm)
429   pre ci = ciℓ ∧ vis = visℓ ∧ gis = gisℓ

```

The calculator signature's $v_ch[ci,vi]$ and $g_ch[ci,gi]$ model the *embedded attribute sharing* between vehicles (their position), respectively gates (their vehicle identification) and the calculator road map [70, Sect. 4.1.1 and 4.5.2].

432 If the communication is from a vehicle inside the toll-road net
 433 then its toll-road net position, vp , is found from the road location function, rlf ,
 434 and the calculator resumes its work with the traffic map, trm , suitably updated,
 435 otherwise the calculator resumes its work with no changes.

```

430 react_to_vehicles(ci,(vis,gis),vplf)(trm) ≡
430   let (vi,(τ,lpos)) = [] {v_ch[ci,vi]?|vi:Vl•vi∈vis} in
432   if vi ∈ dom trm
433   then let vp = vplf(lpos) in
434     calc(ci,(vis,gis),vplf)(trm†[vi→trm^(τ,vp)]) end
435   else calc(ci,(vis,gis),vplf)(trm) end end

```

436 If the communication is from a gate,
 437 then that gate is either an entry gate or an exit gate;
 438 if it is an entry gate
 439 then the calculator resumes its work with the vehicle (that passed the entry gate) now recorded, afresh,
 in the traffic map, trm .
 440 Else it is an exit gate and
 441 the calculator concludes that the vehicle has ended its to-be-paid-for journey inside the toll-road net,
 and hence to be billed;
 442 then the calculator resumes its work with the vehicle now removed from the traffic map, trm .

```

431 react_to_gates(ci,(vis,gis),vplf)(trm) ≡
431   let (ee,(τ,(vi,li))) = [] {g_ch[ci,gi]?|gi:Gl•gi∈gis} in
437   case ee of
438     "Enter" →
439       calc(ci,(vis,gis),vplf)(trm∪[vi→((τ,SonL(li)))]),
440     "Exit" →
441       billing(vi,trm(vi)^(τ,SonL(li)));
442       calc(ci,(vis,gis),vplf)(trm\{vi}) end end

```

The above behaviour is the one for which we are to design software ■

5.5.2 Derived Requirements

Definition 35 Derived Perdurant: By a **derived perdurant** we shall understand a perdurant which is not shared with the domain, but which focus on exploiting facilities of the software or hardware of the machine ■

“Exploiting facilities of the software”, to us, means that requirements, imply the presence, in the machine, of concepts (i.e., hardware and/or software), and that it is these concepts that the **derived requirements** “rely” on. We illustrate all three forms of perdurant extensions: derived actions, derived events and derived behaviours.

Derived Actions

Definition 36 Derived Action: By a **derived action** we shall understand (a) a conceptual action (b) that calculates a usually non-Boolean valued property from, and possibly changes to (c) a machine behaviour state (d) as instigated by some actor ■

Example 5.16. . Domain Requirements. Derived Action: Tracing Vehicles: The example is based on the *Road Pricing Calculator Behaviour* of Example 5.15 on Page 193. The “external” actor, i.e., a user of the *Road Pricing Calculator* system wishes to trace specific vehicles “cruising” the toll-road. That user (a *Road Pricing Calculator* staff), issues a command to the *Road Pricing Calculator* system, with the identity of a vehicle not already being traced. As a result the *Road Pricing Calculator* system augments a possibly void trace of the timed toll-road positions of vehicles. We augment the definition of the *calculator* definition Items 429–442, Pages 193–194.

443 Traces are modeled by a pair of dynamic attributes:

a as a programmable attribute, $tra:TRA$, of the set of identifiers of vehicles being traced, and
b as a reactive attribute, $vdu:VDU^{18}$, that maps vehicle identifiers into time-stamped sequences of simple vehicle positions, i.e., as a subset of the $trm:TRM$ programmable attribute.

444 The actor-to-calculator *begin* or *end* trace command, $cmd:Cmd$, is modeled as an autonomous dynamic attribute of the *calculator*.

445 The *calculator* signature is furthermore augmented with the three attributes mentioned above.

446 The occurrence and handling of an actor trace command is modeled as a non-deterministic external choice and a *react_to_trace_cmd* behaviour.

447 The reactive attribute value ($attr_vdu_ch?$) is that subset of the traffic map (trm) which records just the time-stamped sequences of simple vehicle positions being traced (tra).

type

443a $TRA = VI\text{-set}$
443b $VDU = TRM$
444 $Cmd = BTr \mid ETr$
444 $BTr :: VI$
444 $ETr :: VI$

value

445 $calc: ci:CI \times (vis:VI\text{-set} \times gis:GI\text{-set}) \rightarrow RLF \rightarrow TRM \rightarrow TRA$
430,431 **in** $\{v_c_ch[ci,vi] \mid vi:VI \wedge vi \in vis\}$,
430,431 $\{g_c_ch[ci,gi] \mid gi:GI \wedge gi \in gis\}$,
446,447 $attr_cmd_ch, attr_vdu_ch$ **Unit**
429 $calc(ci, (vis, gis))(rlf)(trm)(tra) \equiv$
430 $react_to_vehicles(ci, (vis, gis),)(rlf)(trm)(tra)$
431 $\square react_to_gates(ci, (vis, gis))(rlf)(trm)(tra)$
446 $\square react_to_trace_cmd(ci, (vis, gis))(rlf)(trm)(tra)$
429 **pre** $ci = ci_{\mathcal{E}} \wedge vis = vis_{\mathcal{E}} \wedge gis = gis_{\mathcal{E}}$
447 **axiom** $\square attr_vdu_ch[ci]? = trm|tra$

The 446,447 $attr_cmd_ch, attr_vdu_ch$ of the *calculator* signature models the *calculator*’s external *command* and *visual display unit* attributes.

448 The *react_to_trace_cmd* alternative behaviour is either a “Begin” or an “End” request which identifies the affected vehicle.

¹⁸ VDU: visual display unit

449 If it is a "Begin" request
 450 and the identified vehicle is already being traced then we do not prescribe what to do !
 451 Else we resume the calculator behaviour, now recording that vehicle as being traced.
 452 If it is an "End" request
 453 and the identified vehicle is already being traced then we do not prescribe what to do !
 454 Else we resume the calculator behaviour, now recording that vehicle as no longer being traced.

```

448 react_to_trace_cmd(ci,(vis,gis))(vplf)(trm)(tra) ≡
448   case attr_cmd_ch[ci]? of
449,450,451   mkBTr(vi) → if vi ∈ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra ∪ {vi}) end
452,453,454   mkETr(vi) → if vi ∉ tra then chaos else calc(ci,(vis,gis))(vplf)(trm)(tra \ {vi}) end
448   end

```

The above behaviour, Items 429–454, is the one for which we are to design software ■

Example 5.16 exemplifies an action requirement as per definition 36: (a) the action is conceptual, it has no physical counterpart in the domain; (b) it calculates (447) a visual display (vdu); (c) the vdu value is based on a conceptual notion of traffic road maps (trm), an element of the calculator state; (d) the calculation is triggered by an actor (attr_cmd_ch).

Derived Events

Definition 37 Derived Event: By a **derived event** we shall understand (a) a conceptual event, (b) that calculates a property or some non-Boolean value (c) from a machine behaviour state change ■

Example 5.17. . Domain Requirements. Derived Event: Current Maximum Flow: The example is based on the *Road Pricing Calculator Behaviour* of Examples 5.16 and 5.15 on Page 193. By "the current maximum flow" we understand a time-stamped natural number, the number representing the highest number of vehicles which at the time-stamped moment cruised or now cruises around the toll-road net. We augment the definition of the calculator definition Items 429–454, Pages 193–196.

455 We augment the calculator signature with
 456 a time-stamped natural number valued dynamic programmable attribute, $(t:\mathbb{T}, max:Max)$.
 457 Whenever a vehicle enters the toll-road net, through one of its [entry] gates,
 a it is checked whether the resulting number of vehicles recorded in the *road traffic map* is higher than
 the hitherto *maximum* recorded number.
 b If so, that programmable attribute has its number element "upped" by one.
 c Otherwise not.
 458 No changes are to be made to the react_to_gates behaviour (Items 431–442 Page 194) when a vehicle exits
 the toll-road net.

type

456 MAX = $\mathbb{T} \times \text{NAT}$

value

```

445,455 calc: ci:CI × (vis:VI-set × gis:GI-set) → RLF → TRM → TRA → MAX
430,431   in {v_c_ch[ci,vi]|vi:VI•vi ∈ vis}, {g_c_ch[ci,gi]|gi:GI•gi ∈ gis}, attr_cmd_ch,attr_vdu_ch Unit
431 react_to_gates(ci,(vis,gis))(vplf)(trm)(tra)(t,m) ≡
431   let (ee,(τ,(vi,li))) = []{g_c_ch[ci,gi]|gi:GI•gi ∈ gis} in
437   case ee of
457     "Enter" →
457       calc(ci,(vis,gis))(vplf)(trm ∪ [vi → ((τ,SonL(li)))])(tra)(τ,if card dom trm=m then m+1 else m end),
458     "Exit" →
458       billing(vi,rm(vi)^(τ,SonL(li))); calc(ci,(vis,gis))(vplf)(trm \ {vi})(tra)(t,m) end
437   end

```


The above behaviour, Items 429 on Page 193 through 457c on the preceding page, is the one for which we are to design software ■

Example 5.17 exemplifies a derived event requirement as per Definition 37: (a) the event is conceptual, it has no physical counterpart in the domain; (b) it calculates (457b) the max value based on a conceptual notion of traffic road maps (trm), (c) which is an element of the calculator state.

No Derived Behaviours

There are no derived behaviours. The reason is as follows. Behaviours are associated with parts. A possibly ‘derived behaviour’ would entail the introduction of an ‘associated’ part. And if such a part made sense it should – in all likelihood – already have been either a proper domain part or become a domain extension. If the domain-to-requirements engineer insist on modeling some interface requirements as a process then we consider that a technical matter, a choice of abstraction.

5.5.3 Discussion

Derived Requirements

Formulation of derived actions or derived events usually involves technical terms not only from the domain but typically from such conceptual ‘domains’ as mathematics, economics, engineering or their visualisation. Derived requirements may, for some requirements developments, constitute “sizable” requirements compared to “all the other” requirements. For their analysis and prescription it makes good sense to first having developed “the other” requirements: domain, interface and machine requirements. The treatment of the present chapter does not offer special techniques and tools for the conception, &c., of derived requirements. Instead we refer to the seminal works of [116, 168, 254].

Introspective Requirements

Humans, including human users are, in this chapter, considered to never be part of the domain for which a requirements prescription is being developed. If it is necessary to involve humans in the domain description or the requirements prescription then their prescription is to reflect assumptions upon whose behaviour the machine rely. It is therefore that we, above, have stated, in passing, that we cannot accept requirements of the kind: “*the machine must be user friendly*”, because, in reality, it means “*the user must rely upon the machine being ‘friendly’*” whatever that may mean. We are not requirements prescribing humans, nor their sentiments !

5.6 Machine Requirements

Other than listing a sizable number of *machine requirement facets* we shall not cover machine requirements in this chapter. The reason for this is as follows. We find, cf. [32, Sect. 19.6], that when the individual machine requirements are expressed then references to domain phenomena are, in fact, abstract references, that is, they do not refer to the semantics of what they name. Hence *machine requirements* “fall” outside the scope of this chapter — with that scope being “*derivation*” of requirements from domain specifications with emphasis on derivation techniques that relate to various aspects of the domain.

(A) There are the *technology requirements* of (1) *performance* and (2) *dependability*. Within *dependability requirements* there are (a) *accessibility*, (b) *availability*, (c) *integrity*, (d) *reliability*, (e) *safety*, (f) *security* and (g) *robustness* requirements. A proper treatment of dependability requirements need a careful definition of such terms as *failure*, *error*, *fault*, and, from these *dependability*. (B) And there are the

development requirements of (i) *process*, (ii) *maintenance*, (iii) *platform*, (iv) *management* and (v) *documentation* requirements. Within *maintenance requirements* there are (ii.1) *adaptive*, (ii.2) *corrective*, (ii.3) *perfective*, (ii.4) *preventive*, and (ii.5) *extensional* requirements. Within *platform requirements* there are (iii.1) *development*, (iii.2) *execution*, (iii.3) *maintenance*, and (iii.4) *demonstration* platform requirements. We refer to [32, Sect. 19.6] for an early treatment of *machine requirements*.

5.7 Conclusion

Conventional requirements engineering considers the domain only rather implicitly. Requirements gathering ('acquisition') is not structured by any pre-existing knowledge of the domain, instead it is "structured" by a number of relevant techniques and tools [158, 254, 159] which, when applied, "fragment-by-fragment" "discovers" such elements of the domain that are immediately relevant to the requirements. The present chapter turns this requirements prescription process "up-side-down". Now the process is guided ("steered", "controlled") almost exclusively by the domain description which is assumed to be existing before the requirements development starts. In conventional requirements engineering many of the relevant techniques and tools can be said to take into account *sociological* and *psychological* facets of gathering the requirements and *linguistic* facets of expressing these requirements. That is, the focus is rather much on the *process*. In the present chapter's requirements "derivation" from domain descriptions the focus is all the time on the descriptions and prescriptions, in particular on their formal expressions and the "transformation" of these. That is (descriptions and) prescriptions are considered formal, *mathematical* objects. That is, the focus is rather much on the *objects*.



We conclude by briefly reviewing what has been achieved, present shortcomings & possible research challenges, and a few words on relations to "classical requirements engineering".

5.7.1 What has been Achieved ?

We have shown how to systematically "derive" initial aspects of requirements prescriptions from domain descriptions. The stages¹⁹ and steps²⁰ of this "derivation"²¹ are new. We claim that current requirements engineering approaches, although they may refer to a or the 'domain', are not really 'serious' about this: they do not describe the domain, and they do not base their techniques and tools on a reasoned understanding of the domain. In contrast we have identified, we claim, a logically motivated decomposition of requirements into three phases, cf. Footnote 19., of domain requirements into five steps, cf. Footnote 20 (Page 198), and of interface requirements, based on a concept of shared entities, tentatively into (α) shared endurants, (β) shared actions, (γ) shared events, and (δ) shared behaviours (with more research into the (α - δ) techniques needed).

5.7.2 Present Shortcomings and Research Challenges

We see three shortcomings: (1) The "derivation" techniques have yet to consider "extracting" requirements from *domain facet descriptions*. Only by including *domain facet descriptions* can we, in "deriving" *requirements prescriptions*, include failures of, for example, support technologies and humans, in the design of fault-tolerant software. (2) The "derivation" principles, techniques and tools should be given a formal treatment. (3) There is a serious need for relating the approach of the present chapter to that of the seminal text book of [254, Axel van Lamsweerde]. [254] is not being "replaced" by the present work. It tackles a different set of problems. We refer to the penultimate paragraph before the **Acknowledgment** closing.

¹⁹ (a) domain, (b) interface and (c) machine requirements

²⁰ For domain requirements: (i) projection, (ii) instantiation, (iii) determination, (iv) extension and (v) fitting; etc.

²¹ We use double quotation marks: "... " to indicate that the derivation is not automatable.

5.7.3 Comparison to “Classical” Requirements Engineering:

Except for a few, represented by two, we are not going to compare the contributions of the present chapter with published journal or conference papers on the subject of requirements engineering. The reason for this is the following. The present chapter, rather completely, we claim, reformulates requirements engineering, giving it a ‘foundation’, in *domain engineering*, and then developing *requirements engineering* from there, viewing requirements prescriptions as “derived” from domain descriptions. We do not see any of the papers, except those reviewed below [162] and [116], referring in any technical sense to ‘domains’ such as we understand them.

[162, Deriving Specifications for Systems That Are Connected to the Physical World]

The paper that comes closest to the present chapter in its serious treatment of the [problem] domain as a precursor for requirements development is that of [162, Jones, Hayes & Jackson]. A purpose of [162] (Sect. 1.1, Page 367, last §) is to see “how little can one say” (about the problem domain) when expressing assumptions about requirements. This is seen by [162] (earlier in the same paragraph) as in contrast to our form of domain modeling. [162] reveals assumptions about the domain when expressing *rely guarantees* in tight conjunction with expressing the *guarantee* (requirements). That is, analysing and expressing requirements, in [162], goes hand-in-hand with analysing and expressing fragments of the domain. The current chapter takes the view that since, as demonstrated in [70], it is possible to model sizable aspects of domains, then it would be interesting to study how one might “derive” — and which — requirements prescriptions from domain descriptions; and having demonstrated that (i.e., the “how much can be derived”) it seems of scientific interest to see how that new start (i.e., starting with a priori given domain descriptions or starting with first developing domain descriptions) can be combined with existing approaches, such as [162]. We do appreciate the “tight coupling” of *rely-guarantees* of [162]. But perhaps one loses understanding the domain due to its fragmented presentation. If the ‘relies’ are not outright, i.e., textually directly expressed in our domain descriptions, then they obviously must be provable properties of what our domain descriptions express. Our, i.e., the present, chapter — with its background in [70, Sect. 4.7] — develops — with a background in [157, M.A. Jackson] — a set of principles and techniques for the access of attributes. The “discovery” of the CM and SG channels of [162] and of the type of their messages, seems, compared to our approach, less systematic. Also, it is not clear how the [162] case study “scales” up to a larger domain. The *sluice gate* of [162] is but part of a large (‘irrigation’) system of reservoirs (water sources), canals, sluice gates and the fields (water sinks) to be irrigated. We obviously would delineate such a larger system and research & develop an appropriate, both informal, a narrative, and formal domain description for such a class of irrigation systems based on assumptions of precipitation and evaporation. Then the users’ requirements, in [162], that the sluice gate, over suitable time intervals, is open 20% of the time and otherwise closed, could now be expressed more pertinently, in terms of the fields being appropriately irrigated.

[116, Goal-directed Requirements Acquisition]

outlines an approach to requirements acquisition that starts with fragments of domain description. The domain description is captured in terms of predicates over *actors*, *actions*, *events*, *entities* and (their) *relations*. Our approach to domain modeling differs from that of [116] as follows: Agents, actions, entities and relations are, in [116], seen as specialisations of a concept of *objects*. The nearest analogy to relations, in [70], as well as in this chapter, is the signatures of perdurants. Our ‘agents’ relate to discrete endurants, i.e., parts, and are the behaviours that evolve around these parts: one agent per part! [116] otherwise include describing parts, relations between parts, actions and events much like [70] and this chapter does. [116] then introduces a notion of *goal*. A **goal**, in [116], is defined as “a nonoperational objective to be achieved by the desired system. Nonoperational means that the objective is not formulated in terms of objects and actions “available” to some agent of the system ■²²” [116] then goes on to exemplify goals. In this, the

²² We have reservations about this definition: Firstly, it is expressed in terms of some of the “things” it is not! (To us, not a very useful approach.) Secondly, we can imagine goals that are indeed formulated in terms of objects

current chapter, we are not considering *goals*, also a major theme of [254].²³ Typically the expression of goals of [116, 254], are “within” computer & computing science and involve the use of temporal logic.²⁴ “Constraints are operational objectives to be achieved by the desired (i.e., required) system, . . . , formulated in terms of objects and actions “available” to some agents of the system. . . . Goals are made operational through constraints. . . . A constraint operationalising a goal amounts to some abstract “implementation” of this goal” [116]. [116] then goes on to express goals and constraints operationalising these. [116] is a fascinating paper²⁵ as it shows how to build goals and constraints on domain description fragments.



These papers, [162] and [116], as well as the current chapter, together with such seminal monographs as [264, 192, 254], clearly shows that there are many diverse ways in which to achieve precise requirements prescriptions. The [264, 192] monographs primarily study the $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ specification and proof techniques from the point of view of the specific tools of their specification languages²⁶. Physics, as a natural science, and its many engineering ‘renditions’, are manifested in many separate sub-fields: Electricity, mechanics, statics, fluid dynamics — each with further sub-fields. It seems, to this author, that there is a need to study the [264, 192, 254] approaches and the approach taken in this chapter in the light of identifying sub-fields of requirements engineering. The title of the present chapter suggests one such sub-field.

5.8 Bibliographical Notes

I have thought about domain engineering for more than 20 years. But serious, focused writing only started to appear since [32, Part IV] — with [25, 22] being exceptions: [34] suggests a number of domain science and engineering research topics; [45] covers the concept of domain facets; [87] explores compositionality and Galois connections. [35, 86] show how to systematically, but, of course, not automatically, “derive” requirements prescriptions from domain descriptions; [50] takes the triptych software development as a basis for outlining principles for believable software management; [41, 58] presents a model for Stanisław Leśniewski’s [104] concept of mereology; [46, 51] present an extensive example and is otherwise a precursor for the present chapter; [53] presents, based on the TripTych view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators; [55] analyses the TripTych, especially its domain engineering approach, with respect to [175, 176, Maslow]’s and [198, Peterson]’s and [Seligman]’s notions of humanity: how can computing relate to notions of humanity; the first part of [61] is a precursor for [70] with the second part of [61] presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current chapter; and with [62] focus on domain safety criticality.

and actions ‘available’ to some agent of the system. For example, wrt. the ongoing library examples of [116], *the system shall automate the borrowing of books*, etcetera. Thirdly, we assume that by “‘available’ to some agent of the system” is meant that these agents, actions, entities, etc., are also required.

²³ An example of a goal — for the road pricing system — could be that of *shortening travel times of motorists, reducing gasoline consumption and air pollution, while recouping investments on toll-road construction*. We consider techniques for ensuring the above kind of goals “outside” the realm of computer & computing science but “inside” the realm of operations research (OR) — while securing that the OR models are commensurate with our domain models.

²⁴ In this chapter we do not exemplify goals, let alone the use of temporal logic. We cannot exemplify all aspects of domain description and requirements prescription, but, if we were, would then use the temporal logic of [264, The Duration Calculus].

²⁵ — that might, however, warrant a complete rewrite.

²⁶ The Duration Calculus [DC], respectively DC, Timed Automata and Z

Some Implications for Software

Demos, Simulators, Monitors and Controllers

A Divertimento of Ideas and Suggestions.

We¹ muse over the concepts of demos, simulators, monitors and controllers.

6.1 Introduction

We sketch some observations of the concepts of domain, requirements and modeling – where abstract interpretations of these models cover both a priori, a posteriori and real-time aspects of the domain as well as 1–1 (i.e., real-time), microscopic and macroscopic simulations, real-time monitoring and real-time monitoring & control of that domain. The reference frame for these concepts are domain models: carefully narrated and formally described domains. On the basis of a familiarising example² of a domain description, we survey more-or-less standard ideas of verifiable software developments and conjecture software product families of demos, simulators, monitors and monitors & controllers – but now these “standard ideas” are recast in the context of core requirements prescriptions being “derived” from domain descriptions.

A background setting for this chapter is the concern for (α) professionally developing the right software, i.e., software which satisfies users expectations, and (ω) software that is right: i.e., software which is correct with respect to user requirements and thus has no “bugs”, no “blue screens”. The present chapter must be seen on the background of a main line of experimental research around the topics of domain science & engineering and requirements engineering and their relation. For details I refer to [70, 76, 66].

“Confusing Demos”:

This author has had the doubtful honour, on his many visits to computer science and software engineering laboratories around the world, to be presented, by his colleagues’ aspiring PhD students, so-called demos of “systems” that they were investigating. There always was a tacit assumption, namely that the audience, i.e., me, knew, a priori, what the domain “behind” the “system” being “demo’ed” was. Certainly, if there was such an understanding, it was brutally demolished by the “demo” presentation. My questions, such as “*what are you demo’ing*” (etcetera) went unanswered. Instead, while we were waiting to see “something interesting” to be displayed on the computer screen we were witnessing frantic, sometimes failed, input of commands and data, “nervous” attempts with “mouse” clickings, etc. – before something intended was displayed. After a, usually 15 minute, grace period, it was time, luckily, to proceed to the next “demo”.

Aims & Objectives:

The aims of this chapter is to present (a) some ideas about software that either “demo”, simulate, monitor or monitor & control domains; (b) some ideas about “time scaling”: demo and simulation time versus

¹ This chapter is a slightly edited rendition of [52].

² – take that of Chapter 1

domain time; and (c) how these kinds of software relate. The (undoubtedly very naïve) objectives of the chapter is also to improve the kind of demo-presentations, alluded to above, so as to ensure that the basis for such demos is crystal clear from the very outset of research & development, i.e., that domains be well-described. The chapter, we think, tackles the issue of so-called ‘model-oriented (or model-based) software development’ from altogether different angles than usually promoted.

An Exploratory Chapter:

The chapter is exploratory. There will be no theorems and therefore there will be no proofs. We are presenting what might eventually emerge into (α) a theory of domains, i.e., a domain science [34, 87, 39, 51], and (β) a software development theory of domain engineering versus requirements engineering [50, 35, 40, 46].

The chapter is not a “standard” research chapter: it does not compare its claimed achievements with corresponding or related achievements of other researchers – simply because we do not claim “achievements” which have been reasonably well formalised. But we would suggest that you might find some of the ideas of the chapter (in Sect. 6.3) worthwhile. Hence the “divertimento” suffix to the chapter title.

Structure of Chapter:

The structure of the chapter is as follows. In Sect. 6.3 we then outline a series of interpretations of domain descriptions. These arise, when developed in an orderly, professional manner, from requirements prescriptions which are themselves orderly developed from the domain description³, cf. [66].

The essence of Sect. 6.3 is (i) the (albeit informal) presentation of such tightly related notions as *demos* (Sect. 6.3.1), *simulators* (Sect. 6.3.2), *monitors* (Sect. 6.3.3) and *monitors & controllers* (Sect. 6.3.3) (these notions can be formalised), and (ii) the conjectures on a product family of domain-based software developments (Sect. 6.3.5). A notion of *script-based simulation* extends demos and is the basis for monitor and controller developments and uses. The scripts used in our examples are related to time, but one can define non-temporal scripts – so the “carrying idea” of Sect. 6.3 extends to a widest variety of software. We claim that Sect. 6.3 thus brings these new ideas: a tightly related software engineering concept of *demo-simulator-monitor-controller* machines, and an extended notion of *reference models for requirements and specifications* [136].

6.2 Domain Descriptions

By a domain description we shall mean a combined narrative, that is, precise, but informal, and a formal description of the application domain **as it is**: no reference to any possible requirements let alone software that is desired for that domain. Thus a requirements prescription is a likewise combined precise, but informal, narrative, and a formal prescription of what we expect from a machine (hardware + software) that is to support endurants, actions, events and behaviours of a possibly business process re-engineered application domain. Requirements expresses a domain **as we would like to be**.

We further refer to the literature for examples: [23, *railways* (2000)], [24, *the ‘market’* (2000)], [40, *public government, IT security, hospitals* (2006) chapters 8–10], [35, *transport nets* (2008)] and [46, *pipelines* (2010)]. On the net you may find technical reports covering “larger” domain descriptions. “Older” publications on the concept of domain descriptions are [46, 51, 41, 87, 35, 34, 45] all summarised in [70, 76, 66].

Domain descriptions do not necessarily describe computable objects. They relate to the described domain in a way similar to the way in which mathematical descriptions of physical phenomena stand to “the physical world”.

6.3 Interpretations

In this main section of the chapter we present a number of interpretations of rôles of domain descriptions.

³ We do not show such orderly “derivations” but outline their basics in Sect. 6.3.4.

6.3.1 What Is a Domain-based Demo?

A *domain-based demo* is a software system which “present” endurants and perdurants⁴: actions, events and behaviours of a domain. The “presentation” abstracts these phenomena and their related concepts in various computer generated forms: visual, acoustic, etc.

Examples

There are two main examples. One was given in Chapter 1. The other is summarised below. It is from Chapter 5 on “deriving requirements prescriptions from domain descriptions”. The summary follows.

The domain description of Sect. 5.2 outlines an abstract concept of transport nets (of hubs [street intersections, train stations, harbours, airports] and links [road segments, rail tracks, shipping lanes, air-lanes]), their development, traffic [of vehicles, trains, ships and aircraft], etc. We shall assume such a transport domain description below.

Endurants are, for example, presented as follows: (a) transport nets by two dimensional (2D) road, railway or air traffic maps, (b) hubs and links by highlighting parts of 2D maps and by related photos – and their unique identifiers by labeling hubs and links, (c) routes by highlighting sequences of paths (hubs and links) on a 2D map, (d) buses by photographs and by dots at hubs or on links of a 2D map, and (e) bus timetables by, well, indeed, by showing a 2D bus timetable.

Actions are, for example, presented as follows: (f) The insertion or removal of a hub or a link by showing “instantaneous” triplets of “before”, “during” and “after” animation sequences. (g) The start or end of a bus ride by showing flashing animations of the appearance, respectively the flashing disappearance of a bus (dot) at the origin, respectively the destination bus stops.

Events are, for example, presented as follows: (h) A mudslide [or fire in a road tunnel, or collapse of a bridge] along a (road) link by showing an animation of part of a (road) map with an instantaneous sequence of (α) the present link, (β) a gap somewhere on the link, (γ) and the appearance of two (“symbolic”) hubs “on either side of the gap”. (i) The congestion of road traffic “grinding to a halt” at, for example, a hub, by showing an animation of part of a (road) map with an instantaneous sequence of the massive accumulation of vehicle dots moving (instantaneously) from two or more links into a hub.

Behaviours are, for example, presented as follows: (k) A bus tour: from its start, on time, or “thereabouts”, from its bus stop of origin, via (all) intermediate stops, with or without delays or advances in times of arrivals and departures, to the bus stop of destination (l) The composite behaviour of “all bus tours”, meeting or missing connection times, with sporadic delays, with cancellation of some bus tours, etc. – by showing the sequence of states of all the buses on the net.

We say that behaviours ((j)–(l)) are *script-based* in that they (try to) satisfy a bus timetable ((e)).

Towards a Theory of Visualisation and Acoustic Manifestation

The above examples shall serve to highlight the general problem of visualisation and acoustic manifestation. Just as we need sciences of visualising scientific data and of diagrammatic logics, so we need *more serious studies of visualisation and acoustic manifestation — so amply, but, this author thinks, inconsistently demonstrated by current uses of interactive computing media.*

6.3.2 Simulations

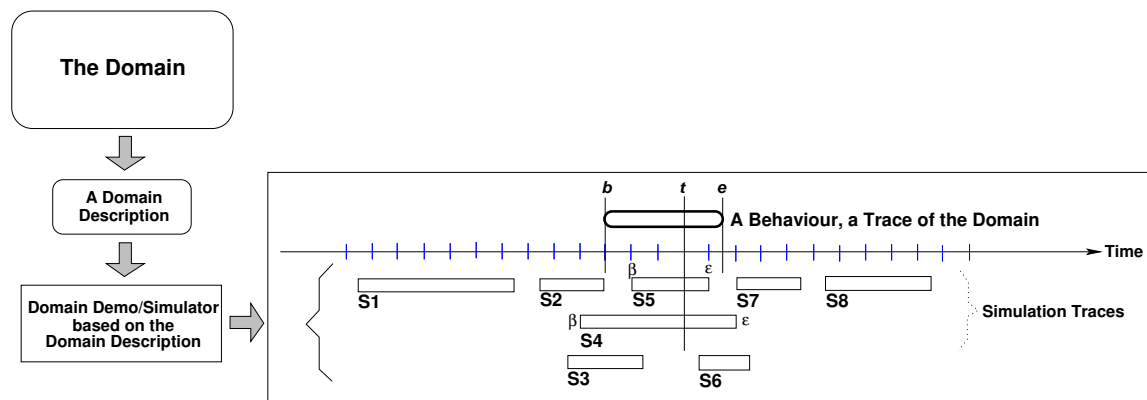
“Simulation is the imitation of some real thing, state of affairs, or process; the act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system” [Wikipedia] for the purposes of testing some hypotheses usually stated in terms of the model being simulated and pairs of statistical data and expected outcomes.

⁴ The concepts of ‘endurants’ and ‘perdurants’ were defined in [70].

Explication of Figure 6.1

Figure 6.1 attempts to indicate four things: (i) Left top: the rounded edge rectangle labeled “The Domain” alludes to some specific domain (“out there”). (ii) Left middle: the small rounded rectangle labeled “A Domain Description” alludes to some document which narrates and formalises a description of “the domain”. (iii) Left bottom: the medium sized rectangle labeled “A Domain Demo based on the Domain Description” (for short “Demo”) alludes to a software system that, in some sense (to be made clear later) “simulates” “The Domain.” (iv) Right: the large rectangle (a) shows a horizontal time axis which basically “divides” that large rectangle into two parts: (b) Above the time axis the “fat” rounded edge rectangle alludes to the time-wise behaviour, a *domain trace*, of “The Domain” (i.e., the actual, the real, domain). (c) Below the time axis there are eight “thin” rectangles. These are labels S1, S2, S3, S4, S5, S6, S7 and S8. (d) Each of these denote a “run”, i.e., a time-stamped “execution”, a *program trace*, of the “Demo”. Their “relationship” to the time axis is this: their execution takes place in the real time as related to that of “The Domain” behaviour.

A *trace* (whether a domain or a program execution trace) is a time-stamped sequence of states: domain states, respectively demo, simulator, monitor and monitor & control states.



Legend: ➡ A development; S1, S2, S3, S4, S5, S6, S7, S8: "runs" of the Domain Simulation

Fig. 6.1. Simulations

From Fig. 6.1 and the above explication we can conclude that “executions” S4 and S5 each share exactly one time point, *t*, at which “The Domain” and “The Simulation” “share” time, that is, the time-stamped execution S4 and S5 reflect a “Simulation” state which at time *t* should reflect (some abstraction of) “The Domain” state.

Only if the domain behaviour (i.e., trace) fully “surrounds” that of the simulation trace, or, vice-versa (cf. Fig. 6.1[S4,S5]), is there a “shared” time. Only if the ‘begin’ and ‘end’ times of the domain behaviour are identical to the ‘start’ and ‘finish’ times of the simulation trace, is there an infinity of shared 1–1 times. Only then do we speak of a real-time simulation.

In Fig 6.2 on the next page we show “the same” “Domain Behaviour” (three times) and a (1) simulation, a (2) monitoring and a (3) monitoring & control, all of whose ‘begin/start’ (*b/β*) and ‘end/finish’ (*e/ε*) times coincide. In such cases the “Demo/Simulation” takes place in real-time throughout the ‘begin...end’ interval.

Let *β* and *ε* be the ‘start’ and ‘finish’ times of either S4 or S5. Then the relationship between *t, β, ε, b* and *e* is $\frac{t-b}{e-t} = \frac{t-\beta}{\epsilon-t}$ — which leads to a second degree polynomial in *t* which can then be solved in the usual, high school manner.

Script-based Simulation

A script-based simulation is the behaviour, i.e., an execution, of, basically, a demo which, step-by-step, follows a script: that is a prescription for highlighting endurants, actions, events and behaviours.

Script-based simulations where the script embodies a notion of time, like a bus timetable, and unlike a route, can be thought of as the execution of a demos where “chunks” of demo operations take place in accordance with “chunks”⁵ of script prescriptions. The latter (i.e., the script prescriptions) can be said to represent simulated (i.e., domain) time in contrast to “actual computer” time. The actual times in which the script-based simulation takes place relate to domain times as shown in Simulations S1 to S8 in Fig. 6.1 and in Fig. 6.2(1–3). Traces Fig. 6.2(1–3) and S8 Fig. 6.1 are said to be *real-time*: there is a one-to-one mapping between computer time and domain time. S1 and S4 Fig. 6.1 are said to be *microscopic*: disjoint computer time intervals map into distinct domain times. S2, S3, S5, S6 and S7 are said to be *macroscopic*: disjoint domain time intervals map into distinct computer times.

In order to concretise the above “vague” statements let us take the example of simulating bus traffic as based on a bus timetable script. A simulation scenario could be as follows. Initially, not relating to any domain time, the simulation “demos” a net, available buses and a bus timetable. The person(s) who are requesting the simulation are asked to decide on the ratio of the domain time interval to simulation time interval. If the ratio is 1 a real-time simulation has been requested. If the ratio is less than 1 a microscopic simulation has been requested. If the ratio is larger than 1 a macroscopic simulation has been requested. A chosen ratio of, say 48 to 1 means that a 24 hour bus traffic is to be simulated in 30 minutes of elapsed simulation time. Then the person(s) who are requesting the simulation are asked to decide on the starting domain time, say 6:00am, and the domain time interval of simulation, say 4 hours – in which case the simulation of bus traffic from 6am till 10am is to be shown in 5 minutes (300 seconds) of elapsed simulation time. The person(s) who are requesting the simulation are then asked to decide on the “*sampling times*” or “*time intervals*”: If ‘*sampling times*’ 6:00 am, 6:30 am, 7:00 am, 8:00 am, 9:00 am, 9:30 am and 10:00 am are chosen, then the simulation is stopped at corresponding simulation times: 0 sec., 37.5 sec., 75 sec., 150 sec., 225 sec., 262.5 sec. and 300 sec. The simulation then shows the state of selected endurants and actions at these domain times. If ‘*sampling time interval*’ is chosen and is set to every 5 min., then the simulation shows the state of selected endurants and actions at corresponding domain times. The simulation is resumed when the person(s) who are requesting the simulation so indicates, say by a “resume” icon click. The time interval between adjacent simulation stops and resumptions contribute with 0 time to elapsed simulation time – which in this case was set to 5 minutes. Finally the requestor provides some statistical data such as numbers of potential and actual bus passengers, etc.

Then two clocks are started: a domain time clock and a simulation time clock. The simulation proceeds as driven by, in this case, the bus time table. To include “unforeseen” events, such as the wreckage of a bus (which is then unable to complete a bus tour), we allow any number of such events to be randomly scheduled. Actually scheduled events “interrupts” the “programmed” simulation and leads to thus unscheduled stops (and resumptions) where the unscheduled stop now focuses on showing the event.

The Development Arrow

The arrow, \Rightarrow , between a pair of boxes (of Fig. 6.1 on Page 208) denote a step of development: (i) from the domain box to the domain description box, \Downarrow , it denotes the development of a domain description based on studies and analyses of the domain; (ii) from the domain description box to the domain demo box, \Downarrow , it denotes the development of a software system — where that development assumes an intermediate requirements box which has not been show; (iii) from the domain demo box to either of a simulation traces, \Rightarrow , it denotes the development of a simulator as the related demo software system, again depending on whichever special requirements have been put to the simulator.

⁵ We deliberately leave the notion of chunk vague so as to allow as wide an spectrum of simulations.

6.3.3 Monitoring & Control

Figure 6.2 shows three different kinds of uses of software systems (where (2) [Monitoring] and (3) [Monitoring & Control] represent further) developments from the demo or simulation software system mentioned in Sect. 6.3.1 and Sect. 6.3.2 on the preceding page. We have added some (three) horizontal and

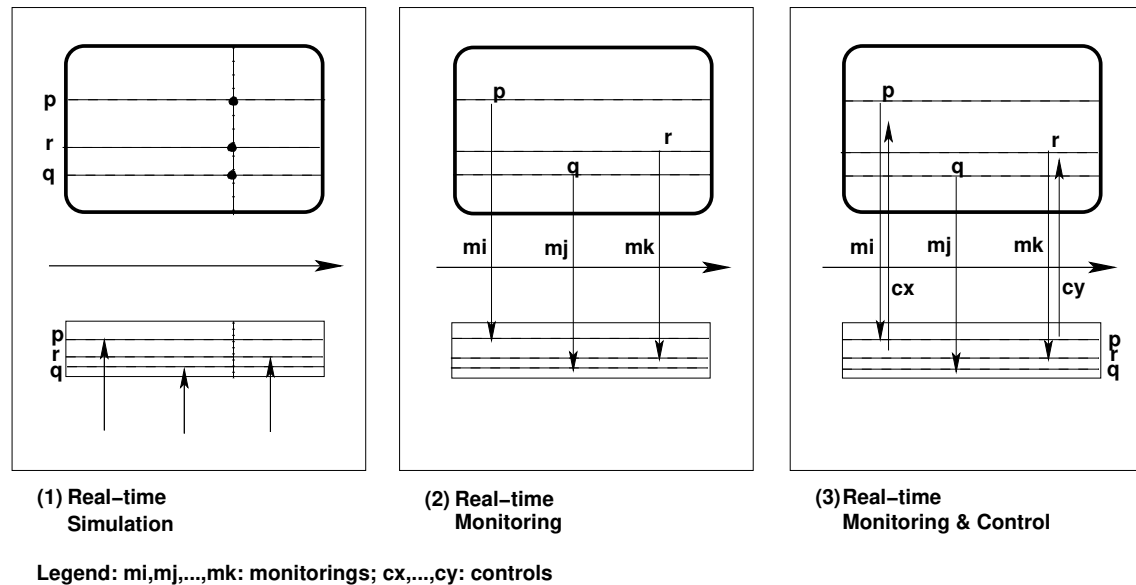


Fig. 6.2. Simulation, Monitoring and Monitoring & Control

labeled (p, q and r) lines to Fig. 6.2(1,2,3) (with respect to the traces of Fig. 6.1 on Page 208). They each denote a trace of an endurant, an action or an event, that is, they are traces of values of these phenomena or concepts. A (named) endurant value entails a description of the endurant, whether atomic ('hub', 'link', 'bus timetable') or composite ('net', 'set of hubs', etc.): of its unique identity, its mereology and a selection of its attributes. A (named) action value could, for example, be the pair of the before and after states of the action and some description of the function ('insertion of a link', 'start of a bus tour') involved in the action. A (named) event value could, for example, be a pair of the before and after states of the endurants causing, respectively being effected by the event and some description of the predicate ('mudslide', 'break-down of a bus') involved in the event. A cross section, such as designated by the vertical lines (one for the domain trace, one for the "corresponding" program trace) of Fig. 6.2 on the preceding page(1) denotes a state: a domain, respectively a program state.

Figure 6.2(1) attempts to show a real-time demo or simulation for the chosen domain. Figure 6.2(2) purports to show the deployment of real-time software for monitoring (chosen aspects of) the chosen domain. Figure 6.2(3) purports to show the deployment of real-time software for monitoring as well as controlling (chosen aspects of) the chosen domain.

Monitoring

By *domain monitoring* we mean "to be aware of the state of a domain", its endurants, actions, events and behaviour. Domain monitoring is thus a process, typically within a distributed system for collecting and storing state data. In this process "observation" points — i.e., endurants, actions and where events may occur — are identified in the domain, cf. points p, q and r of Fig. 6.2. Sensors are inserted at these points. The

“downward” pointing vertical arrows of Figs. 6.2(2–3), from “the domain behaviour” to the “monitoring” and the “monitoring & control” traces express communication of what has been sensed (measured, photographed, etc.) [as directed by and] as input data (etc.) to these monitors. The monitor (being “executed”) may store these “sensings” for future analysis.

Control

By *domain control* we mean “the ability to change the value” of endurants and the course of actions and hence behaviours, including prevention of events of the domain. Domain control is thus based on domain monitoring. Actuators are inserted in the domain “at or near” monitoring points or at points related to these, viz. points *p* and *r* of Fig. 6.2 on the facing page(3). The “upward” pointing vertical arrows of Fig. 6.2 on the preceding page(3), from the “monitoring & control” traces to the “domain behaviour” express communication, to the domain, of what has been computed by the controller as a proper control reaction in response to the monitoring.

6.3.4 Machine Development

Machines

By a *machine* we shall understand a combination of hardware and software. For demos and simulators the machine is “mostly” software with the hardware typically being graphic display units with tactile instruments. For monitors the “main” machine, besides the hardware and software of demos and simulators, additionally includes *sensors* distributed throughout the domain and the technological machine means of *communicating* monitored signals from the sensors to the “main” machine and the processing of these signals by the main machine. For monitors & controllers the machine, besides the monitor machine, further includes actuators placed in the domain and the machine means of computing and communicating control signals to the actuators.

Requirements Development

Essential parts of Requirements to a Machine can be systematically “derived” from a Domain description. These essential parts are the *domain requirements* and the *interface requirements*. Domain requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the domain. These technical terms cover only phenomena and concepts (endurants, actions, events and behaviours) of the domain. Some domain requirements are *projected*, *instantiated*, made more *deterministic* and *extended*⁶. We bring examples that are taken from Sect. 5.2, cf. Sect. 6.3.1 on Page 207 of the present chapter. (a) By *domain projection* we mean a sub-setting of the domain description: parts are left out which the requirements stake-holders, collaborating with the requirements engineer, decide is of no relevance to the requirements. For our example it could be that our domain description had contained models of road net attributes such as “the wear & tear” of road surfaces, the length of links, states of hubs and links (that is, [dis]allowable directions of traffic through hubs and along links), etc. Projection might then omit these attributes. (b) By *domain instantiation* we mean a specialisation of endurants, actions, events and behaviours, refining them from abstract simple entities to more concrete such, etc. For our example it could be that we only model freeways or only model road-pricing nets – or any one or more other aspects. (c) By *domain determination* we mean that of making the domain description cum domain requirements prescription less non-deterministic, i.e., more deterministic (or even the other way around!). For our example it could be that we had domain-described states of street intersections as not controlled by traffic signals – where the determination is now that of introducing an abstract notion of traffic signals which allow only certain states (of red, yellow and green). (d) By *domain extension* we basically mean that of extending the

⁶ We omit consideration of *fitting*.

domain with phenomena and concepts that were not feasible without information technology. For our examples we could extend the domain with bus mounted GPS gadgets that record and communicate (to, say a central bus traffic computer) the more-or-less exact positions of buses – thereby enabling the observation of bus traffic. Interface requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms both of the domain and of the machine. These technical terms thus cover shared phenomena and concepts, that is, phenomena and concepts of the domain which are, in some sense, also (to be) represented by the machine. Interface requirements represent (i) the initialisation and “on-the-fly” update of machine endurants on the basis of *shared* domain endurants; (ii) the interaction between the machine and the domain while the machine is carrying out a (previous domain) action; (iii) machine responses, if any, to domain events — or domain responses, if any, to machine events cum “outputs”; and (iv) machine monitoring and machine control of domain phenomena. Each of these four (i–iv) interface requirement facets themselves involve projection, instantiation, determination, extension and fitting. Machine requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the machine. (An example is: visual display units.)

6.3.5 Verifiable Software Development

An Example Set of Conjectures

We illustrate some conjectures.

(A) From a domain, \mathcal{D} , one can develop a domain description \mathbb{D} . \mathbb{D} cannot be [formally] verified. It can be [informally] validated “against” \mathcal{D} . Individual properties, $\mathbb{P}_{\mathbb{D}}$, of the domain description \mathbb{D} and hence, purportedly, of the domain, \mathcal{D} , can be expressed and possibly proved $\mathbb{D} \models \mathbb{P}_{\mathbb{D}}$ and these may be validated to be properties of \mathcal{D} by observations in (or of) that domain.

(B) From a domain description, \mathbb{D} , one can develop requirements, \mathbb{R}_{DE} , for, and from \mathbb{R}_{DE} one can develop a domain demo machine specification \mathbb{M}_{DE} such that $\mathbb{D}, \mathbb{M}_{\text{DE}} \models \mathbb{R}_{\text{DE}}$. The formula $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ can be read as follows: in order to prove that the Machine satisfies the Requirements, assumptions about the Domain must often be made explicit in steps of the proof.

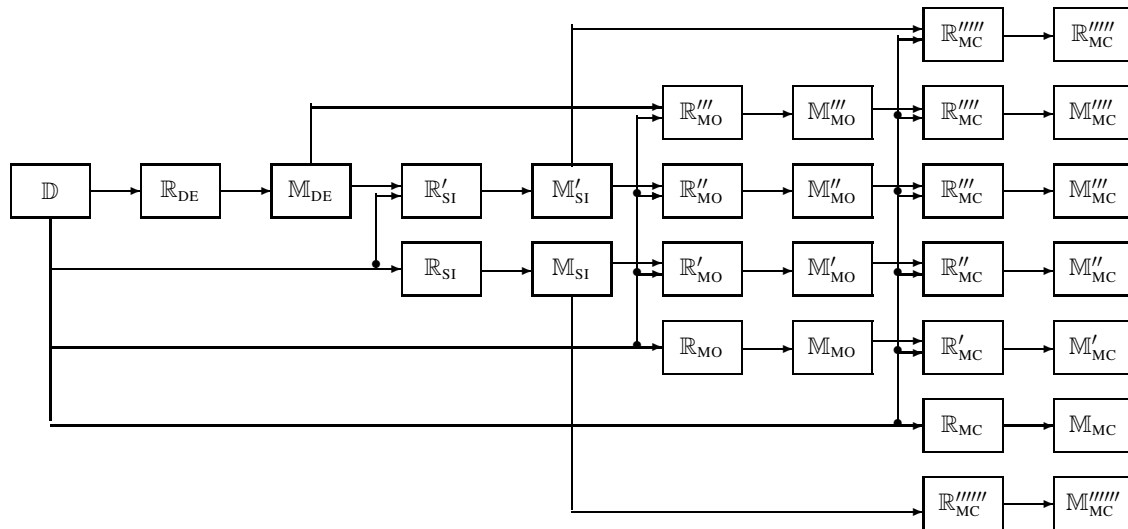
(C) From a domain description, \mathbb{D} , and a domain demo machine specification, \mathbb{S}_{DE} , one can develop requirements, \mathbb{R}_{SI} , for, and from such a \mathbb{R}_{SI} one can develop a domain simulator machine specification \mathbb{M}_{SI} such that $(\mathbb{D}; \mathbb{M}_{\text{DE}}), \mathbb{M}_{\text{SI}} \models \mathbb{R}_{\text{SI}}$. We have “lumped” $(\mathbb{D}; \mathbb{M}_{\text{DE}})$ as the two constitute the extended domain for which we, in this case of development, suggest the next stage requirements and machine development to take place.

(D) From a domain description, \mathbb{D} , and a domain simulator machine specification, \mathbb{M}_{SI} , one can develop requirements, \mathbb{R}_{MO} , for, and from such a \mathbb{R}_{MO} one can develop a domain monitor machine specification \mathbb{M}_{MO} such that $(\mathbb{D}; \mathbb{M}_{\text{SI}}), \mathbb{M}_{\text{MO}} \models \mathbb{R}_{\text{MO}}$.

(E) From a domain description, \mathbb{D} , and a domain monitor machine specification, \mathbb{M}_{MO} , one can develop requirements, \mathbb{R}_{MC} , for, and from such a \mathbb{R}_{MC} one can develop a domain monitor & controller machine specification \mathbb{M}_{MC} such that $(\mathbb{D}; \mathbb{M}_{\text{MO}}), \mathbb{M}_{\text{MC}} \models \mathbb{R}_{\text{MC}}$.

Chains of Verifiable Developments

The above illustrated just one chain (A–E) of developments. There are others. All are shown in Fig. 6.3. Figure 6.3 can also be interpreted as prescribing a widest possible range of machine cum software products [99, 201] for a given domain. One domain may give rise to many different kinds of DEMO machines, Simulators, MONitors and Monitor & Controllers (the unprimed versions of the \mathbb{M}_{T} machines (where T ranges over DE, SI, MO, MC)). For each of these there are similarly, “exponentially” many variants of successor machines (the primed versions of the \mathbb{M}_{T} machines). What does it mean that a machine is a primed version? Well, here it means, for example, that \mathbb{M}'_{SI} embodies facets of the demo machine \mathbb{M}_{DE} , and that \mathbb{M}''_{MC} embodies facets of the demo machine \mathbb{M}_{DE} , of the simulator \mathbb{M}'_{SI} , and the monitor \mathbb{M}''_{MO} . Whether such requirements are desirable is left to product customers and their software providers [99, 201] to decide.



Legend: \mathbb{D} domain, \mathbb{R} requirements, \mathbb{M} machine
 DE: DEMO, SI: SIMULATOR, MO: MONITOR, MC: MONITOR & CONTROLLER

Fig. 6.3. Chains of Verifiable Developments

6.4 Conclusion

Our divertimento is almost over. It is time to conclude.

6.4.1 Discussion

The $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ ('correctness' of) development relation appears to have been first indicated in the Computational Logic Inc. Stack [17, 134] and the EU ESPRIT ProCoS [19, 20] projects; [136] presents this same idea with a purpose much like ours, but with more technical discussions.

The term 'domain engineering' appears to have at least two meanings: the one used here [34, 45] and one [143, 123, 101] emerging out of the Software Engineering Institute at CMU where it is also called *product line engineering*⁷. Our meaning, is, in a sense, more narrow, but then it seems to also be more highly specialised (with detailed description and formalisation principles and techniques). Fig. 6.3 on the previous page illustrates, in capsule form, what we think is the CMU/SEI meaning. The relationship between, say Fig. 6.3 and *model-based software development* seems obvious but need be explored. An extensive discussion of the term 'domain', as it appears in the software engineering literature is found in [70, Sect. 5.3].

What Have We Achieved

We have characterised a spectrum of strongly domain-related as well as strongly inter-related (cf. Fig. 6.3) software product families: *demos*, *simulators*, *monitors* and *monitor & controllers*. We have indicated varieties of these: simulators based on demos, monitors based on simulators, monitor & controllers based on monitors, in fact any of the latter ones in the software product family list as based on any of the earlier ones. We have sketched temporal relations between simulation traces and domain behaviours: *a priori*, *a posteriori*, *macroscopic* and *microscopic*, and we have identified the real-time cases which lead on to monitors and monitor & controllers.

⁷ http://en.wikipedia.org/wiki/Domain_engineering.

What Have We Not Achieved — Some Conjectures

We have not characterised the software product family relations other than by the $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ and $(\mathbb{D}; \mathbb{M}_{XYZ}), \mathbb{M} \models \mathbb{R}$ clauses. That is, we should like to prove conjectured type theoretic inclusion relations like:

$$\wp(\llbracket \mathcal{M}_{x_{\text{mod ext.}}} \rrbracket) \supseteq \wp(\llbracket \mathcal{M}'_{x_{\text{mod ext.}}} \rrbracket), \quad \wp(\llbracket \mathcal{M}'_{x_{\text{mod ext.}}} \rrbracket) \supseteq \wp(\llbracket \mathcal{M}''_{x_{\text{mod ext.}}} \rrbracket)$$

where x and Y range appropriately, where $\llbracket \mathcal{M} \rrbracket$ expresses the meaning of \mathcal{M} , where $\wp(\llbracket \mathcal{M} \rrbracket)$ denote the space of all machine meanings and where $\wp(\llbracket \mathcal{M}_{x_{\text{mod ext.}}} \rrbracket)$ is intended to denote that space modulo (“free of”) the y facet (here *ext.*, for extension).

That is, it is conjectured that the set of more specialised, i.e., n primed, machines of kind x is type theoretically “contained” in the set of m primed (unprimed) x machines ($0 \leq m < n$).

There are undoubtedly many such interesting relations between the DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER machines, unprimed and primed.

What Should We Do Next

This chapter has the subtitle: *A Divertimento of Ideas and Suggestions*. It is not a proper theoretical chapter. It tries to throw some light on families and varieties of software, i.e., their relations. It focuses, in particular, on so-called DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER software and their relation to the “originating” domain, i.e., that in which such software is to serve, and hence that which is being *extended* by such software, cf. the compounded ‘domain’ $(\mathbb{D}; \mathbb{M}_i)$ of in $(\mathbb{D}; \mathbb{M}_i), \mathbb{M}_j \models \mathbb{D}$. These notions should be studied formally. All of these notions: requirements projection, instantiation, determination and extension can be formalised; and the specification language, in the form used here (without CSP processes, [148] has a formal semantics and a proof system — so the various notions of development, $(\mathbb{D}; \mathbb{M}_i), \mathbb{M}_j \models \mathbb{R}$ and $\wp(\mathbb{M})$ can be formalised.

Part **IV**

A Note

Sorts, Types, Intents

In earlier chapters we introduced the notion of discrete endurants, both natural and artefactual, being parts and characterised classes of these as **sorts**. Parts were then analysed with respect to internal qualities such as unique identifiers, mereologies and attributes and these were characterised in terms of **types**. In Chapter 8 we show how Kai Sørlander’s philosophy [238, 239, 245] justifies our ontology of entities not on empirical grounds, but on philosophical grounds – and we brought forward the notion of **intentional pull** mentioned only briefly in [80]. In [75] we further analysed certain attribute types in terms of the *SI: The International System of Units*. In this chapter we shall examine some aspects of sorts, types and intents not covered earlier.

7.1 Introduction

By a **domain** we shall understand a **rationaly describable** segment of a **human assisted** reality, i.e., of the world, its **physical parts: natural** [“God-given”] and **artefactual** [“man-made”], and **living species: plants** and **animals** including, notably, **humans**. These entities are **endurants** (“still”), as well as **perdurants** (“alive”). Emphasis is placed on **“human-assistedness”**, that is, there is *at least one (man-made) artifact* and, therefore, that **humans** are a primary cause for change of endurant **states** as well as perdurant **behaviours**.

7.1.1 Entities, Endurants and Perdurants

Entity

By an **entity** we shall understand a **phenomenon**, i.e., something that can be *observed*: touched by humans, or that can be *conceived* as an *abstraction* of an entity; alternatively, a phenomenon is an entity, *if it exists, it is “being”*, *it is that which makes a “thing” what it is: essence, essential nature* [170, Vol. I, pg. 665] ■ **Examples:** A train, a train ride, an aircraft, a flight ■

Endurant

By an **endurant** we shall understand an entity that can be observed, or conceived and described, as a “complete thing” at no matter which given snapshot of time; alternatively an entity is endurant if it is capable of *enduring*, that is *persist*, “*hold out*” [170, Vol. I, pg. 656]. Were we to “freeze” time we would still be able to observe the entire endurant ■ **Examples:** A road, an automobile, a human driver ■

Perdurant

By a **perdurant** we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time. Were we to freeze time we would only see or touch a fragment of the perdurant, alternatively an entity is perdurant if it endures continuously, over time, persists, lasting [170, Vol. II, pg. 1552] ■ **Examples:** A train ride, an aircraft flight ■

7.1.2 Discrete and Continuous Endurants**Discrete Endurant**

By a **discrete endurant** we shall understand an endurant which is separate, individual or distinct in form or concept ■ **Examples:** A pipeline and its individual units: pipes, valves, pumps, forks, etc. ■

Continuous Endurants: Non-solids

By a **continuous endurant** (a **non-solid**) we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern ■ **Examples:** Water, oil, gas, compressed air, etc. A container, which we consider a discrete endurant, may contain a non-solid, like a gas pipeline unit may contain gas ■

7.1.3 A Domain Ontology

Figure 7.1 on the facing page graphs an essence of the domain ontology of entities, endurants, perdurants, etc., as these concepts were covered in [80]. Sections 7.1.1 – 7.1.2 covered some aspects of the first three layers, from the top, of that domain ontology. Following [80], as also justified, on grounds of philosophy, by [79], we shall claim that the manifest world, i.e., the *physical* and *living endurants*, can be analysed with respect to their observable, i.e., viewable and touchable, i.e., **external qualities**, respectively their measurable, i.e., **internal qualities**. The external qualities are summarised in **sorts**. Values of sorts, i.e., *physical* and *living endurants* [we shall omit treatment of *structures* in this paper], can be summarised in three (*internal quality*) categories: *unique identifiers*, *mereologies*, and *attributes*. These *internal qualities* are summarised by **types**¹.

• • •

We shall, in this paper, make a pragmatic distinction between sorts and types. Sorts will be used to characterise observable endurants. Types will be used to characterise sorts ! Intents are then [something] associated with man-made endurants.

7.2 Sorts

By a **sort** we shall generally mean a named set of endurants which we shall later further characterise.

7.2.1 Physical Parts, Living Species and Structures

With discrete endurants we associate sorts.

¹ The **RAISE** [132] **Specification Language**. **RSL** [131], as we use it in this paper, does not distinguish between *sorts* and *types*.

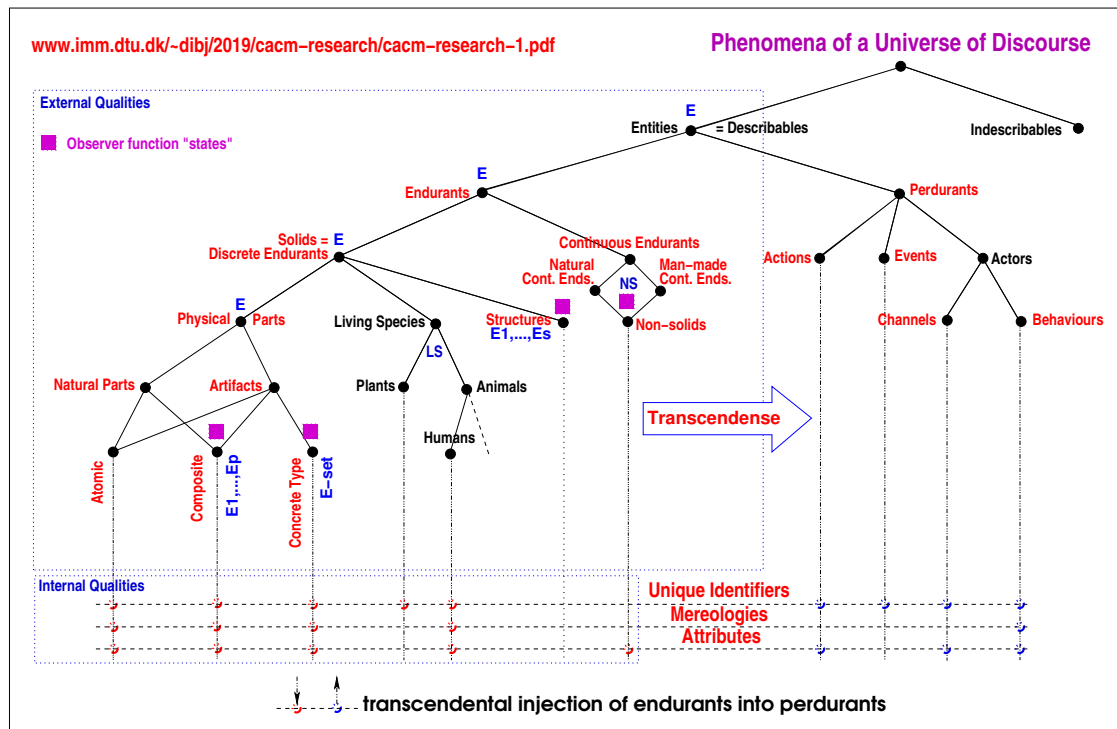


Fig. 7.1. A Domain Ontology

Physical Parts

By a *physical part* we shall understand a discrete endurant existing in time and subject to laws of physics, including the *causality principle* and *gravitational pull*² ■ Classes of “similar” physical parts are given names and these we shall refer to as sort names. Our investigation into sorts, types and intents will focus on physical, in particular artefactual parts.

Living Species

By a *living species* we shall understand a discrete endurant, subject to laws of physics, and additionally subject to *causality of purpose*. Living species must have some *form they can be developed to reach*; which they must be *causally determined to maintain*. This *development and maintenance* must further in an *exchange of matter with an environment*. It must be possible that living species occur in one of two forms: one form which is characterised by *development, form and exchange*; another form which, **additionally**, can be characterised by the *ability to purposeful movement* The first we call **plants**, the second we call **animals** ■ We shall not, in this paper further deal with living species

Structures

By a **structure** we shall understand a discrete endurant which the domain engineer chooses to describe as consisting of one or more endurants, whether discrete or continuous, but to **not** endow with **internal qualities**: unique identifiers, mereology or attributes ■ We shall not, in this paper further deal with the concept of structures.

² This characterisation is the result of our study of relations between philosophy and computing science, notably influenced by Kai Sørlander’s Philosophy. We refer to our research report [79].

7.2.2 Natural Parts and Artefacts

Physical parts are either *natural parts*, or are *artefacts*, i.e. man-made parts, which possess **internal qualities**: **unique identification**, **mereology**, and one or more **attributes** ■ For more on internal qualities, see Sect. 7.3.2.

Natural Parts

Natural parts are in *space* and *time*; are subject to the *laws of physics*, and also subject to the *principle of causality* and *gravitational pull* ■ **Examples**: an island, a mountain, a river, a lake, a granite rock, a gold lode ■

Artefacts

By an **artifact** we shall understand a *man-made physical part* **Examples**: road nets, road intersections (**hubs**), **links** (roads between adjacent hubs); automobiles ■

7.2.3 Various Forms of Physical Parts

We now arrive at the point where **sorts** come into play. Natural parts are either **atomic**, or **composite**, and artefactual parts are either of **atomic** sort, or of **composite** sort, or of **set sort**.

Atomic Parts

Atomic Parts are those which, in a given context, are deemed to *not* consist of meaningful, separately observable proper *sub-parts*. A **sub-part** is a *part* ■ **Examples**: a hub, a link, a pipe, a valve, a wheel, an engine, a door, a window ■

Composite Parts

Composite Parts are those which, in a given context, are deemed to *indeed* consist of meaningful, separately observable proper *sub-parts* ■ **Examples**: an automobile, a road net, a pipeline ■

Set Sort Parts

Set Sort Parts are simplifications of components. A set sort part is a set of parts of the *same* sort. The domain analyser cum describer chooses to **indeed** endow components with **mereology** ■ **Examples**: Road nets are considered compositions of two parts. a hub aggregate and a link aggregate. The hub aggregate is a set sort part and consists of a set of hubs; the link aggregate is a set sort part and consists of a set of links ■ Set sort parts are pragmatic constructions.

7.2.4 Analysis and Description Prompts

Implicit in the “story” of Sects. 7.2.1–7.2.3 are the following **analysis prompts**:

- `is_entity`
- `is_endurant`
- `is_perdurant`
- `is_discrete`
- `is_continuous`
- `is_phys._part`
- `is_liv._species`
- `is_structure`
- `is_natural_part`
- `is_artefact`
- `is_atomic`
- `is_composite` ■
- `is_components` ■
- `is_set_sort` ■
- et cetera (■)

The ■ boxes imply analysis states where the following **description prompts** are applicable:

- `observe_composite_sorts` • `observe_component_sorts` • `observe_set_sort`

respectively (– et cetera). The description observers can be formalised:

type: `observe_composite_sorts`: $E \rightarrow \text{Text}$

Narrative:

- s. narrative text on sorts E_1, \dots, E_n
- o. narrative text on observers $\text{obs}_{E_1}, \dots, \text{obs}_{E_n}$
- p. narrative text on proof obligation: \mathcal{P}

Formalisation:

- s. **type** E_1, \dots, E_n
- o. **value** $\text{obs}_{E_1}: E \rightarrow E_1, \dots, \text{obs}_{E_n}: E \rightarrow E_n$
- p. **proof obligation** $\mathcal{P}: \forall i: \{1..n\} \cdot \text{is}_{E_i}(e) \equiv \bigwedge \{ \sim E_j(e) \mid j: [1..n] \setminus \{i\} \mid j: [1..n] \}$

In any specific domain analysis & description the analyser cum describer chooses which subset of composite sorts to analyse & describe. That is: any one domain model emphasises certain aspects and leaves out many “other” aspects.

type: `observe_set_sort`: $E \rightarrow \text{Text}$

Narratives:

- s. narrative text on sort P
- o. narrative text on observer obs_{Ps}

Formalisation:

- s. **type** P, $Ps = \text{P-set}$
- o. **value** $\text{obs}_{Ps}: E \rightarrow \text{P-set}$

Typically P may be a sort expression: $P_1|P_2|\dots|P_n$ where P_i are sorts.

7.2.5 An Example: Road Transport

External Qualities

459 The road transport system consists of two aggregates: a road net and automobiles.
 460 The road net consists of aggregates of atomic hubs (street intersections) and atomic links (streets).
 461 Hub aggregates are sets of hubs and link aggregates are sets of links.
 462 Automobile aggregates are sets of automobiles.

<p>type</p> <p>459. RTS, RN, AA</p> <p>value</p> <p>459. $\text{obs}_{RN}: \text{RTS} \rightarrow \text{RN}$</p> <p>459. $\text{obs}_{AA}: \text{RTS} \rightarrow \text{AA}$</p> <p>type</p> <p>460. AH, AL</p> <p>value</p> <p>460. $\text{obs}_{AH}: \text{RN} \rightarrow \text{AH}$</p> <p>460. $\text{obs}_{AL}: \text{RN} \rightarrow \text{AL}$</p>	<p>type</p> <p>461. $\text{Hs} = \text{H-set}, \text{H}$</p> <p>461. $\text{Ls} = \text{L-set}, \text{L}$</p> <p>value</p> <p>461. $\text{obs}_{Hs}: \text{AH} \rightarrow \text{Hs}$</p> <p>461. $\text{obs}_{Ls}: \text{AL} \rightarrow \text{Ls}$</p> <p>type</p> <p>462. $\text{As} = \text{A-set}, \text{A}$</p> <p>value</p> <p>462. $\text{obs}_{As}: \text{AA} \rightarrow \text{As}$</p>
--	---

7.3 Types

By a **type** we shall generally mean a named set of values which we, at the instance of introducing the type name, either define as an atomic **token** type, or as a *concrete* type. By an atomic token type we mean a set of further undefined atomic values. By a concrete type we shall here mean either a **set** of values of type **T**, i.e., **T-set**, or a **list** of values of type **T**, i.e., **T***, or a **map** from values of type **A** to values of type **B**, i.e., $A \rightarrow B$, or a **Cartesian product** (a “record”, a “structure”) of **A**, **B**, ..., **C** typed values, i.e., $A \times B \times \dots \times C$. A type can also be a **union** type, that is, the set union of distinct types **A**, **B**, ..., **C**, i.e., $A|B|\dots|C$. **Tokens**, **Integers**, **Natural Numbers**, **Reals**, **Characters**, **POINT**, **T**, and **TI**, for the latter three, see Sect. 7.3.1, are base, or “atomic”, types. Concrete types of common programming languages include **arrays** (**vectors**, **matrices**, **tensors**, etc.) and **records**. Eventually it all ends up in atomic (i.e., base) types.

7.3.1 Space and Time

Space and time “fall” somewhat outside a “standard view” of types. We do not prescribe, really, a type **space**. It is just there. We shall present a view of time different from those of [97, 252, 128].

Space

There is an abstract notion of (definite) **SPACE**(s) of further un-analysable points; and there is a notion of **POINTS** in **SPACE**. Space is not an attribute of endurants. Space is just there. So we do not define an observer, **observe_space**.

463 A point observer, **observe_POINT**, is a function which applies to a[ny] specific “location” on a physical endurant, *e*, and yields a point, $\ell : \text{POINT}$.

value

463 **obs_POINT**: $E \rightarrow \text{POINT}$

Time

By a **definite time** we shall understand an abstract representation of time such as for example year, day, hour, minute, second, et cetera ■ We shall not be concerned with any representation of time. That is, we leave it to the domain analyser cum describer to choose an own representation [128]. Similarly we shall not be concerned with any representation of time intervals.³

464 So there is an abstract type *Time*,

465 and an abstract type **TI**: *TimeInterval*.

466 There is no *Time* origin, but there is a “zero” **TI**me interval.

467 One can add (subtract) a time interval to (from) a time and obtain a time.

468 One can add and subtract two time intervals and obtain a time interval – with subtraction respecting that the subtrahend is smaller than or equal to the minuend.

469 One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.

470 One can multiply a time interval with a real and obtain a time interval.

471 One can compare two times and two time intervals.

³ – but point out, that although a definite time interval may be referred to by number of years, number of days (less than 365), number of hours (less than 24), number of minutes (less than 60) number of seconds (less than 60), et cetera, this is not a time, but a time interval.


```

type
464  $\mathbb{T}$ 
465  $\mathbb{TI}$ 
value
466  $\mathbf{0}:\mathbb{TI}$ 
467  $+, -: \mathbb{T} \times \mathbb{TI} \rightarrow \mathbb{T}$ 
468  $+, -: \mathbb{TI} \times \mathbb{TI} \xrightarrow{\sim} \mathbb{TI}$ 
469  $-: \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{TI}$ 
470  $*: \mathbb{TI} \times \mathbf{Real} \rightarrow \mathbb{TI}$ 
471  $<, \leq, =, \neq, \geq, >: \mathbb{T} \times \mathbb{T} \rightarrow \mathbf{Bool}$ 
471  $<, \leq, =, \neq, \geq, >: \mathbb{TI} \times \mathbb{TI} \rightarrow \mathbf{Bool}$ 
axiom
467  $\forall t:\mathbb{T} \cdot t+\mathbf{0} = t$ 

```

472 We define the signature of the meta-physical time observer.

```

value
472 record_TIME(): Unit  $\rightarrow \mathbb{T}$ 

```

The time recorder applies to nothing and yields a time. **record_TIME**() can only occur in action, event and behavioural descriptions.

7.3.2 Internal Qualities

The internal qualities of endurants may include: unique identifiers, for physical parts and living species; mereologies, for atomic, composite, set sort and human parts; and attributes, for physical parts and living species.

Unique Identifiers

Every discrete endurant, $e:E$, is unique and can hence be ascribed a **unique identifier**; that identifier can be ascertained by applying the **uid_E** observer function to e .

Mereologies

Mereology is the study of parts and the wholes they form ■ We shall interpret the **mereology of a part**, p , here as as the topological and/or conceptual relations between that part and other parts. Typically we can express the mereology of p , i.e., **mereo_P**(p), in terms of the sets of unique identifiers of the other parts with which p is related. Generally, we can express that relationship as a triplet: **mereo_P**(p)=(ips,iops,ops) where ips is the set of unique identifiers of those parts “from” which p “receives input”, whatever ‘input’ means (!); iops is the set of unique identifiers of those parts “with” which p mutually “shares” properties, whatever ‘shares’ means (!); ops is the set of unique identifiers of those parts “to” which p “delivers output”, whatever ‘output’ means (!); and where the three sets are mutually disjoint.

Attributes

Part attributes form more “free-wheeling” sets of **internal qualities** than those of unique identifiers and mereologies.

Non-solids are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), are tangible, in the sense of being spatial [or being abstractions, i.e., concepts, of spatial endurants], attributes are intangible: cannot normally be touched, but can be objectively measured. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective “attributes”: feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics.

Thus, to any part and non-solid, e , we can associate one or more attributes A_1, A_2, \dots, A_m , where A_i is an attribute type name and where **attr_A_i**(e) is the corresponding attribute observer.

Internal Quality Observers

We can summarise the observers for internal qualities while otherwise referring to [80] for details.

type observe_unique_identifier: $P \rightarrow \text{Text}$
<p>Narratives:</p> <ul style="list-style-type: none"> i. text on unique identifier: UI o. text on unique identifier observer: uid_E <p>Formalisation:</p> <ul style="list-style-type: none"> i. type UI o. value uid_E: $E \rightarrow \text{UI}$

type observe_mereology: $P \rightarrow \text{Text}$
<p>Narratives:</p> <ul style="list-style-type: none"> m. text on mereology: M o. text on mereology observer: mereo_E <p>Formalisation:</p> <ul style="list-style-type: none"> m. type $M = \mathcal{E}(\text{UI}_a, \dots, \text{UI}_c)$ o. value mereo_E: $E \rightarrow M$

For the expression of $\mathcal{E}(\text{UI}_a, \dots, \text{UI}_c)$ the domain analyser cum describer need not take into consideration any concern for possible “data structure efficiency” as we are not prescribing software requirements let alone specifying a software design. The choice of $\mathcal{E}(\text{UI}_a, \dots, \text{UI}_c)$, that is, of the mereology of any one sort \mathcal{E} , depends on the aspects of the domain that its analyser cum describer wishes to study. That is, “one and the same domain” may give rise to different models each emphasizing their aspects.

type observe_attributes: $P \rightarrow \text{Text}$
<p>Narratives:</p> <ul style="list-style-type: none"> a. texts on attributes: A_i, \dots, A_k o. texts on attribute observers: attr_$A_i, \dots, \text{attr}_{A_k}$ <p>Formalisation:</p> <ul style="list-style-type: none"> a. type $A_i [= \mathcal{A}_i], \dots, A_k [= \mathcal{A}_k]$ o. value obs_A_i: $E \rightarrow A_i, \dots, \text{obs}_{A_k}$: $E \rightarrow A_k$ <p>where $[= \mathcal{A}_j]$ refer to an optional type expression.</p>

In the expression of \mathcal{A}_j the domain analyser cum describer need not take into consideration any concern for possible data structure efficiency as we are not prescribing software requirements let alone specifying a software design.

One and “seemingly” the same domain may give rise to different analyses & descriptions. Each of these emphasize different aspects. **Example: Road Net:** In one model of a road net emphasis may be on automobile traffic (aiming, eventually, at a road pricing system). In another model of “the same” road net emphasis may be on the topological layout (aiming, eventually, at its construction). In yet a third model “over” a road net emphasis may be on traffic control ■ For each such “road net” model the domain analyser cum describer selects different overlapping sets of attributes.

Three Categories of Attributes

We can identify three kinds of attributes: (i) physics, (i) artefactual and (i) intentional.

7.3.3 Physics Attributes

Typically, when physicists write computer programs, intended for calculating physics behaviours, they “lump” all of these into the **type Real**, thereby hiding some important physics ‘dimensions’. In this section we shall review that which is missing !

The subject of physical dimensions in programming languages is rather decisively treated in David Kennedy’s 1996 PhD Thesis [165] — so there really is no point in trying to cast new light on this subject other than to remind the reader of what these physical dimensions are all about.

SI: The International System of Quantities

In physics we operate on values of attributes of manifest, i.e., physical phenomena. The type of some of these attributes are recorded in well known tables, cf. Tables 7.1–7.3. Table 7.1 shows the base units of physics.

Base quantity	Name	Type
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd

Table 7.1. Base SI Units

Table 7.2 shows the units of physics derived from the base units. Table 7.3 on the next page shows further

Name	Type	Derived Quantity	Derived Type
radian	rad	angle	m/m
steradian	sr	solid angle	$m^2 \times m^{-2}$
Hertz	Hz	frequency	s^{-1}
newton	N	force, weight	$kg \times m \times s^{-2}$
pascal	Pa	pressure, stress	N/m^2
joule	J	energy, work, heat	$N \times m$
watt	W	power, radiant flux	J/s
coulomb	C	electric charge	$s \times A$
volt	V	electromotive force	$W/A (kg \times m^2 \times s^{-3} \times A^{-1})$
farad	F	capacitance	$C/V (kg^{-1} \times m^{-2} \times s^4 \times A^2)$
ohm	Ω	electrical resistance	$V/A (kg \times m^2 \times s^3 \times A^2)$
siemens	S	electrical conductance	$A/V (kg^{-1} \times m^{-2} \times s^3 \times A^2)$
weber	Wb	magnetic flux	$V \times s (kg \times m^2 \times s^{-2} \times A^{-1})$
tesla	T	magnetic flux density	$Wb/m^2 (kg \times s^2 \times A^{-1})$
henry	H	inductance	$Wb/A (kg \times m^2 \times s^{-2} \times A^2)$
degree Celsius	$^{\circ}C$	temp. rel. to 273.15 K	K
lumen	lm	luminous flux	$cd \times sr (cd)$
lux	lx	illuminance	$lm/m^2 (m^2 \times cd)$

Table 7.2. Derived SI Units

units of physics derived from the base units. The upper half of Table 7.5 shows standard prefixes for SI units of measure and the lower half of Table 7.5 shows fractions of SI units.

Name	Explanation	Derived Type
area	square meter	m^2
volume	cubic meter	m^3
speed, velocity	meter per second	m/s
acceleration	meter per second squared	m/s^2
wave number	reciprocal meter	m^{-1}
mass density	kilogram per cubic meter	kg/m^3
specific volume	cubic meter per kilogram	m^3/kg
current density	ampere per square meter	A/m^2
magnetic field strength	ampere per meter	A/m
substance concentration	mole per cubic meter	mol/m^3
luminance	candela per square meter	cd/m^2
mass fraction	kilogram per kilogram	$kg/kg = 1$

Table 7.3. Further SI Units

Prefix name	deca	hecto	kilo	mega	giga	
Prefix symbol	da	h	k	M	G	
Factor	10^0	10^1	10^2	10^3	10^6	10^9
Prefix name	tera	peta	exa	zetta	yotta	
Prefix symbol	T	P	E	Z	Y	
Factor	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}	

Table 7.4. Standard Prefixes for SI Units of Measure

Prefix name	deci	centi	milli	micro	nano	
Prefix symbol	d	c	m	μ	n	
Factor	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}
Prefix name	pico	femto	atto	zepto	yocto	
Prefix symbol	p	f	a	z	y	
Factor	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}	

Table 7.5. Fractions

• • •

The point in bringing this material is that when modelling, i.e., describing domains we must be extremely careful in not falling into the trap of modelling physics types, etc., as we do in programming – by simple **Reals**. We claim, without evidence, that many trivial programming mistakes are due to confusions between especially derived SI units, fractions and prefixes.

Units are Atomic

A volt, $kg \times m^2 \times s^{-3} \times A^{-1}$, see Table 7.2, is atomic. It is not a composite structure of mass, length, time, and electric current – in some intricate relationship.

Example 1: Physics Attributes

Hub attributes:		type
473	number of lanes, surface, etc.;	473. NoL, SUR, ...
Link attributes:		value
474	number of lanes, surface. etc.	473. attr_NoL:H→NoL 473. attr_SUR:H→SUR, ...
Automobile attributes:		value
475	Power, Fuel (Gasoline, Diesel, Electric, ...), Size, ...	474. attr_NoL:L→NoL 474. attr_SUR:L→SUR, ...
type		value
475.	BHp = $\text{Nat:kg} \times \text{m}^{-2} \times \text{s}^{-3}$	475. attr_BHp: A→BHp
475.	Fuel	475. attr_Fuel: A→Fuel
475.	Length = Nat:cm	475. attr_Length: A→Length
475.	Width = Nat:cm	475. attr_Width: A→Width
475.	Height = Nat:cm	475. attr_Height: A→Height
476.	Vel = $\text{Real:m} \times \text{s}^{-1}$	476. attr_Vel: A→Vel
476.	Acc = $\text{Real:m} \times \text{s}^{-2}$	476. attr_Acc: A→Acc

•••

Physical attributes may ascribe mass and volume to endurants. But they do not reveal the substance, i.e., the material from which the endurant is made. That is done by chemical attributes.

Chemical Elements

The *mole*, mol, substance is about chemical molecules. A mole contains exactly $6.02214076 \times 10^{23}$ (the Avogadro number) constituent particles, usually atoms, molecules, or ions – of the elements, cf. 'The Periodic Table', en.wikipedia.org/wiki/Periodic_table, cf. Fig. 7.2. Any specific molecule is then a compound of two or more elements, for example, calciumphosphat: $\text{Ca}_3(\text{PO}_4)_2$.

Moles bring substance to endurants. The physics attributes may ascribe weight and volume to endurants, but they do not explain what it is that gives weight, i.e., fills out the volume.

7.3.4 Artefactual Attributes

Examples of Artefactual Attributes

We exemplify some artefactual attributes.

- **Designs.** Artefacts are man-made endurants. Hence “exhibit” a design. My three dimensional villa has floor plans, etc. The artefact attribute: ‘*design*’ can thus be presented by the architect’s or the construction engineer’s CAD/CAM drawings.
- **States** of an artefact, such as, for example, a road intersection (or railway track) traffic signal; and
- **Currency**, e.g., Kr, \$, £, €, ¥, et cetera, used as an attribute⁴, say the cost of a train ticket.

⁴ One could also consider a [10 €] bank note to be an artefact, i.e., a part.

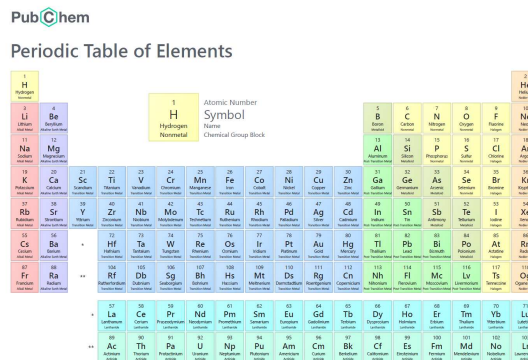


Fig. 7.2. Periodic Table

- Artefactual Dimensions.** Let the domain be that of industrial production whose attributes could then be: production: units produced per year, Units/Year; growth: increase in units produced per year, Units×Year⁻²; productivity: production per staff, Units×Year⁻¹×Staff⁻¹ — where the base for units and staff are natural numbers.

Document Artefactual Attributes

Let us consider *documents* as artefactual parts. Typical document attributes are: (i) kind of document: book, report, pamphlet, letter and ticket, (ii) publication date, (iii) number of pages, (iv) author/publisher and (v) possible colophon information. *All of these attributes are non-physics quantities.*

Road Net Artefactual Attributes

Hub attributes:

477 state: set of pairs of link identifiers from, respectively to which automobiles may traverse the hub;
 478 state space: set of all possible hub states.

type	value
477. $H\Sigma = (LI \times LI)\text{-set}$	477. $\text{attr}_{H\Sigma}: H \rightarrow H\Sigma$
478. $H\Omega = H\Sigma\text{-set}$	478. $\text{attr}_{H\Omega}: H \rightarrow H\Omega$

Link attributes:

479 state: set of 0, 1, 2 or 3 pairs of adjacent hub identifiers, the link is closed, open in one direction (closed in the opposite), open in the other direction, or open in both directions; and
 480 state space: set of all possible link states.

type	value
479. $L\Sigma = (LI \times LI)\text{-set}$	479. $\text{attr}_{L\Sigma}: L \rightarrow L\Sigma$
480. $L\Omega = L\Sigma\text{-set}$	480. $\text{attr}_{L\Omega}: L \rightarrow L\Omega$

7.4 Intents

7.4.1 Expressing Intents

Artefacts are made with an **intent**: one or more purposes for which the parts are to serve. Usually intents involve two or more part sorts.

Examples of Intents

- **Road Transport**: roads are “made to accommodate” automobiles, and automobiles are “made to drive” on roads ■
- **Credit Card System**: credit cards are for “payment of purchased merchandise”, and retailers are “there to sell merchandise” ■

7.4.2 Intent Modelling

We do not here suggest a formal way of expressing intents. That is, we do not formalise “made to accommodate”, “made to drive”, et cetera ! Intents, instead, are expressed as **intentional pulls**, and these are then expressed in terms of “intent-related” attributes.

Examples of Intent-related Attributes

The intent-related attributes are not based on physical evidence, but on what we can, but do not necessarily speak about.

Example: Intentional Attributes

Road Transport:

- 481 Hub traversal history: the recording of which automobiles traversed a hub at which time.
 482 Link traversal history: the recording of which automobiles traversed a link at which time.
 483 Automobile history: the recording of which hubs and links were traversed at which time.

type	value
481. $\text{HHist} = \text{AI} \xrightarrow{-\mathfrak{h}} \mathbb{T}\text{-set}$	481. $\text{attr_HHist}: \text{H} \rightarrow \text{HHist}$
482. $\text{LHist} = \text{AI} \xrightarrow{-\mathfrak{h}} \mathbb{T}\text{-set}$	482. $\text{attr_LHist}: \text{L} \rightarrow \text{LHist}$
483. $\text{AHist} = (\text{HI} \text{LI}) \xrightarrow{-\mathfrak{h}} \mathbb{T}\text{-set}$	483. $\text{attr_AHist}: \text{A} \rightarrow \text{AHist}$

All three history attributes are subject to constraints: the automobile, hub and link identifiers must be of automobiles, hubs and links of a (i.e., the) road net; the same automobile cannot be at two or more hubs and/or links at any one time (481–482) and the timed visits must be commensurate with the road net; et cetera.

Credit Card System:

- 484 Credit card histories X “records” Y^5 , by time, the shop and the merchandise bought.
 485 Shop histories “record”, by time, the credit card and the merchandise sold.

type	value
484. $\text{CHist} = \mathbb{T} \xrightarrow{-\mathfrak{h}} (\text{SI} \times \text{MI})$	484. $\text{attr_CHist}: \text{C} \rightarrow \text{CHist}$
485. $\text{SHist} = \mathbb{T} \xrightarrow{-\mathfrak{h}} (\text{CI} \times \text{MI})$	485. $\text{attr_SHist}: \text{S} \rightarrow \text{SHist}$

The two history attributes are subject to constraints: the shop and credit card identifiers must be of shops and credit cards of the credit card system; and the merchandise identifier must be of a merchandise of the identified shop.

7.4.3 Intentional Pull

The term ‘intentional pull’ was first introduced in [80].

496 to [re-]obtain ahists
as no automobile can be in any two or more places at any one time.

value

494. $\text{inv_ahists}: \text{AHists} \rightarrow \text{HLHists}$
 494. $\text{inv_ahists}(\text{ahists}) \equiv$
 494. $[\text{hli} \mapsto [\text{ai}' \mapsto (\text{ahists}(\text{ai}'))(\text{hli})] \text{ai}' : \text{AI} \cdot \text{ai}' \in \text{dom ahists} \wedge \text{hli} \in \text{dom ahists}(\text{ai}')]]$
 494. $[\text{ai} : \text{AI}, \text{hli} : (\text{HI} | \text{LI}) \cdot \text{ai} \in \text{dom ahists} \wedge \text{hli} \in \text{dom ahists}(\text{ai})]$

assertion:

496. $\forall \text{ahists} : \text{AHists} \cdot \text{inv_hlhists}(\text{inv_ahists}(\text{ahists})) = \text{ahists}$

where:

495. $\text{inv_hlhists}: \text{HLHists} \rightarrow \text{AHists}$
 495. $\text{inv_hlhists}(\text{hlhists}) \equiv$ left to the reader

Examples of Intentional Pulls

Road Transport:

497 If an automobile history records that an automobile was at a hub or on a link at some time, then that hub, respectively link, history records that that automobile was there at that time, and vice versa — and only that.

intentional pull:

497. $\square \forall \text{rtn} : \text{RTN}, \text{hs} : \text{Hs}, \text{ls} : \text{Ls}, \text{as} : \text{As}$
 497. $\text{hs} = \text{obs_Hs}(\text{obs_AH}(\text{rtn})) \wedge \text{ls} = \text{obs_Ls}(\text{obs_AL}(\text{rtn})) \wedge \text{as} = \text{obs_As}(\text{obs_AA}(\text{rtn}))$
 497. $\wedge \text{let ahists} = \text{xtr_AHists}(\text{as}), \text{hlhists} = \text{xtr_HHists}(\text{hs}) \cup \text{xtr_LHists}(\text{ls}) \text{ in}$
 497. $\text{inv_AHists}(\text{ahists}) = \text{hlhists} \text{ end}$

Credit Card System:

498 If a credit card history records that an purchase was made at a shop of some merchandise and at
 499 some time,
 500 then that shop's history records that that such a purchase was made there at that time,
 501 and vice versa — and only that.

We leave the formalisation to the reader ■

7.5 Actions, Events, Behaviours

By a **transcendental deduction** [80] we shall interpret discrete endurants as behaviours. Behaviours are sets of sequences of actions, events and behaviours. Behaviours communicate, for example, by means of CSP channels and output/input commands [148]

7.5.1 Actions

Actions are functions which purposefully are initiated by behaviours and potentially changes a state. Actions apply to behaviour arguments and yield updates to these.

⁶ Note the subtle use of free and bound variables in the map comprehension expressions.

7.5.2 Events

Events are functions which surreptitiously “occur” to behaviours, typically instigated by “some outside”, and usually changes a state. Events updates behaviour arguments. Events can be expressed as CSP [148]inputs.

7.5.3 Behaviours

To every part we shall, in principle, associate a behaviour. The behaviour is unique identified by the unique identifier of the part. The behaviour communicates with such other parts as are identified by the mereology of the parts. The behaviour otherwise depends on arguments: the unique part identifier, the part mereology, the part attributes separated into the static attributes, i.e., those with constant values, the programmable attributes, and the remaining dynamic attributes. The programmable attributes are those whose values are set by the behaviour, i.e., its actions.

7.5.4 Summary

The “miracle” of transcendental deduction is fundamental to domain analysis & description. It “ties” sorts: their external and internal qualities strongly to the dynamics of domains. Details on transcendental deductions, actions, events and behaviours are given in [80].

7.6 Conclusion

The sort, type and intent concepts of the domain analysis & description method covered in [80] has been studied in further detail. Although, as also illustrated by Fig. 8.1 on Page 244, the method includes the analysis of natural and living species, it is primarily aimed at artefacts and domains dominated by such. We refer to [85] for a dozen or so examples of medium-scale domain analysis & description case studies. You will see from those examples that they are all rather frugal with respect to ascribing attributes. That is: An endurant may have very many attributes, but in any one domain description in which it is present the analyser cum describer may have chosen to “abstract some out (!)”, that is, to not consider some — often very many of these — of these attributes.

7.6.1 Sort versus Types

“**Sorts are not recursive !**” That is, parts of sort S do not contain proper sub-parts of same sort S .

Pragmatics

In this paper we have used the terms ‘sorts’ and ‘types’ as follows. **Sorts** are used to describe external qualities of endurants: whether discrete or continuous (solids or non-solids), whether physical parts, living species or structures, whether natural parts or artefacts, and whether atomic, composite, components or set sorts. **Types** are used to describe internal qualities of endurants: unique identifiers, mereologies, and attributes.

Syntactics

Sorts are defined by simple identifiers:

- **type** S .

Types are defined either by base type definitions **type** $T = BTE$, where BTE is an atomic type expression, for example either of,

- **Intg**[:Dim],
- **Nat**[:Dim],
- **Real**[:Dim],
- **Char**[:Dim],
- **Token**,
- POINT and
- T, TI.

where [:Dim] is either absent or some standard prefix and fraction SI unit. Or types are defined by composite type expressions, **type** $T = \text{CTE}$, for example of the form:

$$\text{CTE} = \text{A-set} \mid B \times C \times \dots \times D \mid E \rightarrow F \mid \text{etc.}$$

where A, B, C, ..., D, E, F, etc., are type expressions – where [recursive] T is allowed.

Semantics

We start with types. **Types** are sets of either base (type) values, or structures over these: sets of sets (of etc.), sets of Cartesians (of etc.), sets of maps (from etc.), et cetera. **Sorts** are sets of endurants as characterised by their being discrete or continuous (solids or non-solids), physical parts, living species or structures, natural parts or artefacts, and atomic, composite, components or set sorts; and as furthermore characterised by the types of their possible unique identifiers, possible mereologies (components have no mereologies), and attributes.

7.6.2 An Earlier Review of Types

Section 5.3, Pages 43–48, of [70], brings an extensive review of published papers on types. That review does not make the distinction made in this paper as summarised in Sect. 7.6.1.

MORE TO COME

7.6.3 Domain Oriented Programming Languages

I found out about Kennedy’s work from [133]. My own interest in the subject goes back to the early 1980s. Around year 2000 I had an MSc student work out formal specifications and compilers for two “small” programming languages: one for senior high school [student] physics and one for business college [student] accounting. I otherwise refer to [31, Exercise 9.4, Page 235].

One could, rather easily, augment standard programming languages, for use in physics calculations, to feature a refined type system that reflects the SI units, simple and composite, as well as standard SI prefixes and fractions.

We refer to the very elegant domain-specific actuarial programming language, **Actulus**, [111] for life insurance and pensions.

Our *Domain Specific Language dogma* is this: **the design (and semantics) of any DSL must be based on a carefully analysed and both informally and formally described domain.**

7.6.4 Research Topics

- **Artefactual Types:** A further study of artefactual types seems reasonable: *are there identifiable categories of artefactual types?*
- **Intents:** We have remarked that we suggest no formal representation of intents. But should there be?
- **Intentional Pull:** Although we have illustrated some “*intentional pulls*”, also in [79, 80], it seems only reasonable to study further examples.

Attributes of Living Species: The Swedish botanist, zoologist, and physician, *Carl von Linné*, is the father of modern taxonomy: the science that finds, describes, classifies, and names living things, published [255, in 1748]. In domain analysing & describing living species one, of course, cannot really contribute much new. So we leave that area to the living species taxonomists – while referring to [130, *Formal Concept Analysis — Mathematical Foundations*]. See also [70, Sect. 1.8].

7.6.5 **Acknowledgements**

It is a pleasure to acknowledge my collaboration over recent years with Klaus Havelund.

Issues of Philosophy

Chapter 9 is still in draft form.
I expect to work on this chapter during the summer of 2019.
I Hope to be able to complete my work with a publishable paper.

Domain Analysis & Description

A Philosophy Basis

This chapter consists of two parts: A philosophy part and a terse summary Chapter 1.

In the philosophy part, Sect. 8.1.2, we outline Kai Sørlander’s philosophy on **what must necessarily be in any description of any world**.

In the domain analysis & description part, Sects. 8.3–8.8, we present a new preamble for software engineering, one that precedes requirements engineering. We outline two calculi: one for the analysis of the endurants of human artefact “dominated” domains, and one for their description. By a transcendental deduction endurant domain descriptions are translated into perdurant domain descriptions: manifest parts becoming behaviours.

We show how the ontology of the second part is basically founded on the necessities of description outlined in the first part — thereby contributing to a philosophy basis for computing.

8.1 Introduction

Before **software** can be **designed** the programmer must **grasp** its requirements. Before **requirements** are **prescribed** the engineer must **grasp** an **adequate extent** of the **domain** in which that software is to serve. **But do software engineers today have a sufficient grasp of their target domains?**

Software designs are always formally specified: the program code must be executable by computer – and as such, precise mathematical statements, including properties of computations, as directed by the code, can be expressed, and possibly proved. [29] covers formal software design. **Requirements prescriptions** are sometimes formally specified. **Domain descriptions** are introduced in [45, 70, 76, 80]. It is shown in these papers that domain descriptions can be formalised. And in [35, 66] it is shown how formal domain prescriptions can be for “derived” from domain descriptions.

Formal requirements prescriptions and software designs are expressed in a computable formalism. Not so with domain descriptions. The domains they focus on are not [necessarily] computable. A rôle of requirements prescriptions is to identify and describe relevant computable subsets of the domain. So consistent and complete formal requirements prescriptions and software designs refer, basically, to mathematical objects that do exist in the mathematical universes of their specifications. Domain analysis [& description] does not have this “advantage”! The domains basically “reside” in “mother nature” and, especially, in the systems of man-made entities based on mother nature. Any characterisation of domain entities is therefore “hovering” between meta-physics and physics. Hence our interest in studying philosophical bases for domain analysis & description.

By *an adequate* (domain) we do not mean “an entire, full, complete” domain, but a sizable part of it — usually “somewhat more” than entailed by [subsequent] requirements. By *a grasp* (of a domain) we mean an understanding that enables us to **reason** about the domain. Our ‘reasonable grasp’ is, we suggest, manifested in some text; that is, in some language.

In order to *reason* we expect that language to be **formal**; that is, to have a formal **syntax**, a formal, mathematical **semantics** and a **proof system**. We do not expect the **domain**, the **requirements** and the **software design** [incl. coding] languages to be the same, one **specification** language.

Ideally software development, therefore, to us, entails three major phases: **domain engineering** in which we analyze and describe the domain, \mathcal{D} , in which the software is to serve; **requirements engineering** in which we “derive” the requirements, \mathcal{R} , that the software is to fulfill; and **software design** in which we finally arrive, via one or more steps of software design, \mathcal{S} .

Domain descriptions express **properties** of the chosen domain. **Requirements prescriptions** express **desired properties** of the **desired software**, not how it is implemented. **Software designs** express how requirements are to be **satisfied by executable code**.

Software, \mathcal{S} , is expected to fulfill customer *expectations*, hence must be based on *domain understanding*, \mathcal{D} , and to be *correct* with respect to requirements, \mathcal{R} . We can express this formally: $\mathcal{R} \models \mathcal{D}$ (\mathcal{R} models \mathcal{D}), respectively $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ (\mathcal{S} , in the context of \mathcal{D} , models \mathcal{R}). This obviously means that we expect developers to consider domain, requirements and software specifications (incl. code) as mathematical objects and that tests, model checks and theorem proofs ideally be carried out accordingly.

The above portrays a **hypothetical situation**. Today’s software engineering essentially has no domain engineering phase. This current situation is unlike any other engineering. The classical disciplines of chemical, civil, electrical and mechanical engineering all build on the sciences of physics. Major software systems are today developed without a proper understanding of their underlying domain. It seems that software engineering today starts, “at best”, with requirements engineering.

The present paper, as well as its precedent papers [35, 45, 52, 55, 61, 62, 71, 75, 78, 80], lead us to claim that domain science & engineering offers a compelling *new foundation for software development*. Physics is the basis for all branches classical of engineering. So, we claim, [that] domain science, is a basis for all software. We shall, however, cover only a fragment of this basis.

8.1.1 Domain Science & Engineering

DEF.1: By a **domain** we shall understand a **rationaly describable discrete dynamics** segment of a **human assisted** reality. That is, of the world, its **physical parts: natural** [“God-given”] and **artefactual** [“man-made”], and **living species: plants** and **animals** including, notably, **humans** ■ **DEF.2:** By a *discrete dynamics* segment of the world we shall understand a reality whose dynamics “move” in steps, and where possible continuous changes are otherwise ignored ■ **DEF.3:** By *rationaly describable* we mean describable using, for example, the principles, techniques and analysis & description prompts of this paper and detailed in [80] ■ **DEF.4:** By *human assisted reality* we mean a universe of discourse with at least one artefactual phenomenon and as monitored & controlled by at least one human ■ We present an essence of two calculi: a *calculus* of **domain analysis prompts** and a *calculus* of **domain description prompts**. We shall only present the *prompts*, not their algebra, that is, not the laws of combined uses of prompts. **DEF.5:** By a **prompt** we shall here mean an act of encouraging the domain analyser cum describer, that is, a human, to do something, here: to analyse and/or describe ■ **DEF.6:** By **domain analysis** we mean an *inquiry*, by domain analysers, i.e., humans, into the *make-up* of a domain, with the analysis resulting in affirmative answers, to questions like “*is, what I am observing, such-and-such*” – **true** or **false** ■ **DEF.7:** By a **domain description** we mean a textual document, both informal, the narrative, and formal, the specification ■ The **narrative** is a natural language text which in terse statements introduces the names of the domain, and, possibly, also the definitions, of sorts (types) of syntactic and semantic entities, actions, events and behaviours, and axioms; not anthropomorphically, but by emphasizing their properties. The **formal specification** is a collection of sort, or type definitions, function and behaviour definitions, together with axioms and proof obligations constraining the definitions. So the problem is to analyse and describe a domain, that is, to describe physical parts, whether natural or man-made, and, in [rare] cases also living species: plants and animals, notably humans. The next many sections shows how we tackle – and hence expect others to tackle – that problem. The approach takes its departure in **philosophy**. Most decisively in the philosophy of **Kai Sørlander**.

8.1.2 Some Issues of Philosophy

The question is: “what, if anything, is of such necessity, that it could under no circumstances be otherwise?” or “which are the necessary characteristics of any possible world?”. We take it as the necessary characteristics of any domain is equivalent with the conceptual, logical conditions for any possible description of that domain. Sørlander puts forward the thesis of *the possibility of truth* and then basing *transcendental deductions* on indisputable logical relations to arrive at the conceptual, logical conditions for any possible description of that domain. The starting point, now, in a series of deductions, is that of logic and that we can assert a property, \mathcal{P} , and its negation $\neg\mathcal{P}$; and that *these two assertions cannot both be true*, that is, that $\mathcal{P} \wedge \neg\mathcal{P}$ cannot be true. So the *possibility of truth* is a universally valid condition. When we claim that, we also claim *the contradiction principle*. The *implicit meaning theory* is this: “in assertions there are mutual dependencies between the meaning of designations and consistency relation between assertions”. When we claim that a philosophy basis is that of the possibility of truth, then we assume that this basis include the contradiction principle and the implicit meaning theory. We shall also refer to the implicit meaning theory as the **inescapable meaning assignment**.

As an example of what “goes into” the *inescapable meaning assignment*, we bring, albeit from the world of computer science, that of the description of the **stack** data type (its endurants and operations).

An Inescapable Meaning Assignment Example, Narrative	Formalisation
The meaning of designations:	
502 Stacks, $s:S$, have elements, $e:E$;	
503 the <code>empty_S</code> operation takes no arguments and yields a result stack;	
504 the <code>is_empty_S</code> operation takes an argument stack and yields a Boolean value result.	
505 the <code>stack</code> operation takes two arguments: an element and a stack and yields a result stack.	
506 the <code>unstack</code> operation takes a non-empty argument stack and yields a stack result.	
507 the <code>top</code> operation takes a non-empty argument stack and yields an element result.	
The consistency relations:	
508 an <code>empty_S</code> stack <code>is_empty</code> , and a stack with at least one element is not;	
509 unstacking an argument stack, <code>stack(e,s)</code> , results in the stack <code>s</code> ; and	
510 inquiring as to the top of a non-empty argument stack, <code>stack(e,s)</code> , yields <code>e</code> .	

The Inescapable Meaning Assignment Example, Formalisation	Formalisation
The meaning of designations:	
type	
1. E, S	
value	
2. <code>empty_S</code> : $\mathbf{Unit} \rightarrow S$	
3. <code>is_empty_S</code> : $S \rightarrow \mathbf{Bool}$	
4. <code>stack</code> : $E \times S \rightarrow S$	
5. <code>unstack</code> : $S \xrightarrow{\sim} S$	
6. <code>top</code> : $S \xrightarrow{\sim} E$	
The consistency relations:	
7. <code>is_empty(empty_S())</code> = true	
7. <code>is_empty(stack(e,s))</code> = false	
8. <code>unstack(stack(e,s))</code> = <code>s</code>	
9. <code>top(stack(e,s))</code> = <code>e</code>	

8.1.3 Transcendence

DEF.8: By **transcendental** “we shall understand the philosophical notion: the a priori or intuitive basis of knowledge, independent of experience” ■ **DEF.9:** By a **transcendental deduction** “we shall understand the philosophical notion: a transcendental ‘conversion’ of one kind of knowledge into a seemingly different kind of knowledge” ■ Transcendental philosophy, with Kant and Sørlander, seeks to find the necessary conditions for experience, recognition and understanding. Transcendental deduction is then the “process” based on the principle of contradiction and the implicit meaning theory by means of which – through successive concept definitions one can deduce a system of base concepts which must be assumed in any possible description of the world. The subsequent developments of the logical connectives, modalities, existence, identity, difference, relations, numbers, space, time and causality, are all transcendental deductions.

8.2 Overview of The Sørlander Philosophy

In this section we shall give a very terse summary of main elements of Kai Sørlander’s philosophy. We shall primarily base this overview on [245]. It is necessarily a terse summary. What we overview is developed in [245] Sørlander over some 50 pages. Sørlander’s books [238, 239, 241, 245], relevant to this overview, are all in Danish. Hence the need for this section.

8.2.1 Logical Connectives

Negation: \neg : The logical connective, **negation** (\neg), is defined as follows: if assertion \mathcal{P} holds then assertion $\neg\mathcal{P}$ does not hold. That is, the contradiction principle understood as a definition of the concept of negation.

Conjunction and Disjunction: \wedge and \vee Assertion $\mathcal{P} \wedge \mathcal{Q}$ holds, i.e., is true, if both \mathcal{P} and \mathcal{Q} holds. Assertion $\mathcal{P} \vee \mathcal{Q}$ holds, i.e., is true, if **either** \mathcal{P} or \mathcal{Q} or **both** \mathcal{P} and \mathcal{Q} holds.

Implication: \Rightarrow Assertion $\mathcal{P} \Rightarrow \mathcal{Q}$ holds, i.e., is true, if the first assertion, \mathcal{P} , holds, t , and the second assertion, \mathcal{Q} , is not false, $\neg f$. $[(\mathcal{P}, \mathcal{Q}), \mathcal{P} \Rightarrow \mathcal{Q}]$: $[(t, t), t]$, $[(t, f), f]$, $[(f, t), f]$, and $[(f, f), t]$.

8.2.2 A Philosophy Basis for Physics and Biology

In a somewhat long series of deductions we shall, based on Sørlander’s Philosophy, motivate the laws of Newton and more, not on the basis of empirical observations, but on the basis of transcendental deductions and rational reasoning.

Possibility and Necessity Based on logical implication we can transcendently define the two *modal operators*: **necessity** and **possibility**. **DEF.10:** An assertion is **necessarily** true if its truth follows from the definition of the designations by means of which it is expressed ■ **DEF.11:** An assertion is **possibly** true if its negation is not necessary ■

Empirical Assertions There can be assertions whose truth value does **not only depend** on the definition of the designations by means of which they are expressed. Those are assertions whose truth value **depend also** on the assertions *referring* to something that exists independently of the designations by means of which they are expressed. We shall call such assertions **empirical**.

Existence With Sørlander we shall now argue that **there exist many entities in any world**: [245, pp 145] “Entities, in a first step of reasoning, that can be referred to in empirical assertions do not necessarily exist. It is, however, an empirical fact that they do exist; hence there is a logical necessity that they do not exist. In a second step of reasoning, these entities must exist as a necessary condition for their actually being ascribed the predicates which they must necessarily befit in their capacity of of being entities referred to in empirical assertions.”

Identity, Difference and Relations [245, pp 146] “An entity, referred to by A , is **identical** to an entity, referred to by B , if A cannot be ascribed a predicate, in-commensurable with a predicate ascribed to B .” That is, if A and B cannot be ascribed in-commensurable predicates. [245, pp 146] “Entities A and B are

different if they can be ascribed in-commensurable predicates.” [245, pp 147] “**Identity** and **difference** are thus transcendently derived through these formal definitions and must therefore be presupposed in any description of any domain and must be expressible in any language.” Identity and difference are **relations**. [245, pp 147] “As a consequence identity and difference imply relations. **Symmetry** and **asymmetry** are also relations: *A identical to B is the same as B identical to A. And A different from B is the same as B different from A. Finally **transitivity** follows from A identical to B and B identical to C implies A identical to C.*”

Sets: We can, as a consequence of two or more different entities satisfying a same predicate, say \mathcal{P} , define the notion of the **set** of all those entities satisfying \mathcal{P} . And, as a consequence of two or more entities, e_i, \dots, e_j , all being *distinct*, therefore implying in-commensurable predicates, $\mathcal{Q}_i, \dots, \mathcal{Q}_j$, but still satisfying a common predicate, \mathcal{P} , we can claim that they all belong to a same set. The predicate \mathcal{P} can be said to **type** that set. And so forth: following this line of reasoning we can introduce notions of cardinality of sets, finite and infinite sets, existential (\exists) and universal (\forall) quantifiers, etc.; and we can in this way transcendently deduce the concept of (positive) numbers, their addition and multiplication; and that such are an indispensable aspect of any domain. We leave it then to mathematics to study number theory.

Space and Geometry **DEF.12: Space:** [245, pp 154] “The two relations **asymmetric** and **symmetric**, by a transcendental deduction, can be given an interpretation: the relation (spatial) **direction** is asymmetric; and the relation (spatial) **distance** is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation in-between. Hence we must conclude that **primary entities exist in space**. Space is therefore an unavoidable characteristic of any possible world” ■ [245, pp 155] “Entities, to which reference can be made in simple, empirical assertions, must exist in space; they must be spatial, i.e., have a certain extension in all directions; they must therefore “fill up some space”, have surface and form.” From this, by further reasoning one can develop notions of points, line, surface, etc., i.e., Euclidean as well as non-Euclidean **geometry**.

States We introduce a notion of **state**. [245, pp 158–159] “Entities may be ascribed predicates which it is not logically necessary that they are ascribed. How can that be possible? Only if we accept that entities may be ascribed predicates which are in-commensurable with predicates that they are actually ascribed.” That is possible, we must conclude, if entities can **exist** in distinct **states**. We shall let this notion of state further undefined – till Sect. 8.5.3.

Time and Causality **DEF.13: Time:** [245, pp 159] “Two different states must necessarily be ascribed different incompatible predicates. But how can we ensure so? Only if states stand in an asymmetric relation to one another. This state relation is also transitive. So that is an indispensable property of any world. By a transcendental deduction we say that **primary entities exist in time**. So every possible world must exist in time” ■ So space and time are not phenomena, i.e., are not entities. They are, by transcendental reasoning, aspects of any possible world, hence, of any description of any domain. In a concentrated series [245, 160-163] of logical reasoning and transcendental deductions, Sørlander, introduce the concepts of the empirical circumstances under which entities exist, implying *non-logical implication* between one-and-the-same entity at distinct times, leading to the notions of **causal effect** and **causal implication** – all deduced transcendently. Whereas Kant’s *causal implication* is transcendently deduced as necessary for the *possibility of self-awareness*. Sørlander’s *causal implication* does not assume *possibility of self-awareness*. The **principle of causality** is a necessary condition for assertions being about the same entity at different times.

Kinematics [245, pp 164] “Entities are in both space and time; therefore it must be assumed that they can change their spatial properties; that is, are subject to **movement**. An entity which changes location is said to **move**. An entity which does not change location is said to be **at rest**.” In this way [245] transcendently introduces the notions of velocity and acceleration, hence kinematics.

Dynamics [245, pp 166] “When combining the causality principle with dynamics we deduce that when an entity changes its state of movement then there must be a cause, and we call that cause a **force**.” [245, pp 166] “The **change** of state of entity movement must be **proportional** to the applied force; an entity not subject to an external force will remain in its state of movement: This is **Newton’s 1st Law**.”

[245, pp 166] “But to change an entity’s state of movement, by some force, must imply that the entity exerts a certain **resistance** to that change; the entity must have a **mass**. Changes in an entity’s state of movement besides being proportional to the external force, must be **inverse proportional** to its mass. This is **Newton’s 2nd Law**.”

[245, pp 166-167] “The forces that act upon entities must have as source other entities: entities may **collide**; and when they collide **the forces** they exert on each other must be **the same but with opposite directions**. This is **Newton’s 3rd Law**.”

[245, pp 167-168] “How can entities be the source of forces? How can they have a mass? Transcendentally it must follow from what we shall refer to as **gravitational pull**. Across all entities of mass, there is a mutual attraction, **universal gravitation**.” [245, pp 168-169] “Gravitation must, since it has its origin in the individual entities, propagate with a definite velocity; and that velocity must have a limit, a constant of nature, the **universal speed limit**.”

From Philosophy to Physics

Based on logical reasoning and transcendental deductions one can thus derive major aspects of that which must be (assumed to be) in any description of any world, i.e., domain. In our domain description ontology we shall let the notions of **discrete endurants (parts)** and **continuous endurants (non-solids)** cover what we have covered so far: they are those entities which satisfy the laws of physics, hence are in space and time. In the next sections we shall make further use of Sørlander’s Philosophy to logically and transcendently justify the inevitability of **living species: plants** and **animals** including, notably, **humans**, in any description of any domain.

Purpose, Life and Evolution

[245, pp 174] “For language and meaning to be possible there must exist entities that are not constrained to just the laws of physics. This is possible if such entities are further subject to a “**purpose-causality**” directed at the future. These entities must strive to maintain their own existence.” We shall call such entities **living species**. Living species must maintain and also further develop their form and do so by an exchange of materials with the surroundings, i.e., **metabolism**, with one kind of living species subject only to development, form and **metabolism**, while another kind additionally **move purposefully**. The first we call **plants**, the second **animals**. Animals, consistent with the principle of causality, must possess **sensory organs**, a **motion apparatus**, and **instincts, feelings, promptings** so that what has been sensed, may be responded to [through motion]. The **purpose-directness** of animals must be built into the animals. Biology shows that that is the case. The animal genomes appear to serve the **purpose-directness** of animals. [245, pp 178] “Biology shows that it is so; transcendental deduction that it must be so.”

Awareness, Learning and Language

[245, pp 180] “Animals, to **learn from experience**, must be able to feel **inclination and disinclination**, and must be able to remember that it has acted in some way leading to either the feeling of inclination or disinclination. As a consequence, an animal, if when acting in response to sense impression, ι , experiences the positive feeling of inclination (desire), then it will respond likewise when again receiving sense impression ι , until it is no longer so inclined. If, in contrast, the animal feels the negative feeling of disinclination (dislike), upon sense impression ι , then it will avoid responding in this manner when receiving sense impression ι .” [245, pp 181] “Awareness is built up from the sense impressions and feelings on the basis of, i.e., from what the individual animal has learned. Different animals can be expected to have different levels of consciousness; and different levels of consciousness assume different biological bases for learning. This is possible, biology tells us, because of there being a central nervous system with building blocks, the neurons, having an inner determination for learning and consciousness.” [245, pp 181–182] “In the mutual interaction between animals of a higher order of consciousness these animals learn to use **signs** developing

increasingly complex sign systems, eventually “arriving” at languages.” It is thus we single out **humans**. [245, pp 183] *“Any human language which can describe reality, must assume the full set of concepts that are prerequisites for any world description.”*

•••

We have concluded the presentation of a major issue of this paper. that of a philosophy that may be a possible basis for domain science & engineering. We now “*apply*” this, Kai Sørlander’s, Philosophy to the problem of domain analysis & description.

8.3 Phenomena and Entities

DEF.14: By an entity, **is_entity**, we shall understand a phenomenon, i.e., *something that can be observed, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity*; alternatively, a phenomenon is an entity, *if it exists, it is “being”, it is that which makes a “thing” what it is: essence, essential nature* [170, Vol. I, pg. 665] ■ An entity is what we can analyse and describe using the analysis & description prompts outlined in this paper. Many of the entities that we are concerned with are those with which Kai Sørlanders Philosophy is likewise concerned. They are the ones that are unavoidable in any description of any possible world.

•••

Before main Sects. 8.4–8.8, we introduce two categories of entities: endurants and perdurants.

8.3.1 Endurants

DEF.15: By an **endurant** we shall understand *an entity that can be observed, or conceived and described, as a “complete thing” at no matter which given snapshot of time*; alternatively *an entity is endurant if it is capable of enduring, that is persist, hold out* [170, Vol. I, pg. 656]. Were we to “freeze” time we would still be able to observe the entire endurant ■ Endurants, thus, are capable of enduring. **EX.1. Endurants:** *A train wagon, a rail track and a railway station* ■ We suggest that the concept of endurants can be seen as a transcendental deduction based on the inescapable fact that there is a multitude of entities, and that considering these as existing in just space, are the endurants. But note that endurants are [to be] *observed*.

8.3.2 Perdurants

DEF.16: By a perdurant we shall understand *an entity for which only a fragment exists if we look at or touch them at any given snapshot in time. Were we to freeze time we would only see or touch a fragment of the perdurant, alternatively an entity is perdurant if it endures continuously, over time, persists, lasting* [170, Vol. II, pg. 1552] ■ Perdurants are entities that only exists partially at any given point in time. **EX.2. Perdurant:** *a train ride* ■ We suggest that the concept of perdurants can be seen as a transcendental deduction based on the inescapable fact that there are a multitude of entities, and that considering these as existing in both space and time, are the perdurants.

•••

Sections 8.4–8.6 and Sect. 8.8 reviews the complex, conceptual “universes” of endurants, respectively perdurants. Section 8.7 unveils, by a transcendental deduction, the link between endurants and perdurants: that endurant parts transcend into behaviours. Figure 8.1 suggests a structuring of endurants, perdurants and their relations.

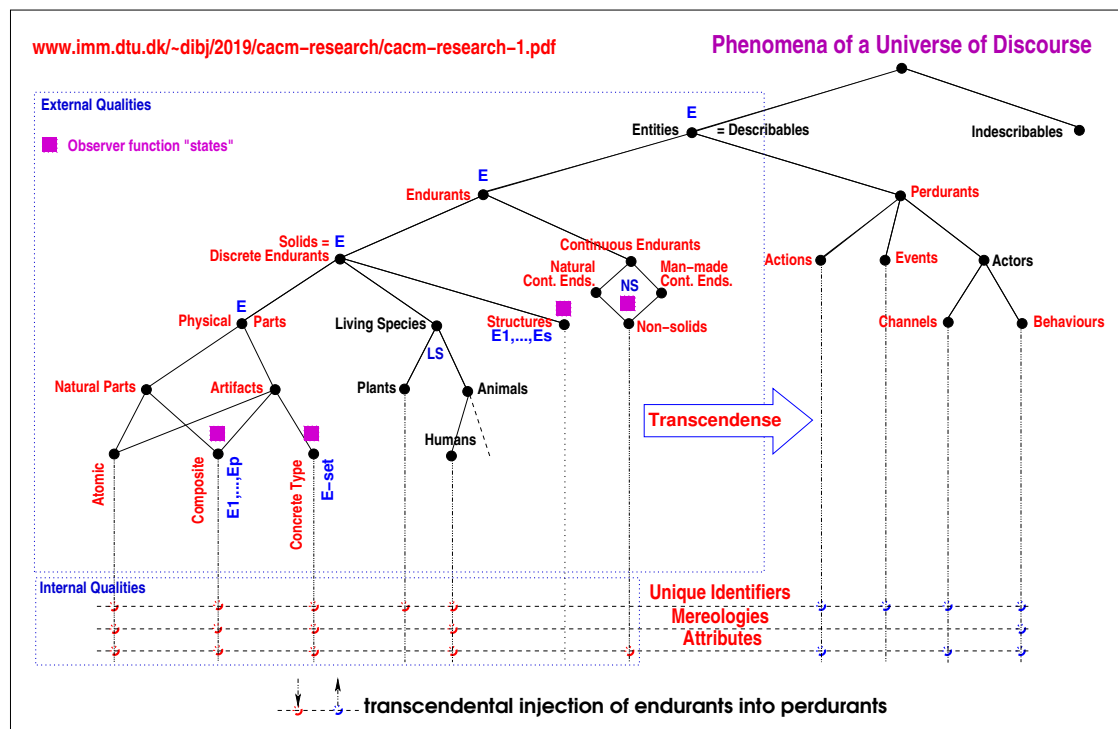


Fig. 8.1. An Upper Ontology for Domain Entities. We shall not discuss black labeled entity classes. The magenta coloured square boxes, ■, designate “analysis states” where description prompts apply.

8.4 Endurants: External Qualities

Observable qualities of endurants are those that can be touched. We choose, it may seem arbitrarily, to analyse endurants into either **discrete** (solid) or **continuous** (non-solid) endurants. That is, we claim that endurants can be so analysed either of one kind, or of the other, but not both! We justify the distinctions based on physics.

8.4.1 Discrete and Continuous Endurants

DEF.17: By a **discrete endurant** (a solid) we shall understand an endurant which is **separate, individual or distinct in form or concept** [170, OED] ■ **EX.3. Discrete Endurants:** a particular canal lock, a particular canal link between two specific adjacent locks, a particular barge ■ **DEF.18:** By a **continuous endurant** (a non-solid) we shall understand an endurant which is **prolonged, without interruption, in an unbroken series or pattern** [170, OED] ■ We think of a non-solid to be either a gas or a plasma or a liquid. **EX.4. Continuous Endurant:** water (in a specific canal), air (in a specific ventilator pump), beer (in a specific bottle) ■ Note, and please accept, the OED definitions. They are not precise in the sense of mathematics. We are not dealing with an exact ‘world’. We are dealing with real worlds.

8.4.2 Solids

Solids, i.e., discrete endurants, are either physical parts, or living species, or structures. We shall motivate the first two categories of solids on the background of Sørlander’s philosophy. Structures are motivated pragmatically.

Physical Parts and Living Species

DEF.19: By a **physical part** we shall understand a discrete endurant existing in space and time and subject to laws of physics, including the causality principle and gravitational pull – and which is not a living species or an animal ■ **EX.5. Physical parts:** *A pipeline system, a specific pipeline, units of a specific pipeline: a specific pipe, a specific valve, a specific pump, etc.* ■ **DEF.20:** By a **living species** we shall understand a discrete endurant, subject to laws of physics, and additionally subject to causality of purpose ■ **EX.6. Living Species:** *a specific garden, a specific flower bed, a specific rhododendron, a specific bird, a specific human* ■ In this paper we shall not elaborate on the possibility of natural versus man-made living species.

Natural Parts and Artefacts

DEF.21: By a **natural part** we shall understand physical parts, i.e., that are in space and time, are subject to the laws of physics, and also subject to the principle of causality and gravitational pull, but are not man made and not living species ■ **EX.7. Natural Parts:** *a particular landscape, a particular lake, a particular forest, a particular mountain* ■ **DEF.22:** By an **artefact** we shall understand physical parts that are man made with one or more intents ■ We shall explain the notion of *intent* later. **EX.8. Artefacts:** *a specific road network, with specific automobiles, specific hubs (at road intersections), specific links (between two adjacent hubs), specific routes (contiguous sequences of zero, one or more adjacent, alternating hubs and links).* The intents are that *automobiles* drive along *routes* and that *hubs* and *links* serve as conduits for *automobiles* ■

Atomic or Composite Parts

DEF.23: By an **atomic part** we shall understand a part which, in a given context, deemed to **not** contain of meaningful, separately observable proper sub-part. A sub-part is a part ■ **EX.9. Atomic Artefacts:** *hubs, links, automobiles* ■ We shall not consider natural parts as other than that, neither atomic, nor composite in this paper. **DEF.24:** By a **composite part** we shall mean physical parts which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts ■ **EX.10. Composite Artefacts:** *road nets, pipeline systems, railway systems* ■ **EX.11. Elements of a Composite Artefact:** The domain of road transport is assumed to contain a road net which then contains a set of links, a set of hubs, a set of automobiles, a set of zero, one or more road maintenance departments, and a set of zero, one or more automobile clubs. It may contain other parts ■

Concrete Type Artefacts

In addition to **atomic** and **composite** artefacts there are **concrete type** artefacts. The analysis into this variety of three kinds is based on pragmatic grounds. **DEF.25:** By a **concrete type artefact** we shall, simplifying, mean a set of endurants, all of the same sort ■ **EX.12. Concrete Type Artefacts:** *a specific set of hubs, a specific set of links, a specific set of automobiles* ■

Plants, Animals and Humans

Living species are either plants or animals. **DEF.26:** By a **plant, animal** and **human** we shall understand what Kari Sørlander's Philosophy transcendently arrives at as such ■ We omit examples !

Structures

DEF.27: By a **structure** we shall understand a discrete endurant which the domain engineer chooses to describe as consisting of one or more endurants, whether discrete or continuous, but to not endow with

internal qualities: unique identifiers, mereology or attributes ■ Structures are “conceptual endurants”. A structure “gathers” one or more endurants under “one umbrella”, often simplifying a presentation of some elements of a domain description. Sometimes, in our domain modelling, we choose to model an endurant as a structure, sometimes as a physical part; it all depends on what we wish to focus on in our domain model. As such structures are “compounds” where we are interested only in the (external and internal) qualities of the elements of the compound, but not in the qualities of the structure itself.

8.4.3 Non-solids

An entity may thus be a non-solid. A composite part, p , natural or man-made, may have one or more non-solid entities, though with at least one solid entity — in which case `has_non_solids(p)`.

8.4.4 The Analysis Prompts

We summarise the analysis prompts informally introduced in this section.

- `is_entity`, • `is_living`, • `is_atomic`, • `has_non_solids`,
- `is_discrete`, • `is_structure`, • `is_composite`, • `has_components`.
- `is_continuous`, • `is_natural`, • `is_concrete`,
- `is_physical`, • `is_artefact`, • `is_component`,

8.5 Endurants: Internal Qualities

Internal qualities of endurants are those qualities that cannot be touched but can be either conceptualised or measured. We consider the following internal qualities: **unique identifiers**, **mereology**, and **attributes**. Physical parts have the full set of internal qualities. Structures are endurants for which the domain analyser cum describer had decided to not endow with internal qualities. We shall not, in this paper, be concerned with the internal qualities of living species.

8.5.1 Unique Identifiers

It is based on the philosophy idea of *identity*, cf. Sect. 8.2.2, that we associate with each solid a unique identifier. **Ex.13. Road Net Links and Hubs:** The road net of a transport system consists of links, i.e., street segments, and hubs, i.e., street intersections. Links of any road net have unique identifiers. (Links of all road nets are distinctly identified.) Hubs of any road net are distinctly identified.

8.5.2 Mereology

DEF.28: Mereology is the study of parts and the wholes they form ■ Mereology, as here put forward, is due to the Polish philosopher/logician/mathematician Stanisław Leśniewski [172, 104]. There are basically two relations that can be relevant for part-hood (i) a topological one, and (ii) a temporal one. (i) Physically two or more parts may be adjacent to one another or one within another. (ii) Temporally some parts, “relate”, over time, to a “therefrom physically distinct” part. **Ex.14. Topological Mereology:** The mereology of links of a road net is a set of two distinct hub identifiers of that net, and of hubs of a road net is a set of zero, one or more link identifiers of that net. The mereology thus defines a concept of routes of a road net, and must be such that there is at least one route from any hub to any other hub of a road net ■ **Ex.15. Temporal Mereology:** The mereology of an automobile (of a road transport system) identifies the hubs and links that it may, over time, traverse and the zero, one or more automobile clubs it may be a member of and may contact, over time. The mereology of a hub and a link, (of a road transport system) in addition

to what has already been ascribed to hubs and links, identifies one road maintenance department that, over time, maintains hubs and links ■

We may model the mereology of a part, p , as a triplet: an input set of unique identifiers of parts from which p “receives input” in a sense not further described here; a pair of input/output sets of unique identifiers of parts from which p “receives input” and to which “delivers output” in a sense not further described here; and an output set of unique identifiers of parts to which p “delivers output” in a sense not further described here.

8.5.3 Attributes

Unique identifications and mereologies form abstract concepts. Although topological mereologies may be observed they, and unique identification, are not manifest – although they can be the quantities that are referred to in empirical assertions. Attributes are measurable properties of endurants, properties that can be referred to in empirical assertions — they, so-to-speak, gives “*flesh and blood*”, that is, substance to endurants. Endurants are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. We equate all endurants which, besides possible type of unique identifiers and possible type of mereologies, have the same types of attributes, with one sort. Removing a quality from an endurant makes no sense: the endurant of that type either becomes an endurant of another type or ceases to exist (i.e., becomes a non-entity) !

Attribute Categories

Attributes [157] are either of **static** value, i.e., does change value, or of **monitorable** value, i.e., **dynamic inert** or **reactive**: monitorable values can change, or of **controllable** value, i.e., **dynamic biddable** or **programmed**: biddable values can be prescribed, but prescription may fail; programmable values can be set. **Ex.16. Link Attributes**: Typical link attributes could be: location (e.g., as a Bézier curve) [static], length [static], road condition (icy, dry, ...) [monitored], state – as a set of pairs of adjacent hub identifiers [controllable], state spaces – as set of all such states [static], and automobile history: recordings of which automobiles have been on the link, at which position and time ■ **Ex.17. Hub Attributes**: Typical hub attributes could be: location [static], state – as set of pairs of adjacent link identifiers [controllable], and automobile history: recordings of which automobiles have been on the hub, and at which time. States are abstractions of traffic signals ■ **Ex.18. Automobile Attributes**: Typical automobile attributes could be: position (on hub or link), velocity, etc., road net history: recordings of the hubs and links on which the automobile has been, at which position and time ■

Artefact Intents

With artefacts we can associate **intents**. **DEF.29**: By an **intent** of an artefact we shall understand a simple label which informally indicates the purpose for which the artefact is intended ■ An artefact may be ascribed more than one intent. Artefacts are usually ascribed at least one intent. **Ex.19. Intents of Automobiles and Road Nets**: To automobiles we may ascribe the intent that they are located on the road net, i.e., on hubs and links; and to hubs and links we then ascribe the intent that they accommodate automobiles ■

Intentional Pull

Gravitational pull, cf. Sect. 8.2.2, follows from Newton’s Third Law. Intentional pull “follows” from the fact that pairs or triples, etc., of artefacts of different sorts, may be ascribed commensurate intents. **Ex.20. Intentional Pull between Automobiles and Road Net**: If an automobile’s road net history records that it has visited a road net unit at time t and position π , then that road net unit’s automobile history records that very same fact ! And vice versa. It cannot be otherwise ! ■

States

By a state we shall understand any collection of endurants for which any one endurant has at least one dynamic, i.e., non-static, attribute. By the state of a behaviour we shall understand its current *program point*, that is, its point of execution, and the collection of its monitorable and controllable variables, that is, of their current values.

8.6 Endurants: Description Prompts

So far we have outlined a number of domain analysis prompts, cf. Sect. 8.4.4. We now summarise some description prompts. We refer to Fig. 8.1. The “analysis states” marked with magenta colored square boxes, ■, correspond, left-to-right in the ontology graph to the following description prompts: **observe_endurant_sorts**, **observe_concrete_part**, **observe_component_sort**, **observe_structure_components** and **observe_non_solids**. Sections 8.3–8.4 can be summarised formally:

type: observe_endurant_sorts : $E \rightarrow \text{Text}$
<p>Narrative:</p> <ul style="list-style-type: none"> s. narrative text on sorts E_1, \dots, E_n o. narrative text on observers $\text{obs}_{E_1}, \dots, \text{obs}_{E_n}$ p. narrative text on proof obligation: \mathcal{P} <p>Formalisation:</p> <ul style="list-style-type: none"> s. type E_1, \dots, E_n o. value $\text{obs}_{E_1}: E \rightarrow E_1, \dots, \text{obs}_{E_n}: E \rightarrow E_n$ p. proof obligation $\mathcal{P}: \forall i: \{1..n\} \cdot \text{is}_{E_i}(e) \equiv \bigwedge \{ \sim E_j(e) \mid j: \{1..n\} \setminus \{i\} \}$

In any specific domain analysis & description the analyser cum describer chooses which subset of composite sorts to analyse & describe. That is: any one domain model emphasises certain aspects and leaves out many “other” aspects. A similar observer is defined for concrete type parts, cf. Sect. 8.4.2:

type: observe_concrete_part : $E \rightarrow \text{Text}$
<p>Narratives:</p> <ul style="list-style-type: none"> s. narrative text on sort P o. narrative text on observer obs_P <p>Formalisation:</p> <ul style="list-style-type: none"> s. type P, $P_s = \text{P-set}$ o. value $\text{obs}_P: E \rightarrow \text{P-set}$

Typically P may be a sort expression: $P_1|P_2|\dots|P_n$ where P_i are sorts. We refer to [80] for observers of structures and non-solids.

The above covered the prompts for describing external qualities. Prompts for describing internal qualities are: **observe_unique_identifier**, **observe_mereology** and **observe_attributes**. Section 8.5 can be summarised formally:

type observe_unique_identifier : $P \rightarrow \text{Text}$
<p>Narratives:</p> <ul style="list-style-type: none"> i. text on unique identifier: UI o. text on unique identifier observer: uid_E <p>Formalisation:</p> <ul style="list-style-type: none"> i. type UI o. value $\text{uid}_E: E \rightarrow \text{UI}$

type **observe_mereology**: $P \rightarrow \text{Text}$

Narratives:

- m. text on mereology: M
- o. text on mereology observer: mereo_E

Formalisation:

- m. **type** $M = \mathcal{E}(U_{1a}, \dots, U_{1c})$
- o. **value** $\text{mereo_E}: E \rightarrow M$

The choice of $\mathcal{E}(U_{1a}, \dots, U_{1c})$, that is, of the mereology of any one sort \mathcal{E} , depends on the aspects of the domain that its analyser cum describer wishes to study. That is, “one and the same domain” may give rise to different models each emphasizing their aspects.

type **observe_attributes**: $P \rightarrow \text{Text}$

Narratives:

- a. texts on attributes: A_1, \dots, A_k
- o. texts on attribute observers: $\text{attr_A}_1, \dots, \text{attr_A}_k$

Formalisation:

- a. **type** A_1, \dots, A_k
- o. **value** $\text{obs_A}_1: E \rightarrow A_1, \dots, \text{obs_A}_k: E \rightarrow A_k$

One and “seemingly” the same domain may give rise to different analyses & descriptions. Each of these emphasize different aspects. **Example: Road Net:** In one model of a road net emphasis may be on automobile traffic (aiming, eventually, at a road pricing system). In another model of “the same” road net emphasis may be on the topological layout (aiming, eventually, at its construction). In yet a third model “over” a road net emphasis may be on traffic control. For each such “road net” model the domain analyser cum describer selects different overlapping sets of attributes.

8.7 From Parts to Behaviours

It is often said “every noun can be verbed” and “every verb can be nouned”. That may be so. In any case we shall perform the following one-way *transcendental deduction*: “to every [endurant] physical part” “we shall associated a perdurant behaviour”. That deduction is “inspired” by the following observations: (i) *there is the train, as an enduring entity, as it stands, there, on the platform, statically observable, over time*; (ii) *there is the train, as a perdurant behaviour, as it “speeds” down the railway track, only a part of it visible at any given point and time*; and (iii) *there is the train, as a railway system attribute, as it “appears” in a time table; programmable*.

By an **action** we shall understand a function which, among its arguments, take a state and delivers an updated state, and where that action has been knowingly, willfully applied. By an **event** we shall understand a state change for which we do not seek its origin, i.e., who or what caused that state change. By a **behaviour** we shall understand a sequence of one or more actions, events and behaviours.

In Sect. 8.8 we shall formally summarise, cf. [80], that deduction. Since physical parts coexist — their translated behaviours operate concurrently. Since physical parts relate — these behaviours communicate. For that reason we shall express the part behaviours in terms of Hoare’s CSP [146, 148, 225, 233]. Hence we shall express communication via channels.

8.8 Perdurants

To simplify matters we shall just deal with artefacts. These are described in terms of their sorts, whether atomic, composite, or concrete (like ‘sets of’), and unique identifiers, mereology and attributes. Transcendentally deduced artefact behaviours are described in terms of their signatures and definition bodies. We shall now show how to relate all of the endurant descriptions with their perdurant counterpart; that is, how the transcendental deduction works! Each have a crucial rôle!

8.8.1 Behaviour Signatures

A behaviour signature, for part p , is of the form:

value

```
Name: ui:Ul →
      (st1,...,sts):Statics →
      me:Mereology →      [me = (ichs,iochs,ochs)]
      (ca1,...,cac):Programmables →
      in   Monitorables, in_Mereology,
      in,out in_out_Mereology,
      out  out_Mereology → Unit
```

We explain this signature: Name is an analyser cum describer chosen name, usually a meaningful mnemonic; Ul is the type of the unique identifier for the translated sort, i.e., of part p ; Statics designates the zero ($s=0$), one or more static attributes of part p ; Mereology designates the triplet mereology of the part, cf. last paragraph of Sect. 8.5.2; Programmables designates the zero ($c=0$), one or more programmable attributes of part p ; Monitorable designates zero, one or more **input** channel references designating monitorable attributes of part p ; in_Mereology designates zero, one or more **input** channel references designating the parts from whom part p “receives” input; in_out_Mereology designates zero, one or more **input/output** channel references designating the parts from whom part p “receives” input and to whom it “delivers” output; out_Mereology designates zero, one or more **output** channel references designating the parts to whom part p “delivers” output; and **Unit** designates that the behaviour goes on forever! Technicalities are given in [80], Sects. 7.4.3–7.4.4.

8.8.2 Behaviour Definition Bodies: \mathcal{B}_p

In general the signature expresses that behaviour Name(uid) evolves around (i) constant values whose type is given in Statics; (ii) input from monitorable attributes (of values “residing” in part p , but not otherwise expressible) are expressed in the body of the behaviour definition by the CSP input expression $\text{attr}_A?$ where A is a monitorable attribute of p ; (iii) input from topologically related parts, q , are expressed by $\text{ch}[ui_p,ui_q]?$; and (iv) output of values v to topologically related parts, q , are expressed by $\text{ch}[ui_p,ui_q]!v$. (v) In other words, the channel designations of the signature are of the form: $\text{attr}_{A_i}, \dots, \text{attr}_{A_j}$ and $\text{ch}[ui_p,ui_q]$. Further technicalities are given in [80], Sect. 7.4.5.

8.8.3 From Part Descriptions to Behaviour Definitions

Composite parts: **type** $\text{Translate}_p: P \rightarrow \text{RSL}^+ \text{Text}$

value

```
Translatep: P → RSL+Text
Translatep(p) ≡
  let ui = uid_P(p),      me = mereo_P(p),   Sects. 8.5.1,8.5.2
      sa = stat_attr_vals(p), ca = ctrl_attr_vals(p), Sect. 8.5.3
```

```

MT = mereo_typs(p), ST = stat_attr_typs(p),
CT = ctrl_attr_typs(p), IOR = calc_i_o_chn_refs(p),
IOD = calc_all_chn_dcls(p) in
⋈ channel IOD
value
   $\mathcal{M}_P: P\_UI \rightarrow ST \rightarrow MT \rightarrow CT \rightarrow IOR \text{ Unit}$ 
   $\mathcal{M}_P(ui)(sa)(me)(ca) \equiv \mathcal{B}_P(ui)(sa)(me)(ca)$ 
  , ⋈ TranslateP1(obs_endurant_sorts_E1(p))
  ⋈⋈ TranslateP2(obs_endurant_sorts_E2(p))
  ⋈⋈ ...
  ⋈⋈ TranslatePn(obs_endurant_sorts_En(p))
end

```

The above schema specifies the translation of composite parts into RSL^+Text , where RSL is the RAISE Specification Language, [131]. The $\llcorner \dots \lrcorner$ designate the texts, ..., as written.

Concrete parts: **type** Translate_P: $P \rightarrow RSL^+Text$

```

type
  Qs = Q-set
value
  qs:Q-set = obs_part_Qs(p)
  TranslateP(p) ≡
    let ui = uid_P(p), me = mereo_P(p),
        sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
        ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
        IOR = calc_i_o_chn_refs(p), IOD = calc_all_chn_dcls(p) in
    ⋈ channel IOD
    value
       $\mathcal{M}_P: UI \rightarrow ST \rightarrow CT \rightarrow IOR \text{ Unit}$ 
       $\mathcal{M}_P(ui)(sa)(me)(ca) \equiv \mathcal{B}_P(ui)(sa)(me)(ca) \lrcorner$ 
      {  $\llcorner, \lrcorner$  TranslateQ(q) | q:Q•q ∈ qs }
    end
end

```

Atomic parts: **type** Translate_P: $P \rightarrow RSL^+Text$

```

value
  TranslateP(p) ≡
    let ui = uid_P(p), me = mereo_P(p),
        sa = stat_attr_vals(p), ca = ctrl_attr_vals(p),
        ST = stat_attr_typs(p), CT = ctrl_attr_typs(p),
        IOR = calc_i_o_chn_refs(p), IOD = calc_all_chns(p) in
    ⋈ channel IOD
    value
       $\mathcal{M}_P: P\_UI \times MT \times ST \text{ PT } IOR \text{ Unit}$ 
       $\mathcal{M}_P(ui)(sa)(me)(ca) \equiv \mathcal{B}_P(ui)(sa)(me)(ca) \lrcorner$ 
    end
end

```

$\mathcal{B}_P(ui)(sa)(me)(ca)$ designate the “body” of the definition of behaviour $\mathcal{B}_P(ui)$. For details we refer to the **Core Behaviour Schema** of [80, Sect. 7.5].

8.8.4 Channel Declarations

Here we shall just mention that the above **Translate** schemas refer to **channels**. The channel declaration, in RSL, are of the form

- **channel** $ch[\{ui_p, ui_q\}]$: CH_MSG

where CH_MSG is a type expression for the values communicated over CSP channels. That is, the channel array indexes are two element sets of unique identifiers of relevant distinct parts as implied by their respective mereologies.

8.8.5 Concrete System

An instantiation of any given *universe of discourse*, *uod*, thus amounts to the parallel, ||, composition of behaviours, potentially one for each composite and each atomic part. [80, Sect. 7.6] illustrates an example.

8.9 Conclusion

In [71, Sect. 3.1.5] we elaborate extensively on the analysis & description process, while giving, in [71, Sect. 5.3], an extensive review of related work. In [80, Sect. 9, *Closing*] we discuss, extensively, the wider ramifications of the domain science and engineering approach of the present paper. [82] elaborates on issues of sorts, types and intents.

8.9.1 What Have We Achieved?

We have summarised an essence of Kai Sørlander's Philosophy [245]: recounted how, from a basis of the **inescapable meaning theory**, the concepts of *space*, *time* and *Newton's Laws* can be transcendently deduced, and from these the concepts of *living species: plants and animals*. And we have summarised the essence of [80]: an ontology of endurants and perdurants, discrete and continuous (non-solid) endurants, physical parts, living species and structures, natural parts and artefacts; and their internal qualities: unique identifiers, mereology and attributes. Finally we have shown, by a transcendental deduction, how discrete endurants can be "morphed" into perdurants, i.e., [in this paper] CSP behaviours whose signature can be derived from internal qualities of appropriate discrete endurants: unique identifiers, mereology and attributes. Throughout we have related the two areas: philosophy and computing.

The Philosophy aspect of this paper is new. That is, it is, to our knowledge, the first time a serious attempt has been made to strongly relate an area of the science of computing to philosophy. The domain science & engineering of [80] is also new. For the first time we see a straight line from the domain of artefact problems to their solution by computing [35, 45, 52, 55, 61, 78]. I find that remarkable. If you have a need for examples, please consult [80] and [85, twelve domain case studies].

8.9.2 Open Problems

General

We shall only focus on issues that relate to Sects. 8.3-8.8. The issue of **artefacts** is not dealt with specifically in Sørlander's Philosophy; and the issue of the **intent[s]** of *artefacts* is new. Further studies seem necessary in order to secure the inevitability of the distinction between discreteness and continuity, justify the presence of structures, and the distinction between atomic and composite parts. The transcendental deduction of endurants into perdurants may not exactly satisfy Kai Sørlander's strict criteria for such deductions.

Intentional Pull: Invariants

Intentional pull seems to relate very strongly to the notion of invariants. In [218, Item 4, pp. 4–5] Wolfgang Reisig identifies the issue of invariants, for example, of distributed discrete dynamic systems, such as we transcendently deduce them from composite artefacts, as a central characteristic of, and a hard problem for, such systems. It seems to us that it might be worthwhile to study the point made by Reisig by taking into account the philosophical basis that we have proposed in this paper. **Ex.21. Invariants of Distributed Systems:** We show some informal examples: **Simple Credit Card Systems:** For a system of credit cards (surrogates for owners), their honouring banks and accepting shops, one invariant could be: *the sum of cash with credit cards (i.e., their owners purses), banks, and shops remain a constant across purchase and refund operations!* **Simple Market System:** For a system of end-use customers (e.g. with credit cards) retailers and wholesalers, one invariant could be: the set of goods with end-use customers, retailers and wholesalers remain unchanged across customer, retailed and wholesale operations ■ The examples are ‘simple’ in that they, e.g., omit consideration of the advent of new automobiles on the road, new merchandise in wholesalers warehouses and the destruction of merchandise with end-users, retailers, etc.

In [218] Reisig suggests an intriguing list of aspects that appear to form main characteristics of our science.

8.9.3 Acknowledgment

I am grateful to Kai Sørlander for his patience and help in properly understanding his philosophy and for creating that philosophy [238, 239, 240, 241, 242, 243, 244, 245]: truly a remarkable feat — as also observed by Georg Henrik von Wright, Wittgenstein’s successor at Cambridge, England [en.wikipedia.org/wiki/Georg_Henrik_von_Wright] [238].

8.9.4 Acknowledgment

I am grateful to Kai Sørlander for his patience and help in properly understanding his philosophy and for creating that philosophy [238, 239, 240, 241, 242, 243, 244, 245]: truly a remarkable feat — as also observed by Georg Henrik von Wright, Wittgenstein’s successor at Cambridge, England, at age 32 [en.wikipedia.org/wiki/Georg_Henrik_von_Wright] [238].

Part **VI**

Conclusion

Summing Up

Each of Chapters 1–6 have their own closings. Here we summarise their conclusions.

9.1 What Have We Achieved ?

9.1.1 Chapter-by-Chapter Achievement Enumeration

Chapter 1, Pages 3–76: Domain Analysis & Description

The main contribution is that of introducing the domain analysis & description method: principles, techniques and tools. It was hinted at that the ontology for domain entities can be justified on philosophical grounds. A paper, [79], will expand considerably on this topic. Conventional software engineering previously began with requirements engineering. Now a predecessor phase has been “put” before that.

I consider this the major contribution of this monograph.

Chapter 2, Pages 77–106: Domain Facets

The main contribution is that of introducing the concept of domain facets and its manifestation: domain intrinsics, domain support technology, domain rules & regulations, domain scripts, domain license languages, domain management & organisation, and domain human behaviour.

Chapter 3, Pages 107–130: Towards Formal Models of Processes and Prompts

The contribution of this chapter is rather somewhat contrary to traditionalist thinking. Instead of formulating semantic domains for syntactic quantities we turn matters “upside-down”: from semantic entities we “derive” syntactic ones !

Chapter 4, Pages 131–151: To Every Manifest Mereology a CSP Expression

The contribution of this chapter is both traditional and novel. Traditional, in that we show how Casati and Varzi’s axiom system [104] for Leśniewski’s mereology, can be given a model in terms of the domain ontology sorts of Chapter 1. Novel, in that we show, for the first time, in 2009, how manifest mereologies, by transcendental deduction, can be “modelled” as CSP [148] processes.

Chapter 5, Pages 155–201: From Domains ... to Requirements ...

The contribution of this chapter is methodological. It is not a theory, but it is a set of principles and techniques for systematically “deriving” requirements prescriptions from domain descriptions. The principles include the separation of requirements concerns into *domain*, *interface* and *machine requirements*, and, within *domain requirements*, the novel concepts of *domain projection*, *instantiation*, *determination*, *extension* and *fitting*.

Chapter 6, Pages 205–214: Demos, Simulators, Monitors and Controllers

The contributions of this chapter are not of scientific nature. They are rather of a “pedagogical” engineering nature – in that they throw a different light on such notions as *demos*, *simulators*, *monitors* and *controllers* by relating these to relations between *domain descriptions*, *requirements prescriptions* and *software designs*.

9.1.2 Fulfillment of Aim**THE AIM OF THIS MONOGRAPH**

The aim of this monograph is twofold. To show that:

- (i) *domain science & engineering* is a possible, initial phase of software development;
- (ii) *domain science & engineering* is a worthwhile topic of research.

We support this claim as follows:

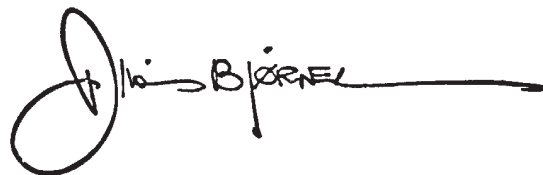
- (a) the concepts of *domain science* and *domain engineering* are new;
- (b) the terms *domain science* and *domain engineering* are well-defined;
- (c) *domain science* and *domain engineering* are given a foundation in this thesis;
- (d) and their rôle in software development is established.

Domain Science & Engineering casts a completely new light on Software Development.

We can now conclude:

AIM FULFILLED

We claim that the aim has been fulfilled.



*Dines Bjørner, November 24, 2019: 13:16
Fredsvvej 11, DK-2840 Holte, Denmark*

9.2 Acknowledgements

The author has worked on the topic of domain science and engineering since the early 1990s. Upon his retirement in 2007 he was able to devote full time to its study. The chapters of this thesis are all based on publications after 2007. Here I want to acknowledge the encouragement I have received from colleagues in Europe and Asia – for example by their giving me much needed interaction with PhD students. The list below includes references to either ■ books or domain case studies ■ [85] that were worked out during my stays (usually PhD lectures) at these colleagues:

- **Jin Song Dong**, July 2004 – June 2005, NUS, Singapore ■ [28]
- **Kokichi Futatsugi**, Feb. 2006 – Jan. 2007, JAIST, Kanazawa, Japan ■ [84, 36, 37, 38, 42, 43, 44]
- **Dominique Méry**, Oct. – Nov. 2007, Nancy, France
- **Wolfgang J. Paul**, March 2008, Saarbrücken, Germany
- **Bernhard K. Aichernig**, Oct. – Nov. 2008, Graz, Austria ■ [56]
- **Alan Bundy**, Sept. – Nov. 2009, Edinburgh, Scotland
- **Tetsuo Tamai**, Nov. – Dec. 2009, Tokyo, Japan ■ [49]
- **Jens Knoop**, April 2010, Vienna, Austria ■ [47]
- **Niklaj Nikitchenko**, May 2010, Kiev, Lviv, Odessa, Ukraine
- **Istenes Zoltán**, Oct. 2010, Budapest, Hungary
- **Andreas Hamfeldt**, Nov. 2010, Uppsala, Sweden
- **Sun Meng**, Nov. 2012, Beijing, China
- **Zhu HuBiao**, Dec. 2012, ECNU, Shanghai, China
- **Zhan NaiJun**, Dec. 2012, Beijing, China
- **Viktor Ivannikov**, April 2014, Moscow, Russia
- **Jin Song Dong**, May 2014, NUS, Singapore
- **José N. Oliveira**, May – June 2015, Braga, Portugal
- **Jens Knoop**, Oct. 2015, Vienna, Austria
- **Andreas Hamfeldt**, May 2016, Uppsala, Sweden ■ [63]
- **Magne Haveraaen**, Nov. 2016, Bergen, Norway ■ [67]
- **Otthein Herzog**, Sept., 2017, Tong Ji Univ., Shanghai, China ■ [95, 72]
- **Chin Wei Ngan**, NUS, Singapore
- **Zhu HuBiao**, 2018, ECNU, Shanghai, China ■ [74]
- **Mauro Pezze**, May 2019, Lugano, Switzerland
- **Dino Mandrioli**, May 2019, Milano, Italy

A Common Bibliography

Bibliography

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings and Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, England, 1996 and 2009.
2. Open Mobile Alliance. *OMA DRM V2.0 Candidate Enabler*. http://www.openmobilealliance.org/-release_program/drm_v2_0.html, Sep 2005.
3. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.
4. K. Araki et al., editors. *IFM 1999–2013: Integrated Formal Methods*,
 - 1st IFM, 1999: e-ISBN-13:978-1-1-4471-851-1, and Lecture Notes in Computer Science vols.:
 - 2nd IFM, 2000: LNCS 1945,
 - 3rd IFM, 2002: LNCS 2335,
 - 4th IFM, 2004: LNCS 2999,
 - 5th IFM, 2005: LNCS 3771,
 - 6th IFM, 2007: LNCS 4591,
 - 7th IFM, 2009: LNCS 5423,
 - 8th IFM, 2010: LNCS 6396,
 - 9th IFM, 2012: LNCS 7321 and
 - 10th IFM, 2013: LNCS 7940.
 . Springer, 1999–2013.
5. Alapan Arnab and Andrew Hutchison. Fairer Usage Contracts for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
6. Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
7. John Longshaw Austin. *How To Do Things With Words*. Oxford University Press, second edition, 1976.
8. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, January 2003. 570 pages, 14 tables, 53 figures; ISBN: 0521781760.
9. Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics as Ontology Languages for the Semantic Web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, pages 228–248. Springer, Heidelberg, 2005.
10. C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.
11. Ralph-Johan Back, Abo Akademi, J. von Wright, and F. B. Schneider. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.
12. Alain Badiou. *Being and Event*. Continuum, 2005. (L'être et l'événements, Edition du Seuil, 1988).
13. Jaco W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
14. V. Richard Benjamins and Dieter Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. <http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm>.
15. Yochai Benkler. Coase's Penguin, or Linux and the Nature of the Firm. *The Yale Law Journal*, 112, 2002.
16. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. EATCS Series: Texts in Theoretical Computer Science. Springer, 2004.

17. W.R. Bevier, W.A. Hunt Jr., J Strother Moore, and W.D. Young. An approach to system verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989. Special Issue on System Verification.
18. Thomas Bittner, Maureen Donnelly, and Barry Smith. Endurants and Perdurants in Directly Depicting Ontologies. *AI Communications*, 17(4):247–258, December 2004. IOS Press, in [219].
19. Dines Bjørner. A ProCoS Project Description. *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Proceedings: Intl. Conf. on AI & Robotics, Strebse Pleso, Slovakia, Nov. 5-9, 1989, North-Holland, Publ., Dept. of Computer Science, Technical University of Denmark, October 1989.*
20. Dines Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia*, pages 15–34. ACM Press, May 11–15 1992.
21. Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In *2nd Asia-Pacific Software Engineering Conference (APSEC '95)*. IEEE Computer Society, 6–9 December 1995. Brisbane, Queensland, Australia.
22. Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society.
23. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik, VDI-Gesellschaft für Fahrzeug- und Verkehrstechnik. Invited talk.
24. Dines Bjørner. Domain Models of "The Market" — in Preparation for E-Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Balcawski)*, The Netherlands, December 2002. Kluwer Academic Press. URL: <http://www2.imm.dtu.dk/~dibj/themarket.pdf>.
25. Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, Heidelberg, October 7–11 2003. Springer-Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003. URL: <http://www2.imm.dtu.dk/~dibj/zohar.pdf>.
26. Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki. URL: <http://www2.imm.dtu.dk/~dibj/ifac-dynamics.pdf>.
27. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
28. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [36, 42].
29. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
30. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.
31. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen. See [37, 43].
32. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
33. Dines Bjørner. A Container Line Industry Domain. Techn. report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, June 2007. URL: imm.dtu.dk/~db/container-paper.pdf.
34. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
35. Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer. URL: imm.dtu.dk/~dibj/montanari.pdf.
36. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.

37. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.
38. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.
39. Dines Bjørner. An Emerging Domain Science – A Rôle for Stanisław Leśniewski’s Mereology and Bertrand Russell’s Philosophy of Logical Atomism. *Higher-order and Symbolic Computation*, 2009.
40. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. Research Monograph (# 4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan, This Research Monograph contains the following main chapters:
 - 1 *On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38.
 - 2 *Possible Collaborative Domain Projects – A Management Brief*, pages 39–56.
 - 3 *The Rôle of Domain Engineering in Software Development*, pages 57–72.
 - 4 *Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106.
 - 5 *The Triptych Process Model – Process Assessment and Improvement*, pages 107–138.
 - 6 *Domains and Problem Frames – The Triptych Dogma and M.A.Jackson’s PF Paradigm*, pages 139–175.
 - 7 *Documents – A Rough Sketch Domain Analysis*, pages 179–200.
 - 8 *Public Government – A Rough Sketch Domain Analysis*, pages 201–222.
 - 9 *Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282.
 - 10 *Towards a Family of Script Languages – – Licenses and Contracts – An Incomplete Sketch*, pages 283–328.
 2009.
41. Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare, History of Computing* (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer. URL: imm.dtu.dk/~dibj/bjorner-hoare75-p.pdf.
42. Dines Bjørner. **Chinese: Software Engineering, Vol. 1: Abstraction and Modelling**. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
43. Dines Bjørner. **Chinese: Software Engineering, Vol. 2: Specification of Systems and Languages**. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
44. Dines Bjørner. **Chinese: Software Engineering, Vol. 3: Domains, Requirements and Software Design**. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.
45. Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.
46. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, 2(4):100–116, May 2010.
47. Dines Bjørner. On Development of Web-based Software: A Divertimento of Ideas and Suggestions. Technical, Technical University of Vienna, August–October 2010. URL: imm.dtu.dk/~dibj/wfdftp.pdf.
48. Dines Bjørner. The Tokyo Stock Exchange Trading Rules. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2010. URL: imm.dtu.dk/~db/todai/tse-1.pdf, [imm.dtu.-dk/~db/todai/tse-2.pdf](http://imm.dtu.dk/~db/todai/tse-2.pdf).
49. Dines Bjørner. The Tokyo Stock Exchange Trading Rules URL: himm.dtu.dk/~db/todai/tse-1.pdf, imm.dtu.dk/~db/todai/tse-2.pdf. R&D Experiment, Techn. Univ. of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, January, February 2010.
50. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.
51. Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, 2(3):100–120, June 2011.
52. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011. URL: imm.dtu.dk/~dibj/maurer-bjorner.pdf.
53. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His*

- 70th Anniversary., Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011. URL: imm.dtu.dk/~dibj/maurer-bjorner.pdf.
54. Dines Bjørner. Documents – a Domain Description. Experimental Research Report 2013-3, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
 55. Dines Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pages 159–177. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2013. (eds.: Justyna Zander and Pieter J. Mosterman).
 56. Dines Bjørner. Pipelines – a Domain. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013. URL: imm.dtu.dk/~dibj/pipe-p.pdf.
 57. Dines Bjørner. Road Transportation – a Domain Description. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013. URL: imm.dtu.dk/~dibj/road-p.pdf.
 58. Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.
 59. Dines Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, May 2014.
 60. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida and José Meseguer and Kazuhiro Ogata, editor, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014. URL: imm.dtu.dk/~dibj/2014/kanazawa/kanazawa-p.pdf.
 61. Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida and José Meseguer and Kazuhiro Ogata, editor, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014. URL: imm.dtu.dk/~dibj/2014/kanazawa/kanazawa-p.pdf.
 62. Dines Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May. , Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, December 2014. URL: imm.dtu.dk/~dibj/2014/assc-april-bw.pdf.
 63. Dines Bjørner. A Credit Card System: Uppsala Draft. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. URL: imm.dtu.dk/~dibj/2016/credit/accs.pdf.
 64. Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [61]. URL: imm.dtu.dk/~dibj/2016/process/process-p.pdf.
 65. Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [52]. URL: imm.dtu.dk/~dibj/2016/demos/faoc-demo.pdf.
 66. Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [35] URL: compute.dtu.dk/~dibj/2015/faoc-req/faoc-req.pdf.
 67. Dines Bjørner. Weather Information Systems: Towards a Domain Description. Technical Report: Experimental Research, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2016. URL: imm.dtu.dk/~dibj/2016/wis/wis-p.pdf.
 68. Dines Bjørner. Manifest Domains: Analysis & Description – A Philosophical Basis. , 2016–2017. URL: imm.dtu.dk/~dibj/2016/apb/daad-apb.pdf.
 69. Dines Bjørner. A Space of Swarms of Drones. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, December 2017. URL: imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf.
 70. Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. Online: 26 July 2016.
 71. Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, March 2017. First Online: 26 July 2016. DOI 10.1007/s00165-016-0385-z.
 72. Dines Bjørner. What are Documents? Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017. URL: imm.dtu.dk/~dibj/2017/docs/docs.pdf.
 73. Dines Bjørner. A Philosophy of Domain Science & Engineering – An Interpretation of Kai Sørlander’s Philosophy. Research Note, 95 pages, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, Spring 2018. URL: imm.dtu.dk/~dibj/2018/philosophy/filo.pdf.

74. Dines Bjørner. Container Terminals. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2018. An incomplete draft report; currently 60+ pages. URL: imm.dtu.dk/~dibj/2018/yangshan/maersk-pa.pdf.
 75. Dines Bjørner. Domain analysis & description - the implicit and explicit semantics problem. In Régine Laleau, Dominique Méry, Shin Nakajima, and Elena Troubitsyna, editors, Proceedings Joint Workshop on *Handling IMPLICIT and EXPLICIT knowledge in formal system development (IMPEX)* and *Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD)*, Xi'An, China, 16th November 2017, volume 271 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–23. Open Publishing Association, 2018.
 76. Dines Bjørner. Domain Facets: Analysis & Description. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, May 2018. Extensive revision of [45]. URL: imm.dtu.dk/~dibj/-2016/facets/faoc-facets.pdf.
 77. Dines Bjørner. Domain Science & Engineering – A Review of 10 Years Work and a Laudatio. In NaiJun Zhan and Cliff B. Jones, editors, *Symposium on Real-Time and Hybrid Systems – A Festschrift Symposium in Honour of Zhou ChaoChen*, LNCS 11180, pp. 6184. Springer Nature Switzerland AG URL: imm.dtu.dk/~dibj/2017/zcc/ZhouBjorner2017.pdf, June 2018.
 78. Dines Bjørner. To Every Manifest Domain a CSP Expression — A Rôle for Mereology in Computer Science. *Journal of Logical and Algebraic Methods in Programming*, 1(94):91–108, January 2018. URL: compute.dtu.dk/~dibj/2016/mereo/mereo.pdf.
 79. Dines Bjørner. Domain Analysis & Description – A Philosophy Basis. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2019. URL: imm.dtu.dk/~dibj/-2019/filo/main2.pdf.
 80. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. *ACM Trans. on Software Engineering and Methodology*, 29(2):..., April 2019. URL: imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.
 81. Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. *ACM Trans. on Software Engineering and Methodology*, 28(2), April 2019. 68 pages. URL: imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.
 82. Dines Bjørner. Domain Analysis & Description: Sorts, Types, Intents. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, November 2019. Paper for Klaus Havelund Festschrift, October 2020. URL: imm.dtu.dk/~dibj/2019/tyso/main2.pdf.
 83. Dines Bjørner. [84] *Chap. 10: Towards a Family of Script Languages – Licenses and Contracts – Incomplete Sketch*, pages 283–328. JAIST Press, March 2009.
 84. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.
 85. Dines Bjørner. Domain Case Studies:
 - 2019: *Container Line*, ECNU, Shanghai, China URL: imm.dtu.dk/~db/container-paper.pdf
 - 2018: *Documents*, TongJi Univ., Shanghai, China URL: imm.dtu.dk/~dibj/2017/docs/docs.pdf
 - 2017: *Urban Planning*, TongJi Univ., Shanghai, China URL: imm.dtu.dk/~dibj2017/up/urban-planning.pdf
 - 2017: *Swarms of Drones*, Inst. of Softw., Chinese Acad. of Sci., Peking, China URL: imm.dtu.dk/~dibj/2017/swarms/swarm-paper.pdf
 - 2013: *Road Transport*, Techn. Univ. of Denmark URL: imm.dtu.dk/~dibj/road-p.pdf
 - 2012: *Credit Cards*, Uppsala, Sweden URL: imm.dtu.dk/~dibj/2016/credit/accs.pdf
 - 2012: *Weather Information*, Bergen, Norway URL: imm.dtu.dk/~dibj/2016/wis/wis-p.pdf
 - 2010: *Web-based Transaction Processing*, Techn. Univ. of Vienna, Austria URL: imm.dtu.dk/~dibj/wdfftp.pdf
 - 2010: *The Tokyo Stock Exchange*, Tokyo Univ., Japan URL: imm.dtu.dk/~db/todai/tse-1.pdf, URL: imm.dtu.dk/~db/todai/tse-2.pdf
 - 2009: *Pipelines*, Techn. Univ. of Graz, Austria URL: imm.dtu.dk/~dibj/pipe-p.pdf
 - 2007: *A Container Line Industry Domain*, Techn. Univ. of Denmark URL: imm.dtu.dk/~dibj/container-paper.pdf
 - 2002: *The Market*, Techn. Univ. of Denmark URL: imm.dtu.dk/~dibj/themarket.pdf, [24]
 - 1995–2004: *Railways*, Techn. Univ. of Denmark - a compendium URL: imm.dtu.dk/~dibj/train-book.pdf
- Experimental research reports, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark.

86. Dines Bjørner. The Rôle of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.
87. Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.
88. Dines Bjørner, Chris W. George, and Søren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, P.O.Box 1299, Grand View, Texas 76050-1299, USA, 24–28 June 2002. Society for Design and Process Science. URL: <http://www2.imm.dtu.dk/~dibj/pasadena-25.pdf>.
89. Dines Bjørner and Klaus Havelund. 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities. In *FM 2014, Singapore, May 14-16, 2014*. Springer, 2014. Distinguished Lecture. URL: imm.dtu.dk/~dibj/2014/fm14-paper.pdf.
90. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
91. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
92. Dines Bjørner and Jørgen Fischer Nilsson. Algorithmic & Knowledge Based Methods — Do they “Unify” ? In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages 191–198. ICOT, June 1–5 1992.
93. Nikolaj Bjørner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16:227–270, 2000.
94. Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. In *Higher-Order Program Analysis*, June 2013. <http://hopa.cs.rhul.ac.uk/files/proceedings.html>.
95. Dines Bjørner. Urban Planning Processes. Research Note, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, July 2017. URL: imm.dtu.dk/~dibj/2017/up/urban-planning.pdf.
96. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Laurent Mauborgne Jerome Feret, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, pages 196–207, 2003 .
97. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
98. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
99. Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.
100. Nicholas Bunnin and E.P. Tsui-James, editors. *The Blackwell Companion to Philosophy*. Blackwell Companions to Philosophy. Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK, 1996.
101. F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., England, 2007.
102. Roberto Casati and Achille Varzi. Events. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, spring 2010 edition, 2010.
103. Roberto Casati and Achille C. Varzi, editors. *Events*. Ashgate Publishing Group – Dartmouth Publishing Co. Ltd., Wey Court East, Union Road, Farnham, Surrey, GU9 7PT, United Kingdom, 23 March 1996.
104. Roberto Casati and Achille C. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.
105. C.E.C. Digital Rights: Background, Systems, Assessment. Commission of The European Communities, Staff Working Paper, 2002. Brussels, 14.02.2002, SEC(2002) 197.
106. Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.
107. C. N. Chong, R. J. Corin, J. M. Doumen, S. Etalle, P. H. Hartel, Y. W. Law, and A. Tokmakoff. LicenseScript: a logical language for digital rights management. *Annals of telecommunications special issue on Information systems security*, 2006.

108. C. N. Chong, S. Etalle, and P. H. Hartel. Comparing Logic-based and XML-based Rights Expression Languages. In *Confederated Int. Workshops: On The Move to Meaningful Internet Systems (OTM)*, number 2889 in LNCS, pages 779–792, Catania, Sicily, Italy, 2003. Springer.
109. Cheun Ngen Chong, Ricardo Corin, and Sandro Etalle. LicenseScript: A novel digital rights languages and its semantics. In *Proc. of the Third International Conference WEB Delivering of Music (WEDEL-MUSIC'03)*, pages 122–129. IEEE Computer Society Press, 2003.
110. Jesper Vinther Christensen. *Specifying Geographic Information – Ontology, Knowledge Representation, and Formal Constraints*. Phd thesis, Technical University of Denmark, Computer Science and Engineering, DK 2800 Kgs. Lyngby, August 2007.
111. David R. Christiansen, Klaus Grue, Henning Niss, Peter Sestoft, and Kristján S. Sigtryggsson. Actulus Modeling Language - An actuarial programming language for life insurance and pensions. Technical Report, URL: edlund.dk/sites/default/files/Downloads/paper_actulus-modeling-language.pdf, Edlund A/S, Denmark, Bjerregårds Sidevej 4, DK-2500 Valby. (+45) 36 15 06 30. edlund@edlund.dk, <http://www.edlund.dk/en/insights/scientific-papers>, 2015. This paper illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety.
112. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
113. CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer-Verlag, 2004.
114. Patrick Cousot and Rhadia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL: Principles of Programming and Languages*, pages 238–252. ACM Press, 1977.
115. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
116. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993.
117. Donald Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.
118. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
119. Merlin Dorfman and Richard H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.
120. F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. Reprinted in [103, 1996], pp. 415-428.
121. ESA. Global Navigation Satellite Systems. Web, European Space Agency. http://en.wikipedia.org/wiki/Satellite_navigation.
122. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.
123. R. Falbo, G. Guizzardi, and K.C. Duarte. An Ontological Approach to Domain Engineering. In *Software Engineering and Knowledge Engineering*, Proceedings of the 14th international conference SEKE'02, pages 351–358, Ischia, Italy, July 15-19 2002. ACM.
124. David John Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.
125. Edward A. Feigenbaum and Pamela McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.
126. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
127. Martin Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.
128. Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.
129. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial-Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL-1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.
130. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999.

131. Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1992.
132. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1995.
133. J Paul Gibson and Dominique Méry. Explicit modelling of physical measures: From event-b to java. In Régine Laleau, Dominique Méry, Shin Nakajima, and Elena Troubitsyna, editors, Proceedings Joint Workshop on *Handling IMPLICIT and EXPLICIT knowledge in formal system development (IMPEX)* and *Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD)*, Xi'An, China, 16th November 2017, volume 271 of *Electronic Proceedings in Theoretical Computer Science*, pages 64–79. Open Publishing Association, 2018.
134. Don I. Good and William D. Young. Mathematical Methods for Digital Systems Development. In *VDM '91: Formal Software Development Methods*, pages 406–430. Springer-Verlag, October 1991. Volume 2.
135. T. Grötter, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, Dordrecht, 2002.
136. Carl A. Gunter, Elsa L. Gunter, Michael A. Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.
137. Carl A. Gunter, Stephen T. Weeks, and Andrew K. Wright. Models and Languages for Digital Rights. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 4034–4038, Maui, Hawaii, USA, January 2001. IEEE Computer Society Press.
138. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
139. P.M.S. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [103], pp. 429–447.
140. Joseph Y. Halpern and Vicky Weissman. A Formal Foundation for XrML. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.
141. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
142. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
143. M. Harsu. A Survey on Domain Engineering. Review, Institute of Software Systems, Tampere University of Technology, Finland, December 2002.
144. A.E. Haxthausen, J. Peleska, and S. Kinder. A formal approach for the construction and verification of railway control systems. *Formal Aspects of Computing*, 23:191–219, 2011.
145. Dan Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'). <http://pragprog.com/>, 2009.
146. Charles Anthony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.
147. Charles Anthony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.
148. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: <URL: usingcsp.com/cspbook.pdf> (2004).
149. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
150. Charles Anthony Richard Hoare. Communicating Sequential Processes. Published electronically: <URL: usingcsp.com/cspbook.pdf>, 2004. Second edition of [149]. See also <URL: usingcsp.com/>.
151. Gerard J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
152. Ted Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.
153. G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The Coq Proof Assistant - A tutorial - Version 7.1*, October 2001. <http://coq.inria.fr>.
154. ContentGuard Inc. XrML: Extensible rights Markup Language. <http://www.xrml.org>, 2000.
155. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
156. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

157. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.
158. Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.
159. Michael A. Jackson. Program Verification and System Dependability. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.
160. Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
161. Ingvar Johansson. Qualities, Quantities, and the Endurant-Perdurant Distinction in Top-Level Ontologies. In K.D. Althoff, A. Dengel, R. Bergmann, M. Nick, and Th. Roth-Berghofer, editors, *Professional Knowledge Management WM 2005*, volume 3782 of *Lecture Notes in Artificial Intelligence*, pages 543–550. Springer, 2005. 3rd Biennial Conference, Kaiserslautern, Germany, April 10-13, 2005, Revised Selected Papers.
162. Cliff B. Jones, Ian Hayes, and Michael A. Jackson. Deriving Specifications for Systems That Are Connected to the Physical World. In Cliff Jones, Zhiming Liu, and James Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer, 2007.
163. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. FODA: Feature-Oriented Domain Analysis. Feasibility Study CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. <http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm>.
164. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.
165. Andrew Kennedy. *Programming languages and dimensions*. PhD thesis, University of Cambridge, Computer Laboratory, April 1996. 149 pages: [URL: cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf](http://cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf). Technical report UCAM-CL-TR-391, ISSN 1476-298.
166. R.H. Koenen, J. Lacy, M. Mackay, and S. Mitchell. The long march to interoperable digital rights management. *Proceedings of the IEEE*, 92(6):883–897, June 2004.
167. Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., USA, 2002.
168. Søren Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.
169. Henry S. Leonard and Nelson Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.
170. W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1973, 1987. Two vols.
171. IPR Systems Pty Ltd. Open Digital Rights Language (ODRL). <http://odrl.net>, 2001.
172. E.C. Luschei. *The Logical Systems of Leśniewski*. North Holland, Amsterdam, The Netherlands, 1962.
173. Gordon E. Lyon. Information Technology: A Quick-Reference List of Organizations and Standards for Digital Rights Management. NIST Special Publication 500-241, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce, Oct 2002.
174. Tom Maibaum. Conservative Extensions, Interpretations Between Theories and All That. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 40–66, 1997.
175. Abraham Maslow. A Theory of Human Motivation. *Psychological Review*, 50(4):370–96, 1943. <http://psychclassics.yorku.ca/Maslow/motivation.htm>.
176. Abraham Maslow. *Motivation and Personality*. Harper and Row Publishers, 3rd ed., 1954.
177. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
178. John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
179. J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [202].
180. Neno Medvidovic and Edward Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: Abs[S/W], 5 March 2004.
181. Usama Mehmood, Radu Grosu, Ashish Tiwari, Nicola Paoletti, Shan Lin, Yang JunXing, Dung Phan, Scott D. Stoller, and Scott A. Smolka. *Declarative vs Rule-based Control for Flocking Dynamics*. In *Proceedings of ACM/SIGAPP Symposium on Applied Computing (SACC 2018)*. ACM Press, April 9–13, 2018. 8 pages.

182. D.H. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.
183. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.
184. Merriam Webster Staff. Online Dictionary: <http://www.m-w.com/home.htm>, 2004. Merriam-Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.
185. Erik Mettala and Marc H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.
186. S. Michiels, K. Verslype, W. Joosen, and B. De Decker. Towards a Software Architecture for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
187. C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.
188. D. Mulligan and A. Burstein. Implementing copyright limitations in rights expression languages. In *Proc. of 2002 ACM Workshop on Digital Rights Management*, volume 2696 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, 2002.
189. Deirdre K. Mulligan, John Han, and Aaron J. Burstein. How DRM-Based Content Delivery Systems Disrupt Expectations of “Personal Use”. In *Proc. of The 3rd International Workshop on Digital Rights Management*, pages 77–89, Washington DC, USA, Oct 2003. ACM.
190. James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions of Software Engineering*, SE-10(5), September 1984.
191. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
192. Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.
193. Ernst Rüdiger Olderog, Anders Peter Ravn, and Rafael Wisniewski. Linking Discrete and Continuous Models, Applied to Traffic Maneuvers. In Jonathan Bowen, Michael Hinchey, and Ernst Rüdiger Olderog, editors, *BCS FACS – ProCoS Workshop on Provably Correct Systems*, Lecture Notes in Computer Science. Springer, 2016.
194. Reza Olfati-Saber. Flocking for Multi-agent Dynamic Systems: Algorithms and Theory. *IEEE Transactions on Automatic Control*, 51(3):401–420, 13 March 2006. <http://ieeexplore.ieee.org/document/1605401/>; DOI: 10.1109/TAC.2005.864190; Thayer School of Engineering, Dartmouth College, Hanover, NH, USA.
195. Leon Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
196. Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
197. Christine Paulin-Mohring. Modelisation of timed automata in Coq. In N. Kobayashi and B. Pierce, editors, *Theoretical Aspects of Computer Software (TACS'2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 298–315. Springer-Verlag, 2001.
198. Christopher Peterson and Martin E.P. Seligman. *Character strengths and virtues: A handbook and classification*. Oxford University Press, 2004.
199. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice-Hall, 2nd edition, 2001.
200. Chia-Yi Tony Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.
201. K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.
202. Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.
203. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw-Hill, 5th edition, 1981–2001.
204. Rubén Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.
205. Rubén Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
206. Rubén Prieto-Díaz and Guillermo Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
207. Arthur Prior. *Changes in Events and Changes in Things*, chapter in [202]. Oxford University Press, 1993.

208. Arthur N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.
209. Arthur N. Prior. *Formal Logic*. Clarendon Press, Oxford, UK, 1955.
210. Arthur N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.
211. Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.
212. Arthur N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.
213. Riccardo Pucella and Vicky Weissman. A Logic for Reasoning about Digital Rights. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.
214. Riccardo Pucella and Vicky Weissman. A Formal Foundation for ODRL. In *Proc. of the Workshop on Issues in the Theory of Security (WIST'04)*, 2004.
215. Martin Pěnička, Alben Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. URL: <http://www2.imm.dtu.dk/~dibj/martin.pdf>.
216. A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.
217. Wolfgang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
218. Wolfgang Reisig. Informatik als Wissenschaft. Essay, Humboldt-Universität zu Berlin, Institut für Informatik, Rudower Chaussee 25, D-12489 Berlin, Germany, August 2019.
219. Jochen Renz and Hans W. Guesgen, editors. *Spatial and Temporal Reasoning*, volume 14, vol. 4, Journal: AI Communications, Amsterdam, The Netherlands, Special Issue. IOS Press, December 2004.
220. Craig Reynolds. *Flocks, Herds and Schools: A Distributed Behavioral Model*. *SIGGRAPH Computer Graphics*, 21(4), August 1987. <https://doi.org/10.1145/37402.37406>.
221. Craig Reynolds. *Steering Behaviors for Autonomous Characters*. In *Proceedings of Game Developers Conference*, pages 763–782, 1999.
222. Craig Reynolds. OpenSteer, *Steering Behaviours for Autonomous Characters*, 2004. <http://opensteer.sourceforge.net>.
223. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
224. Gerald Rochelle. *Behind time: The incoherence of time and McTaggart's atemporal replacement*. Avebury series in philosophy. Ashgate, Brookfield, Vt., USA, 1998. vii + 221 pages.
225. A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. URL: <http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf>.
226. Douglas T. Ross. Computer-aided design. *Commun. ACM*, 4(5):41–63, 1961.
227. Douglas T. Ross. Toward foundations for the understanding of type. In *Proceedings of the 1976 conference on Data: Abstraction, definition and structure*, pages 63–65, New York, NY, USA, 1976. ACM. <http://doi.acm.org/10.1145/800237.807120>.
228. Douglas T. Ross and J. E. Ward. Investigations in computer-aided design for numerically controlled production. Final Technical Report ESL-FR-351, , May 1968. 1 December 1959 – 3 May 1967. Electronic Systems Laboratory Electrical Engineering Department, MIT, Cambridge, Massachusetts 02139.
229. Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
230. Pamela Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, Apr 2003.
231. Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Semantics and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.
232. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
233. Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.
234. John R. Searle. *Speech Act*. CUP, 1969.
235. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
236. Barry Smith. Mereotopology: A Theory of Parts and Boundaries. *Data and Knowledge Engineering*, 20:287–303, 1996.
237. Ian Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.

238. Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, 1994. 168 pages.
239. Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, 1997. 200 pages.
240. Kai Sørlander. *Om Menneskerettigheder*. Rosinante, 2000. 171 pages.
241. Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, 2002. 187 pages.
242. Kai Sørlander. *Forsvaret for Rationaliteten*. Informations Forlag (Publ.), 2008. 232 pages.
243. Kai Sørlander. *Den Politiske Forpligtelse – Det Filosofiske Fundament for Demokratisk Stillingtagen*. Informations Forlag, 2011. 280 pages.
244. Kai Sørlander. *Fornuftens Skæbne – Tanker om Menneskets Vilkår*. Informations Forlag, 2014. 238 pages.
245. Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, 2016. 233 pages.
246. Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
247. J.T.J. Srzednicki and Z. Stachniak, editors. *Leśniewski's Lecture Notes in Logic*. Dordrecht, 1988.
248. Staff of Encyclopædia Britannica. Encyclopædia Britannica. Merriam Webster/Britannica: Access over the Web: <http://www.eb.com:180/>, 1999.
249. Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany. URL: <http://www2.imm.dtu.dk/~dibj/albena.pdf>.
250. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
251. Will Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.
252. Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methodology, and Philosophy of Science (Editor: Jaakko Hintikka)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
253. F. Van der Rhee, H.R. Van Nauta Lemke, and J.G. Dukman. Knowledge based fuzzy control of systems. *IEEE Trans. Autom. Control*, 35(2):148–155, February 1990.
254. Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
255. Carl von Linné. *An Introduction to the Science of Botany*. Lipsiae [Leipzig]: Impensis Godofr. Kiesewetteri, 1748.
256. H. Wang, J. S. Dong, and J. Sun. Reasoning Support for Semantic Web Ontology Family Languages Using Alloy. *International Journal of Multiagent and Grid Systems*, IOS Press, 2(4):455–471, 2006.
257. Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, 3 vols. Cambridge University Press, 1910, 1912, and 1913. Second edition, 1925 (Vol. 1), 1927 (Vols 2, 3), also Cambridge University Press, 1962.
258. George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, summer 2012 edition, 2012.
259. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.
260. James Charles Paul Woodcock and James Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
261. JianWen Xiang and Dines Bjørner. The Electronic Media Industry: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
262. Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1991.
263. Edward N. Zalta. The Stanford Encyclopedia of Philosophy. 2016. Principal Editor: <https://plato.stanford.edu/>.
264. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real-time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

Experimental Case Studies

- Appendix A **Credit Cards** Pages 279–287
- Appendix B **Mereorological Information** Pages 289–301
- Appendix C **Pipelines** Pages 303–320
- Appendix D **Documents** Pages 321–344
- Appendix E **Urban Planning** Pages 345–401
- Appendix F **Swarms of Drones** Pages 403–438

A

Credit Card Systems

Summary

This report presents an attempt at a model of a credit card system.

A.1 Introduction

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse. I might “repair” this shortness if told so. A reference is made to my paper: [70, Manifest Domains: Analysis & Description]. That paper can be found on the Internet: <http://www2.compute.dtu.dk/~dibj/2015/faoc/faoc-bjorner.pdf>.

Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modelled) moves from keying in security codes to eye iris “prints”, and/or finger prints or voice prints or combinations thereof.

This document abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

A.2 Endurants

A.2.1 Credit Card Systems

511 Credit card systems, *ccs:CCS*,¹ consists of three kinds of parts:
512 an assembly, *cs:CS*, of credit cards³,
513 an assembly, *bs:BS*, of banks, and
514 an assembly, *ss:SS*, of shops.

type

511 CCS

¹ The composite part *CS* can be thought of as a credit card company, say VISA². The composite part *BS* can be thought of as a bank society, say BBA: British Banking Association. The composite part *SS* can be thought of as the association of retailers, say bira: British Independent Retailers Association. The model does not prevent “shops” from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case *SS* would be some appropriate association.

³ We “equate” credit cards with their holders.

512 CS

513 BS

514 SS

value512 **obs_part_CS**: CCS \rightarrow CS513 **obs_part_BS**: CCS \rightarrow BS514 **obs_part_SS**: CCS \rightarrow SS515 There are credit cards, $c:C$, banks $b:B$, and shops $s:S$.516 The credit card part, $cs:CS$, abstracts a set, $soc:Cs$, of card.517 The bank part, $bs:BS$, abstracts a set, $sob:Bs$, of banks.518 The shop part, $ss:SS$, abstracts a set, $sos:Ss$, of shops.**type**

515 C, B, S

516 Cs = C-set

517 Bs = B-set

518 Ss = S-set

value516 **obs_part_CS**: CS \rightarrow Cs, **obs_part_Cs**: CS \rightarrow Cs517 **obs_part_BS**: BS \rightarrow Bs, **obs_part_Bs**: BS \rightarrow Bs518 **obs_part_SS**: SS \rightarrow Ss, **obs_part_Ss**: SS \rightarrow Ss519 Assemblers of credit cards, banks and shops have unique identifiers, $csi:CSI$, $bsi:BSI$, and $ssi:SSI$.520 Credit cards, banks and shops have unique identifiers, $ci:CI$, $bi:BI$, and $si:SI$.

521 One can define functions which extract all the

522 unique credit card,

523 bank and

524 shop identifiers from a credit card system.

519 CSI, BSI, SSI

520 CI, BI, SI

value519 **uid_CS**: CS \rightarrow CSI, **uid_BS**: BS \rightarrow BSI, **uid_SS**: SS \rightarrow SSI,520 **uid_C**: C \rightarrow CI, **uid_B**: B \rightarrow BI, **uid_S**: S \rightarrow SI,522 **xtr_CIs**: CCS \rightarrow CI-set522 $xtr_CIs(ccs) \equiv \{\mathbf{uid_C}(c) \mid c:C \cdot c \in \mathbf{obs_part_Cs}(\mathbf{obs_part_CS}(ccs))\}$ 523 **xtr_BIs**: CCS \rightarrow BI-set523 $xtr_BIs(ccs) \equiv \{\mathbf{uid_B}(s) \mid b:B \cdot b \in \mathbf{obs_part_Bs}(\mathbf{obs_part_BS}(ccs))\}$ 524 **xtr_SIs**: CCS \rightarrow SI-set524 $xtr_SIs(ccs) \equiv \{\mathbf{uid_S}(s) \mid s:S \cdot s \in \mathbf{obs_part_Ss}(\mathbf{obs_part_SS}(ccs))\}$

525 For all credit card systems it is the case that

526 all credit card identifiers are distinct from bank identifiers,

527 all credit card identifiers are distinct from shop identifiers,

528 all shop identifiers are distinct from bank identifiers,

axiom525 $\forall ccs:CCS \cdot$ 525 **let** $cis=xtr_CIs(ccs)$, $bis=xtr_BIs(ccs)$, $sis = xtr_SIs(ccs)$ **in**526 $cis \cap bis = \{\}$


```

527   $\wedge$  cis  $\cap$  sis = {}
528   $\wedge$  sis  $\cap$  bis = {} end

```

A.2.2 Credit Cards

- 529 A credit card has a mereology which “connects” it to any of the shops of the system and to exactly one bank of the system,
530 and some attributes — which we shall presently disregard.
531 The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:
532 The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and
533 the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

type

```
529. CM = SI-set  $\times$  BI
```

value

```

529. obs_mereo_CM: C  $\rightarrow$  CM
531 wf_CM_of_C: CCS  $\rightarrow$  Bool
531 wf_CM_of_C(ccs)  $\equiv$ 
529   let bis=xtr_BIs(ccs), sis=xtr_SIs(ccs) in
529      $\forall$  c:C·c  $\in$  obs_part_Cs(obs_part_CS(ccs))  $\Rightarrow$ 
529       let (ccsis,bi)=obs_mereo_CM(c) in
532         ccsis  $\subseteq$  sis
533          $\wedge$  bi  $\in$  bis
529   end end

```

A.2.3 Banks

Our model of banks is (also) very limited.

- 534 A bank has a mereology which “connects” it to a subset of all credit cards and a subset of all shops,
535 and, as attributes:
536 a cash register, and
537 a ledger.
538 The ledger records for every card, by unique credit card identifier,
539 the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,
540 the dates-of-issue and -expiry of the credit card, and
541 the name, address, and other information about the credit card holder.
542 The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:
543 the bank mereology’s
544 must list a subset of the credit card identifiers and a subset of the shop identifiers.

type

```

534 BM = CI-set  $\times$  SI-set
536 CR = Bal
537 LG = CI  $\rightarrow$  (Bal  $\times$  DoI  $\times$  DoE  $\times$  ...)

```

```

539 Bal = Int
value
534 obs_mereo_B: B → BM
536 attr_CR: B → CR
537 attr_LG: B → LG
542 wf_BM_B: CCS → Bool
542 wf_BM_B(ccs) ≡
542   let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in
542   ∀ b:B • b ∈ obs_part_Bs(obs_part_BS(ccs)) in
543     let (cis, sis) = obs_mereo_B(b) in
544       cis ⊆ ∀ cis ∧ sis ⊆ allsis end end

```

A.2.4 Shops

545 The mereology of a shop is a pair: a unique bank identifiers, and a set of unique credit card identifiers.
546 The mereology of a shop
547 must list a bank of the credit card system,
548 band a subset (or all) of the unique credit identifiers.

We omit treatment of shop attributes.

```

type
545 SM = CI-set × BI
value
545 obs_mereo_S: S → SM
546 wf_SM_S: CCS → Bool
546 wf_SM_S(ccs) ≡
546   let allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) in
546   ∀ s:S • s ∈ obs_part_Ss(obs_part_SS(ccs)) ⇒
546     let (cis, bi) = obs_mereo_S(s) in
547       bi ∈ allbis
548       ∧ cis ⊆ allcis
546   end end

```

A.3 Perdurants

A.3.1 Behaviours

549 We ignore the behaviours related to the *CCS*, *CS*, *BS* and *SS* parts.
550 We therefore only consider the behaviours related to the *Cs*, *Bs* and *Ss* parts.
551 And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, *crd*, all the bank, *bnk*, and all the shop, *shp*, behaviours.

```

value
549 ccs:CCS
549 cs:CS = obs_part_CS(ccs),
549 uics:CSI = uid_CS(cs),
549 bs:BS = obs_part_BS(ccs),
549 uibs:BSI = uid_BS(bs),
549 ss:SS = obs_part_SS(ccs),

```

```

549 uiss:SSI = uid_SS(ss),
550 socs:Cs = obs_part-Cs(cs),
550 sobs:Bs = obs_part-Bs(bs),
550 soss:Ss = obs_part-Ss(ss),

```

value

```

551 sys: Unit → Unit,
549 sys() ≡
551   cardsuics(obs_mereo_CS(cs),...)
551   || || {crduid_C(c)(obs_mereo_C(c))|c:C•c ∈ socs}
551   || banksuibs(obs_mereo_BS(bs),...)
551   || || {bnkuid_B(b)(obs_mereo_B(b))|b:B•b ∈ sobs}
551   || shopsuiss(obs_mereo_SS(ss),...)
551   || || {shpuid_S(s)(obs_mereo_S(s))|s:S•s ∈ soss},
549 cardsuics(...) ≡ skip,
549 banksuibs(...) ≡ skip,
549 shopsuiss(...) ≡ skip

```

axiom skip || behaviour(...) ≡ behaviour(...)

A.3.2 Channels

552 Credit card behaviours interact with bank (each with one) and many shop behaviours.

553 Shop behaviours interact with bank (each with one) and many credit card behaviours.

554 Bank behaviours interact with many credit card and many shop behaviours.

The inter-behaviour interactions concern:

555 between credit cards and banks: withdrawal requests as to a sufficient, mk_Wdr(am), balance on the credit card account for buying am:AM amounts of goods or services, with the bank response of either is_OK() or is_NOK(), or the revoke of a card;

556 between credit cards and shops: the buying, for an amount, am:AM, of goods or services: mk_Buy(am), or the refund of an amount;

557 between shops and banks: the deposit of an amount, am:AM, in the shops' bank account: mk_Depost(ui,am) or the removal of an amount, am:AM, from the shops' bank account: mk_Removl(bi,si,am)

channel

```

552 {ch_cb[ci,bi]|ci:CI,bi:BI•ci ∈ cis ∧ bi ∈ bis}:CB_Msg
553 {ch_cs[ci,si]|ci:CI,si:SI•ci ∈ cis ∧ si ∈ sis}:CS_Msg
554 {ch_sb[si,bi]|si:SI,bi:BI•si ∈ sis ∧ bi ∈ bis}:SB_Msg
555 CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | ...
556 CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM) | ...
557 SB_Msg == Depost | Removl | ...
557 Depost == mk_Dep((ci:CI|si:SI),am:aM) |
557 Removl == mk_Rem(bi:BI,si:SI,am:aM)

```

A.3.3 Behaviour Interactions

558 The credit card initiates

a buy transactions

i [1.Buy] by enquiring with its bank as to sufficient purchase funds (am:aM);

- ii [2.Buy] if NOK then there are presently no further actions; if OK
 - iii [3.Buy] the credit card requests the purchase from the shop – handing it an appropriate amount;
 - iv [4.Buy] finally the shop requests its bank to deposit the purchase amount into its bank account.
- b refund transactions
- i [1.Refund] by requesting such refunds, in the amount of $am:aM$, from a[ny] shop; whereupon
 - ii [2.Refund] the shop requests its bank to move the amount $am:aM$ from the shop’s bank account
 - iii [3.Refund] to the credit card’s account.

Thus the three sets of behaviours, *crd*, *bnk* and *shp* interact as sketched in Fig. A.1.

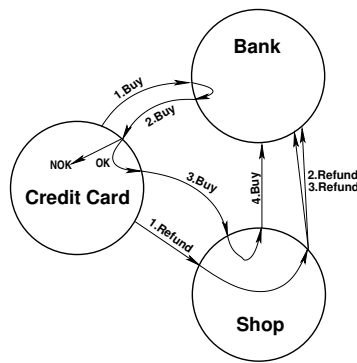


Fig. A.1. Credit Card, Bank and Shop Behaviours

[1.Buy]	Item 564, Pg.285 Item 573, Pg.286	card $ch_cb[ci,bi]!mk_Wdrw(am)$ (shown as ... three lines down) and bank $mk_Wdrw(ci,am)=\square\{ch_cb[bi,bi]? ci:Cl\cdot ci \in cis\}$.
[2.Buy]	Items 566-567, Pg.285 Item 564, Pg.285	bank $ch_cb[ci,bi]!lis_N[OK]OK()$ and shop $(...;ch_cb[ci,bi]?)$.
[3.Buy]	Item 566, Pg.285 Item 588, Pg.287	card $ch_cs[ci,si]!mk_Buy(am)$ and shop $mk_Buy(am)=\square\{ch_cs[ci,si]? ci:Cl\cdot ci \in cis\}$.
[4.Buy]	Item 589, Pg.287 Item 578, Pg.286	shop $ch_sb[si,bi]!mk_Dep(si,am)$ and bank $mk_Dep(si,am)=\square\{ch_cs[ci,si]? si:Sl\cdot si \in sis\}$.
[1.Refund]	Item 570, Pg.285 Item 589, Pg.287	card $ch_cs[ci,si]!mk_Ref((ci,si),am)$ and shop $(si,mk_Ref(ci,am))=\square\{si',ch_sb[si,bi]? si,si':Sl\cdot\{si,si'\} \subseteq sis \wedge si=si'\}$.
[2.Refund]	Item 593, Pg.287 Item 582, Pg.286	shop $ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am)$ and bank $(si,mk_Ref(cbi,(ci,am)))=\square\{(si',ch_sb[si,bi]?) si,si':Sl\cdot\{si,si'\} \subseteq sis \wedge si=si'\}$.
[3.Refund]	Item 594, Pg.287 Item 583, Pg.286	shop $ch_sb[si,sbi]!mk_Wdr(si,am)$ end and bank $(si,mk_Wdr(ci,am))=\square\{(si',ch_sb[si,bi]?) si,si':Sl\cdot\{si,si'\} \subseteq sis \wedge si=si'\}$

A.3.4 Credit Card

- 559 The credit card behaviour, *crd*, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, *crd*, accepts inputs from and offers outputs to the bank, *bi*, and any of the shops, $si \in sis$.
- 560 The credit card behaviour, *crd*, non-deterministically, internally “cycles” between buying and getting refunds.

value

559 $\text{crd}_{ci:CI}: (bi, sis):CM \rightarrow \mathbf{in, out} \text{ ch_cb}[ci, bi], \{\text{ch_cs}[ci, si] | si:SI \cdot si \in sis\} \mathbf{Unit}$
 559 $\text{crd}_{ci}(bi, sis) \equiv (\text{buy}(ci, (bi, sis)) \sqcap \text{ref}(ci, (bi, sis))) ; \text{crd}_{ci}(ci, (bi, sis))$

561 By $am:AM$ we mean an amount of money, and by $si:SI$ we refer to a shop in which we have selected a number or goods or services (not detailed) costing $am:AM$.

562 The buyer action is simple.

563 The amount for which to buy and the shop from which to buy are selected (arbitrarily).

564 The credit card (holder) withdraws $am:AM$ from the bank, if sufficient funds are available⁴.

565 The response from the bank

566 is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,

567 or the response is “not OK”, and the transaction is skipped.

type

561 $AM = \mathbf{Int}$

value

562 $\text{buy}: ci:CI \times (bi, sis):CM \rightarrow$
 562 $\mathbf{in, out} \text{ ch_cb}[ci, bi] \mathbf{out} \{\text{ch_cs}[ci, si] | si:SI \cdot si \in sis\} \mathbf{Unit}$
 562 $\text{buy}(ci, (bi, sis)) \equiv$
 563 $\mathbf{let} \text{ am}:aM \cdot \text{am} > 0, si:SI \cdot si \in sis \mathbf{in}$
 564 $\mathbf{let} \text{ msg} = (\text{ch_cb}[ci, bi]!\text{mk_Wdrw}(\text{am}); \text{ch_cb}[ci, bi]?) \mathbf{in}$
 565 $\mathbf{case} \text{ msg} \mathbf{of}$
 566 $\text{is_OK}() \rightarrow \text{ch_cs}[ci, si]!\text{mk_Buy}(\text{am}),$
 567 $\text{is_NOK}() \rightarrow \mathbf{skip}$
 562 $\mathbf{end\ end\ end}$

568 The refund action is simple.

569 The credit card [handler] requests a refund $am:AM$

570 from shop $si:SI$.

This request is handled by the shop behaviour’s sub-action *ref*, see lines 586.–595. page 287.

value

568 $\text{rfu}: ci:CI \times (bi, sis):CM \rightarrow \mathbf{out} \{\text{ch_cs}[ci, si] | si:SI \cdot si \in sis\} \mathbf{Unit}$
 568 $\text{rfu}(ci, (bi, sis)) \equiv$
 569 $\mathbf{let} \text{ am}:AM \cdot \text{am} > 0, si:SI \cdot si \in sis \mathbf{in}$
 570 $\text{ch_cs}[ci, si]!\text{mk_Ref}(bi, (ci, si), \text{am})$
 568 \mathbf{end}

A.3.5 Banks

571 The bank behaviour, *bnk*, takes the bank’s unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, *bnk*, accepts inputs from and offers outputs to the any of the credit cards, $ci \in cis$, and any of the shops, $si \in sis$.

572 The bank behaviour non-deterministically externally chooses to accept either ‘withdraw’al requests from credit cards or ‘deposit’ requests from shops or ‘refund’ requests from credit cards.

⁴ First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

value

```

571  $\text{bnk}_{bi:BI}: (\text{cis}, \text{sis}): \text{BM} \rightarrow (\text{LG} \times \text{CR}) \rightarrow$ 
571   in, out  $\{\text{ch\_cb}[ci, bi] \mid ci: \text{Cl} \cdot ci \in \text{cis}\} \{\text{ch\_sb}[si, bi] \mid si: \text{Sl} \cdot si \in \text{sis}\}$  Unit
571  $\text{bnk}_{bi}((\text{cis}, \text{sis}))(\text{lg}: (\text{bal}, \text{doi}, \text{doe}, \dots), \text{cr}) \equiv$ 
572    $\text{wdrw}(bi, (\text{cis}, \text{sis}))(\text{lg}, \text{cr})$ 
572    $\square \text{depo}(bi, (\text{cis}, \text{sis}))(\text{lg}, \text{cr})$ 
572    $\square \text{refu}(bi, (\text{cis}, \text{sis}))(\text{lg}, \text{cr})$ 

```

573 The ‘withdraw’ request, wdrw , (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, $\text{mk_Wdrw}(ci, \text{am})$, with the identity of the credit card.

574 If the requested amount (to be withdrawn) is not within balance on the account

575 then we, at present, refrain from defining an outcome (**chaos**), whereupon the bank behaviour is resumed with no changes to the ledger and cash register;

576 otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn; whereupon the bank behaviour is resumed notifying a lower balance and ‘withdraws’ the monies from the cash register.

value

```

572  $\text{wdrw}: bi: \text{Bl} \times (\text{cis}, \text{sis}): \text{BM} \rightarrow (\text{LG} \times \text{CR}) \rightarrow$  in, out  $\{\text{ch\_cb}[bi, ci] \mid ci: \text{Cl} \cdot ci \in \text{cis}\}$  Unit
572  $\text{wdrw}(bi, (\text{cis}, \text{sis}))(\text{lg}, \text{cr}) \equiv$ 
573   let  $\text{mk\_Wdrw}(ci, \text{am}) = \square \{\text{ch\_cb}[ci, bi] \mid ci: \text{Cl} \cdot ci \in \text{cis}\}$  in
572   let  $(\text{bal}, \text{doi}, \text{doe}) = \text{lg}(ci)$  in
574   if  $\text{am} > \text{bal}$ 
575     then  $(\text{ch\_cb}[ci, bi] \text{!is\_NOK}()); \text{bnk}_{bi}(\text{cis}, \text{sis})(\text{lg}, \text{cr})$ 
576     else  $(\text{ch\_cb}[ci, bi] \text{!is\_OK}()); \text{bnk}_{bi}(\text{cis}, \text{sis})(\text{lg} \dagger [ci \mapsto (\text{bal} - \text{am}, \text{doi}, \text{doe})], \text{cr} - \text{am})$  end
571   end end

```

The ledger and cash register attributes, lg, cr , are programmable attributes. Hence they are modeled as separate function arguments.

577 The deposit action is invoked, either by a shop behaviour, when a credit card [holder] buy’s for a certain amount, $\text{am}: \text{AM}$, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence $\text{am}: \text{AM}$, is to be inserted into the shops’ bank account, $si: \text{Sl}$, or on behalf of a credit card [i.e., a customer], hence $\text{am}: \text{AM}$, is to be inserted into the credit card holder’s bank account, $si: \text{Sl}$.

578 The message, $\text{ch_cs}[ci, si]?$, received from a credit card behaviour is either concerning a buy [in which case i is a $ci: \text{Cl}$, hence sale, or a refund order [in which case i is a $si: \text{Sl}$].

579 In either case, the respective bank account is ‘upped’ by $\text{am}: \text{AM}$ – and the bank behaviour is resumed.

value

```

577  $\text{deposit}: bi: \text{Bl} \times (\text{cis}, \text{sis}): \text{BM} \rightarrow (\text{LG} \times \text{CR}) \rightarrow$ 
577   in, out  $\{\text{ch\_sb}[bi, si] \mid si: \text{Sl} \cdot si \in \text{sis}\}$  Unit
577  $\text{deposit}(bi, (\text{cis}, \text{sis}))(\text{lg}, \text{cr}) \equiv$ 
578   let  $\text{mk\_Dep}(si, \text{am}) = \square \{\text{ch\_cs}[ci, si] \mid si: \text{Sl} \cdot si \in \text{sis}\}$  in
577   let  $(\text{bal}, \text{doi}, \text{doe}) = \text{lg}(si)$  in
579    $\text{bnk}_{bi}(\text{cis}, \text{sis})(\text{lg} \dagger [si \mapsto (\text{bal} + \text{am}, \text{doi}, \text{doe})], \text{cr} + \text{am})$ 
577   end end

```

580 The refund action

581 non-deterministically externally offers to either

582 non-deterministically externally accept a $\text{mk_Ref}(ci,am)$ request from a shop behaviour, si , or
 583 non-deterministically externally accept a $\text{mk_Wdr}(ci,am)$ request from a shop behaviour, si .
 The bank behaviour is then resumed with the
 584 credit card's bank balance and cash register incremented by am and the
 585 shop's bank balance and cash register decremented by that same amount.

value

```
580 rfu: BI × (cis, sis): BM → (LG × CR) → in, out {ch_sb[bi, si] | si: SI • si ∈ sis} Unit
580 rfu(bi, (cis, sis))(lg, cr) ≡
582   (let (si, mk_Ref(cbi, (ci, am))) = [] {(si', ch_sb[si, bi]?) | si, si': SI • {si, si'} ⊆ sis ∧ si = si'} in
580     let (balc, doic, doec) = lg(ci) in
584     bnkbi(cis, sis)(lg†[ci → (balc + am, doic, doec)], cr + am)
580     end end)
581   []
583   (let (si, mk_Wdr(ci, am)) = [] {(si', ch_sb[si, bi]?) | si, si': SI • {si, si'} ⊆ sis ∧ si = si'} in
580     let (bals, dois, does) = lg(sis) in
585     bnkbi(cis, sis)(lg†[si → (bals - am, dois, does)], cr - am)
580     end end)
```

A.3.6 Shops

586 The shop behaviour, shp , takes the shop's unique identifier, the shop mereology, etcetera.
 587 The shop behaviour non-deterministically, externally
 either
 588 offers to accept a Buy request from a credit card behaviour,
 589 and instructs the shop's bank to deposit the purchase amount.
 590 whereupon the shop behaviour resumes being a shop behaviour;
 591 or
 592 offers to accept a refund request in this amount, am , from a credit card [holder].
 593 It then proceeds to inform the shop's bank to withdraw the refund from its ledger and cash register,
 594 and the credit card's bank to deposit the refund into its ledger and cash register.
 595 Whereupon the shop behaviour resumes being a shop behaviour.

value

```
586 shpsi: SI: (CI-set × BI) × ... → in, out: {ch_cs[ci, si] | ci: CI • ci ∈ cis}, {ch_sb[si, bi'] | bi': BI • bi' isin bis} Unit
586 shpsi((cis, bi), ...) ≡
588   (sal(si, (bi, cis), ...)
591     []
592     ref(si, (cis, bi), ...))

586 sal: SI × (CI-set × BI) × ... → in, out: {cs[ci, si] | ci: CI • ci ∈ cis}, sb[si, bi] Unit
586 sal(si, (cis, bi), ...) ≡
588   let mk_Buy(am) = [] {ch_cs[ci, si] | ci: CI • ci ∈ cis} in
589   ch_sb[si, bi]!mk_Dep(si, am) end ;
590   shpsi((cis, bi), ...)

586 ref: SI × (CI-set × BI) × ... → in, out: {ch_cs[ci, si] | ci: CI • ci ∈ cis}, {ch_sb[si, bi'] | bi': BI • bi' isin bis} Unit
592 ref(si, (cis, sbi), ...) ≡
592   let mk_Ref((ci, cbi, si), am) = [] {ch_cs[ci, si] | ci: CI • ci ∈ cis} in
593   (ch_sb[si, cbi]!mk_Ref(cbi, (ci, si), am)
594   || ch_sb[si, sbi]!mk_Wdr(si, am)) end ;
595   shpsi((cis, sbi), ...)
```

A.4 Discussion

TO BE WRITTEN

B

Weather Information Systems

Summary

This document reports *work in progress*. We show an example domain description. It is developed and presented as outlined in [70]. The domain being described is that of a generic weather information system. Four main endurants (i.e., aspects) of a generic weather information system are those of the weather, weather stations (collecting weather data), weather data interpretation (i.e., meteorological institute[s]), and weather forecast consumers. There are, correspondingly, two kinds of weather information: the weather data, and the weather forecasts. These forms of weather information are acted upon: the weather data interpreter (i.e., a meteorological institute) is gathering weather data; based on such interpretations the meteorological institute is “calculating” weather forecasts; and weather forecast consumers are requesting and further “interpreting” (i.e., rendering) such forecasts. Thus weather data is communicated from weather stations to the weather data interpreter; and weather forecasts are communicated from the weather data interpreter to the weather forecast consumers. It is the dual purpose of this technical report to present a domain description of the essence of generic weather information systems, and to add to the “pile” [48, 47, 56, 54, 57, 65, 63, 67] of technical reports that illustrate the use[fulness] of the principles, techniques and tools of [70].

B.1 On Weather Information Systems

B.1.1 On a Base Terminology

From Wikipedia:

- 596 **Weather** is the state of the atmosphere, to the degree that it is hot or cold, wet or dry, calm or stormy, clear or cloudy, atmospheric (barometric) pressure: high or low.
- 597 So weather is characterized by **temperature**, **humidity** (incl. **rain**, **wind** (direction, velocity, center, incl. its possible mobility), **atmospheric pressure**, etcetera.
- 598 By **weather information** we mean
- either weather data that characterizes the weather as defined above (Item 596),
 - or weather forecast, i.e., a prediction of the state of the atmosphere for a given location and time or time interval.
- 599 Weather data are collected by **weather stations**. We shall here not be concerned with technical means of weather data collection.
- 600 **Weather forecasts** are used by forecast consumers, anyone: you and me.

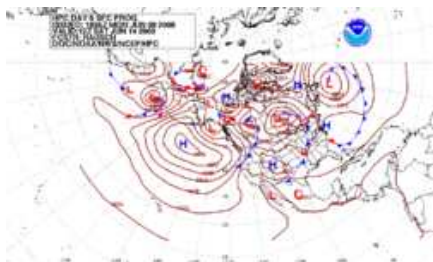
- 601 Weather data interpretation (i.e., **forecasting**) is the science and technology of creating weather forecasts based on **time-** or **time interval-stamped weather data** and **locations**. Weather data interpretation is amongst the charges of meteorological institutes.
- 602 **Meteorology** is the interdisciplinary scientific study of the atmosphere.
- 603 An **atmosphere** (from Greek *ατμοζ* (atmos), meaning “vapour”, and *σφαιρα* (sphaira), meaning “sphere”) is a layer of gases surrounding a planet or other material body, that is held in place by the gravity of that body.
- 604 Meteorological institutes work together with the World Meteorological Organization (WMO). Besides weather forecasting, meteorological institutes (and hence WMO) are concerned also with aviation, agricultural, nuclear, maritime, military and environmental meteorology, hydrometeorology and renewable energy.
- 605 Agricultural meteorologists, soil scientists, agricultural hydrologists, and agronomists are persons concerned with studying the effects of weather and climate on plant distribution, crop yield, water-use efficiency, phenology of plant and animal development, and the energy balance of managed and natural ecosystems. Conversely, they are interested in the rôle of vegetation on climate and weather.

B.1.2 Some Illustrations

Weather Stations



Weather Forecasts



Forecast Consumers



B.2 Major Parts of a Weather Information System

We think of the following parts as being of concern in the kind of weather information systems that we shall analyse and describe: Figure B.1 shows one **weather** (dashed rounded corner all embracing rectangle), one central **weather data interpreter** (cum meteorological institute) seven **weather stations** (rounded corner squares), nineteen **weather forecast consumers**, and one global **clock**. All are distributed, as hinted at, in some geographical space. Figure B.2 on the next page shows “an orderly diagram” of “the same”

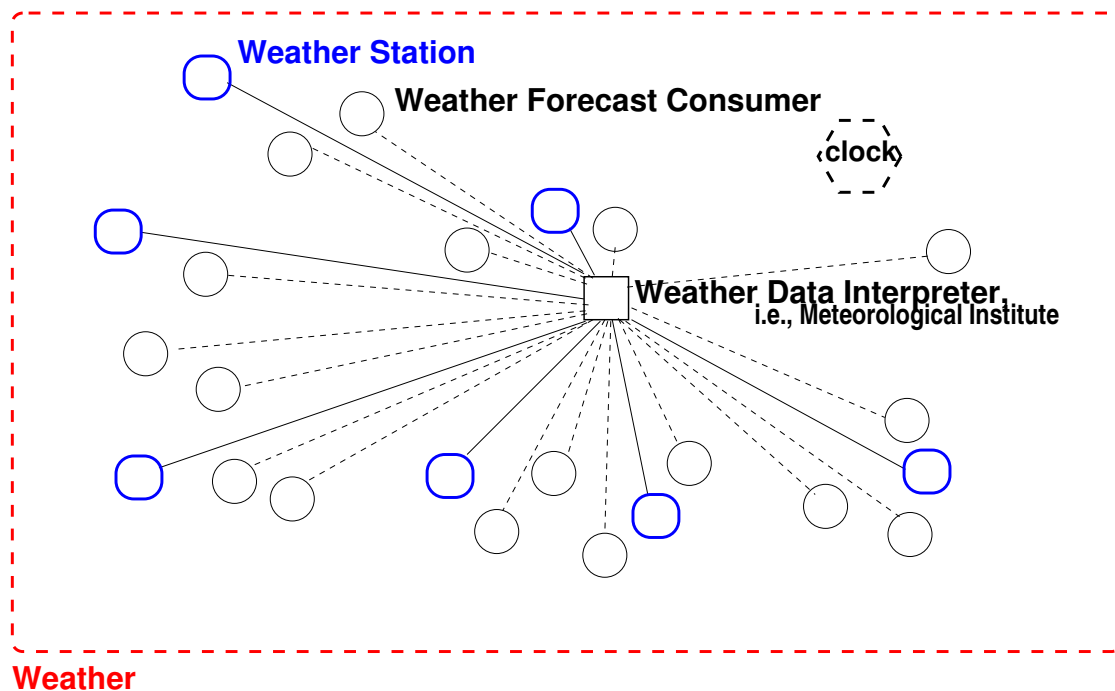


Fig. B.1. A Weather Information System

weather information system as Figure B.1. The lines between pairs of the various parts shall indicate means communication between the pairs of (thus) connected parts. Dashed lines “crossing” bundles of these communication lines are labeled ch_{xy} . These labels, ch_{xy} , designated CSP-like channels. An input, by a weather station (wsi), of weather data from the weather (wi), is designated by the CSP expression $ch_{ws}[wi, wsi]?$. An output, say from the weather data interpreter (wdi) to a weather forecast consumer (fci), of a forecast f , is designated by $ch_{ic}[wdii, fci]! f$

B.3 Endurants

B.3.1 Parts and Materials

- 606 The WIS domain contains a number of sub-domains:
- the weather, W , which we consider a material,
 - the weather stations sub-domain, WSS (a composite part),
 - the weather data interpretation sub-domain, $WDIS$ (an atomic part),
 - the weather forecast consumers sub-domain, $WFCS$ (a composite part), and

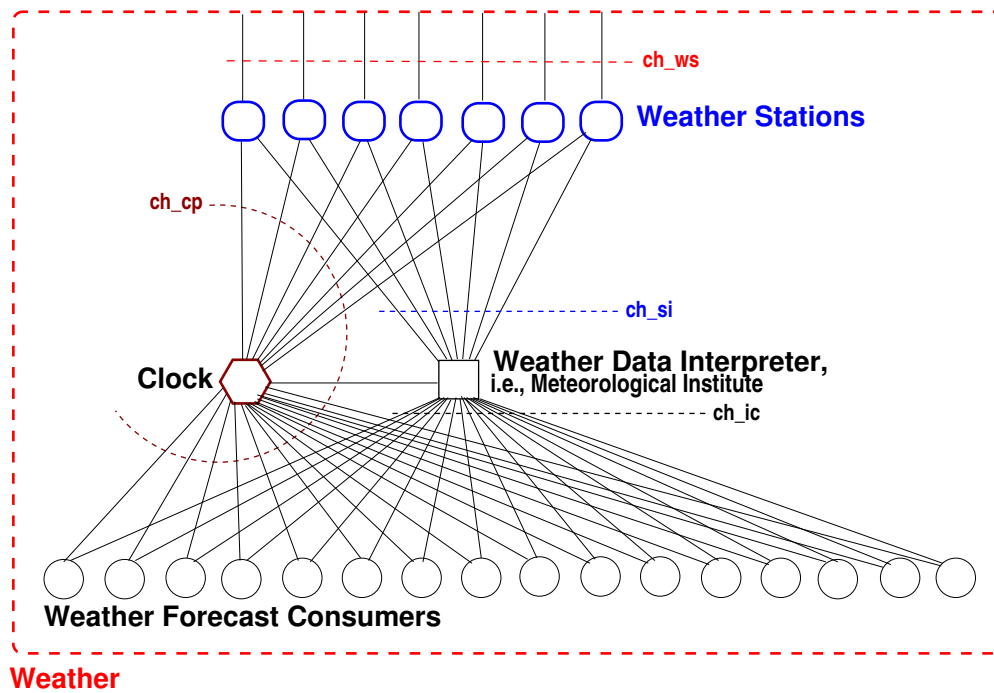


Fig. B.2. A Weather Information System Diagram

e the (“global”) clock (an atomic part).

type

- 606 WIS
- 606a W
- 606b WSS
- 606c WDIS
- 606d WFCS
- 606e CLK

value

- 606a obs_material_W: WIS → W
- 606b obs_part_WSS: WIS → WSS
- 606c obs_part_WDIS: WIS → WDIS
- 606d obs_part_WFCS: WIS → WFCS
- 606e obs_part_CLK: WIS → CLK

- 607 The weather station sub-domain, WSS, consists of a set, WSs,
- 608 of atomic weather stations, WS.
- 609 The weather forecast consumers sub-domain, WFCS, consists of a set, WFCs,
- 610 of atomic weather forecast consumers, WFC.

type

- 607 WSs = WS-set
- 608 WS
- 609 WFCs = WFC-set
- 610 WFC

value

607 obs_part_WSs: WSS \rightarrow WSs
 609 obs_part_WFCs: WFCS \rightarrow WFCs

B.3.2 Unique Identifiers

We shall consider only atomic parts.

- 611 Every single weather station has a unique identifier.
 612 The weather data interpretation (i.e., the weather forecast “creator”) has a unique identifier.
 613 Every single weather forecast consumer has a unique identifier.
 614 The global clock has a unique identifier.

type

611 WSI
 612 WDII
 613 WFCI
 614 CLKI

value

611 uid_WSI: WS \rightarrow WSI
 612 uid_WDII: WDIS \rightarrow WDII
 613 uid_WFCI: WFC \rightarrow WFCI
 613 uid_CLKI: CLK \rightarrow CLKI

B.3.3 Mereologies

We shall restrict ourselves to consider the mereologies only of the atomic parts.

- 615 The mereology of weather stations is the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
 616 The mereology of weather data interpreter is the triple of the unique clock identifier, set of unique identifiers of all the weather stations and the set of unique identifiers of all the weather forecast consumers.
 617 The mereology of weather forecast consumer is the the pair of the unique clock identifier and the unique identifier of the weather data interpreter.
 618 The mereology of the global clock is the triple of the set of all the unique identifiers of weather stations, the unique identifier of the weather data interpreter, and the set of all the unique identifiers of weather forecast consumers.

type

615 WSM = CLKI \times WDII
 616 WDIM = CLKI \times WSI-set \times WFCI-set
 617 WFCM = CLKI \times WDII
 618 CLKM = CLKI \times WDGI-set \times WDII \times WFCI-set

value

615 mereo_WSM: WS \rightarrow WSM
 616 mereo_WDI: WDI \rightarrow WDIM
 617 mereo_WFC: WFC \rightarrow WFCM
 618 mereo_CLK: CLK \rightarrow CLKM

B.3.4 Attributes**Clock, Time and Time-intervals**

- 619 The global clock has an autonomous time attribute.
 620 Time values are further undefined, but times are considered absolute in the sense as representing some intervals since “the birth of time”, an example, concrete time could be NOVEMBER 24, 2019: 13:16.
 621 Time intervals are further undefined, but time intervals can be considered relative in the sense of representing a quantity elapsed between two times, examples are: 1 day 2 hours and 3 minutes, etc. When a time interval, ti , is specified it is always to be understood to designate the times from now, or from a specified time, t , until the time $t + ti$.
 622 We postulate \oplus , \ominus , and can postulate further “arithmetic” operators, and
 623 we can postulate relational operators.

type

619 TIME

620 TI

value619 attr_TIME: CLK \rightarrow TIME622 \oplus : TIME \times TI \rightarrow TIME, TI \times TI \rightarrow TI622 \ominus : TIME \times TI \rightarrow TIME, TIME \times TIME \rightarrow TI623 =, \neq , <, \leq , \geq , >: TIME \times TIME \rightarrow **Bool**, TI \times TI \rightarrow **Bool**, ...

We do not here define these operations and relations.

Locations

- 624 Locations are metric, topological spaces and can thus be considered dense spaces of three dimensional points.
 625 We can speak of one location properly contained (\subset) within, or contained or equal (\subseteq), or equal ($=$), or not equal (\neq) to another location.

type

624. LOC

value625. \subset , \subseteq , =, \neq : LOC \times LOC \rightarrow **Bool****Weather**

- 626 The weather material is considered a dense, infinite set of weather point volumes WP. Some dense, infinite subsets (still proper volumes) of such points may be liquid, i.e., rain, water in rivers, lakes and oceans. Other dense, infinite subsets (still proper volumes) of such points may be gaseous, i.e., the air, or atmosphere. These two forms of proper volumes “border” along infinite subsets (curved planes, surfaces) of weather points.
 627 From the material weather one can observe its location.

type626 W = WP-**infset**

626 WP

value627 attr_LOC: W \rightarrow LOC

- 628 Some meteorological quantities are:

- a *Humidity*,
- b *Temperature*,
- c *Wind* and
- d *Barometric pressure*.

- 629 The weather has an indefinite number of attributes at any one time.
- a Humidity distribution, at level (above sea) and by location,
 - b Temperature distribution, at level (above sea) and by location,
 - c Wind direction, velocity and mobility of wind center, and by location,
 - d Barometric pressure, and by location,
 - e etc., etc.

type

- 628a Hu
- 628b Te
- 628c Wi
- 628d Ba
- 629a HDL = LOC \rightarrow Hu
- 629b TDL = LOC \rightarrow Te
- 629c WDL = LOC \rightarrow Wi
- 629d BPL = LOC \rightarrow Ba
- 629e ...

value

- 629a attr_HDL: W \rightarrow HDL
- 629b attr_TDL: W \rightarrow TDL
- 629c attr_WDL: W \rightarrow WDL
- 629d attr_APL: W \rightarrow BPL
- 629e ...

Weather Stations

- 630 Weather stations have static location attributes.
- 631 Weather stations sample the weather gathering humidity, temperature, wind, barometric pressure, and possibly other data, into time and location stamped weather data.

value

- 630 attr_LOC: WS \rightarrow LOC

type

- 631 WD :: mkWD((TIME×LOC)×(TDL×HDL×WDL×BPL×...))

Weather Data Interpreter

- 632 There is a programmable attribute: weather data repository, wdr:WDR, of weather data, wd:WD, collected from weather stations.
- 633 And there is programmable attribute: weather forecast repository, wfr:WFR, of forecasts, wf:WF, disseminate-able to weather forecast consumers.
- These repositories are updated when
- 634 received from the weather stations, respectively when
- 635 calculated by the weather data interpreter.

type

632 WDR

633 WFR

value634 $\text{update_wdr}: \text{TIME} \times \text{WD} \rightarrow \text{WDR} \rightarrow \text{WDR}$ 635 $\text{update_wfr}: \text{TIME} \times \text{WF} \rightarrow \text{WFR} \rightarrow \text{WFR}$

It is a standard exercise to define these two functions (say algebraically).

Weather Forecasts

636 Weather forecasts are weather forecast format-, time- and location-stamped quantities, the latter referred to as wefo:WeFo .

637 There are a definite number ($n \geq 1$) of weather forecast formats.

638 We do not presently define these various weather forecast formats.

639 They are here thought of as being requested, mkWFReq , by weather forecast consumers.

type636 $\text{WF} = \text{WFF} \times (\text{TIME} \times \text{TI}) \times \text{LOC} \times \text{WeFo}$ 637 $\text{WFF} = \text{WFF1} \mid \text{WFF2} \mid \dots \mid \text{WFFn}$ 638 $\text{WFF1}, \text{WFF2}, \dots, \text{WFFn}$ 639 $\text{WFReq} :: \text{mkWFReq}(s_wff:\text{WFF}, s_ti:(\text{TIME} \times \text{TI}), s_loc:\text{LOC})$ **Weather Forecast Consumer**

640 There is a programmable attribute, $d:D$, D for display (!).

641 Displays can be rendered (RND): visualized, tabularised, made audible, translated (between languages and language dialects, ...), etc.

642 A rendered display can be “abstracted back” into its basic form.

643 Any abstracted rendered display is identical to its abstracted form.

type640 D 641 RND **value**640 $\text{attr}_D: \text{WFC} \rightarrow D$ 641 $\text{rndr}_D: \text{RND} \times D \rightarrow D$ 642 $\text{abs}_D: D \rightarrow D$ **axiom**643 $\forall d:D, r:\text{RND} \cdot \text{abs}_D(\text{rndr}(r,d)) = d$ **B.4 Perdurants****B.4.1 A WIS Context**

644 We postulate a given system, wis:WIS .

That system is characterized by

645 a dynamic weather

646 and its unique identifier,

647 a set of weather stations

648 and their unique identifiers,

649 a single weather data interpreter

650 and its unique identifier,

651 a set of weather forecast consumers

652 and their unique identifiers, and

653 a single clock

654 and its unique identifier.

655 Given any specific wis:WIS there is [therefore] a full set of part identifiers, is, of weather, clock, all weather stations, the weather data interpreter and all weather forecast consumers.

We list the above-mentioned values. They will be referenced by the channel declarations and the behaviour definitions of this section.

value

```

644 wis:WIS
645 w:W = obs_material_W(wis)
646 wi:WI = uid_WI(w)
647 wss:WSs = obs_part_WSs(obs_part_WSS(wis))
648 wsis:WDGI-set = {uid_WSI(ws)|ws:WS•ws ∈ wss}
649 wdi:WDI = obs_part_WDIS(wis)
650 wdii:WDII = uid_WDII(wdi)
651 wfcs:WFCs = obs_part_WFCs(obs_part_WFCS(wis))
652 wfci:WFI-set = {uid_WFCI(wfc)|wfc:WFC•wfc ∈ wfcs}
653 clk:CLK = obs_part_CLK(wis)
654 clki:CLKI = uid_CLKI(clk)
655 is:(WI|WSI|WDII|WFCI)-set = {wi} ∪ wsis ∪ {wdii} ∪ wfci

```

B.4.2 Channels

- 656 Weather stations share weather data, WD, with the weather data interpreter — so there is a set of channels, one each, “connecting” weather stations to the weather data interpreter.
- 657 The weather data interpreter shares weather forecast requests, WReq, and interpreted weather data (i.e., forecasts), WF, with each and every forecast consumer — so there is a set of channels, one each, “connecting” the weather data interpreter to the interpreted weather data (i.e., forecast) consumers.
- 658 The clock offers its current time value to each and every part, except the weather, of the WIS system.

channel

```

656 { ch_si[ws_i,wdii]:WD | ws_i:WSI•ws_i ∈ wsis }
657 { ch_ic[wdii,fc_i]:(WReq|WF) | fc_i:FCI•fc_i ∈ fcis }
658 { ch_cp[clki,i]:TIME | i:(WI|CLKI|WSI|WDII|WFCI)•i ∈ is }

```

B.4.3 WIS Behaviours

- 659 WIS behaviour, wis_beh, is the
- 660 parallel composition of all the weather station behaviours, in parallel with the
- 661 weather data interpreter behaviour, in parallel with the
- 662 parallel composition of all the weather forecast consumer behaviours, in parallel with the
- 663 clock behaviour.

value

```

659 wis_beh: Unit → Unit
659 wis_beh() ≡
660   || { ws_beh(uid_WSI(ws),mereo_WS(ws),...) | ws:WS•ws ∈ wss } ||
661   || wdi_beh(uid_WDI(wdi),mereo_WDI(wdi),...)(wd_rep,wf_rep) ||
662   || { wfc_beh(uid_WFCI(wfc),mereo_WDG(wfc),...) | wfc:WFC•wfc ∈ wfcs } ||
663   clk_beh(uid_CLKI(clk),mereo_CLK(clk),...)("November 24, 2019: 13:16")

```

B.4.4 Clock

664 The clock behaviour has a programmable attribute, t .
 665 It repeatedly offers its current time to any part of the WIS system.
 It nondeterministically internally “cycles” between
 666 retaining its current time, or
 667 increment that time with a “small” time interval, δ , or
 668 offering the current time to a requesting part.

value

```
664. clk_beh: clki:CLKI × clkm:CLKM → TIME →
665.   out {ch_cp[clki,i]|i:(WSI|WDII|WFCI)•i ∈ wsis ∪ {wdii} ∪ wfcis } Unit
664. clk_beh(clki,is)(t) ≡
666.   clk_beh(clki,is)(t)
667.   □ clk_beh(clki,is)(t ⊕ δ)
668.   □ ( □ { ch_cp[clki,i] ! t | i:(WSI|WDII|WFCI)•i ∈ is } ; clk_beh(clki,is)(t) )
```

B.4.5 Weather Station

669 The weather station behaviour communicates with the global clock and the weather data interpreter.
 670 The weather station behaviour simply “cycles” between sampling the weather, reporting its findings to
 the weather data interpreter and resume being that overall behaviour.
 671 The weather station time-stamp “sample” the weather (i.e., meteorological information).
 672 The meteorological information obtained is analysed with respect to temperature (distribution etc.),
 673 humidity (distribution etc.),
 674 wind (distribution etc.),
 675 barometric pressure (distribution etc.), etcetera,
 676 and this is time-stamp and location aggregated (mkWD) and “sent” to the (central ?) weather data
 interpreter,
 677 whereupon the weather data generator behaviour resumes.

value

```
669 ws_beh: wsi:WSI × (clki,wi,wdii):WDGM × (LOC × ...) →
669   in ch_cp[clki,wsi] out ch_gi[wsi,wdii] Unit
670 ws_beh(wsi,(clki,wi,wdii),(loc,...)) ≡
672   let tdl = attr_TDL(w),
673       hdl = attr_HDL(w),
674       wdl = attr_WDL(w),
675       bpl = attr_BPL(w), ... in
676   ch_gi[wsi,wdii] ! mkWD((ch_cp[clki,wsi] ?,loc),(tdl,hdl,wdl,bpl,...)) end ;
677   wdg_beh(wsi,(clki,wi,wdii),(loc,...))
```

B.4.6 Weather Data Interpreter

678 The weather data interpreter behaviour communicates with the global clock, all the weather stations
 and all the weather forecast consumers.
 679 The weather data interpreter behaviour non-deterministically internally (□) chooses to
 680 either collect weather data,
 681 or calculate some weather forecast,
 682 or disseminate a weather forecast.

value

```

678 wdi_beh: wdii:WDII × (clki, wsis, wfcis):WDIM × ... → (WD_Rep × WF_Rep) →
678     in ch_cp[clki, wdii], { ch_si[ws_i, wdii] | wsi:WSI • wsi ∈ wsis },
678     out { ch_ic[wdii, wfc_i] | wfc_i:WFCI • wfc_i ∈ wfcis } Unit
678 wdi_beh(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep) ≡
680   collect_wd(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep)
679   []
681   calculate_wf(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep)
679   []
682   disseminate_wf(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep)

```

collect_wd

683 The collect weather data behaviour communicates with the global clock and all the weather stations – but “passes-on” the capability to communicate with all of the weather forecast consumers.
684 The collect weather data behaviour
685 non-deterministically externally offers to accept weather data from some weather station,
686 updates the weather data repository with a time-stamped version of that weather data,
687 and resumes being a weather data interpreter behaviour, now with an updated weather data repository.

value

```

683 collect_wd: wdii:WDII × (clki, wsis, wfcis):WDIM × ...
683   → (WD_Rep × WF_Rep) →
683   in ch_cp[clki, wdii], { ch_si[ws_i, wdii] | wsi:WSI • wsi ∈ wsis },
683   out { ch_ic[wdii, wfc_i] | wfc_i:WFCI • wfc_i ∈ wfcis } Unit
684 collect_wd(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep) ≡
685   let ((ti, loc), (hdl, tdl, wdl, bpl, ...)) = [] { wsi[ws_i, wdii]? | wsi:WSI • wsi ∈ wsis } in
686   let wd_rep' = update_wdr(ch_cp[clki, wdii]? , ((ti, loc), (hdl, tdl, wdl, bpl, ...)))(wd_rep) in
687   wdi_beh(wdii, (clki, wsis, wfcis), ...)(wd_rep', wf_rep) end end

```

calculate_wf

688 The calculate forecast behaviour communicates with the global clock – but “passes-on” the capability to communicate with all of weather stations and the weather forecast consumers.
689 The calculate forecast behaviour
690 non-deterministically internally chooses a forecast type from among a indefinite set of such,
691 and a current or “future” time-interval,
692 whereupon it calculates the weather forecast and updates the weather forecast repository,
693 and then resumes being a weather data interpreter behaviour now with the weather forecast repository updated with the calculated forecast.

value

```

688 calculate_wf: wdii:WDII × (clki, wsis, wfcis):WDIM × ... → (WD_Rep × WF_Rep) →
688   in ch_cp[clki, wdii], { ch_si[ws_i, wdii] | wsi:WSI • wsi ∈ wsis },
688   out { ch_ic[wdii, wfc_i] | wfc_i:WFCI • wfc_i ∈ wfcis } Unit
689 calculate_wf(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep) ≡
690   let tf:WWF = ft1 [] ft2 [] ... [] ft_n,
691   ti:(TIME × TIVAL) • toti ≥ ch_cp[clki, wdii] ? in
692   let wf_rep' = update_wfr(calc_wf(tf, ti)(wf_rep)) in
693   wdi_beh(wdii, (clki, wsis, wfcis), ...)(wd_rep, wf_rep') end end

```

694 The calculate_weather forecast function is, at present, further undefined.

value

694. $\text{calc_wf}: \text{WFF} \times (\text{TIME} \times \text{TI}) \rightarrow \text{WFRep} \rightarrow \text{WF}$

694. $\text{calc_wf}(\text{tf}, \text{ti})(\text{wf_rep}) \equiv \dots$

disseminate_wf

695 The disseminate weather forecast behaviour communicates with the global clock and all the weather forecast consumers – but “passes-on” the capability to communicate with all of weather stations.

696 The disseminate weather forecast behaviour non-deterministically externally offers to received a weather forecast request from any of the weather forecast consumers, *wfci*, that request is for a specific format forecast, *tf*, and either for a specific time or for a time-interval, *toti*, as well as for a specific location, *loc*.

697 The disseminate weather forecast behaviour retrieves an appropriate forecast and

698 sends it to the requesting consumer –

699 whereupon the disseminate weather forecast behaviour resumes being a weather data interpreter behaviour

value

695 $\text{disseminate_wf}: \text{wdii}:\text{WDII} \times (\text{clki}, \text{wsis}, \text{wfcis}):\text{WDIM} \times \dots \rightarrow (\text{WD_Rep} \times \text{WF_Rep}) \rightarrow$

695 $\quad \text{in } \text{ch_cp}[\text{clki}, \text{wdii}] \text{ in, out } \{ \text{ch_ic}[\text{wdii}, \text{wfci}] \mid \text{wfci}:\text{WFCI} \cdot \text{wfci} \in \text{wfcis} \} \text{ Unit}$

695 $\text{disseminate_wf}(\text{wdii}, (\text{clki}, \text{wsis}, \text{wfcis}), \dots)(\text{wd_rep}, \text{wf_rep}) \equiv$

696 $\quad \text{let } \text{mkReqWF}((\text{tf}, \text{toti}, \text{loc}), \text{wfci}) = \square \{ \text{ch_ic}[\text{wdii}, \text{wfci}] ? \mid \text{wfci}:\text{WFCI} \cdot \text{wfci} \in \text{wfcis} \} \text{ in}$

697 $\quad \text{let } \text{wf} = \text{retr_WF}((\text{tf}, \text{toti}, \text{loc}), \text{wf_rep}) \text{ in}$

698 $\quad \text{ch_ic}[\text{wdii}, \text{wfci}] ! \text{wf} ;$

699 $\quad \text{disseminate_wf}(\text{wdii}, (\text{clki}, \text{wsis}, \text{wfcis}), \dots)(\text{wd_rep}, \text{wf_rep}) \text{ end end}$

700 The $\text{retr_WF}((\text{tf}, \text{toti}, \text{loc}), \text{wf_rep})$ function invocation retrieves the weather forecast from the weather forecast repository most “closely” matching the format, *tf*, time, *toti*, and location of the request received from the weather forecast consumer. We do not define this function.

700. $\text{retr_WF}: (\text{WFF} \times (\text{TIME} \times \text{TI}) \times \text{LOC}) \times \text{WFRep} \rightarrow \text{WF}$

700. $\text{retr_WF}(\text{tf}, \text{toti}, \text{loc}), \text{wf_rep}) \equiv \dots$

We could have included, in our model, the time-stamping of receipt (formula Item 696) of requests, and the time-stamping of delivery of requested forecast in which case we would insert $\text{ch_cp}[\text{clki}, \text{wdii}]?$ at respective points in formula Items 696 and 698.

B.4.7 Weather Forecast Consumer

701 The weather forecast consumer communicates with the global clock and the weather data interpreter.

702 The weather forecast consumer behaviour

703 nondeterministically internally either

704 selects a suitable weather cast format, *tf*,

705 selects a suitable location, *loc'*, and

706 selects, *toti*, a suitable time (past, present or future) or a time interval (that is supposed to start when forecast request is received by the weather data interpreter).

707 With a suitable formatting of this triple, $\text{mkReqWF}(\text{tf}, \text{loc}', \text{toti})$, the weather forecast consumer behaviour “outputs” a request for a forecast to the weather data interpreter (first “half” of formula Item 706) whereupon it awaits (:) its response (last “half” of formula Item 706) which is a weather forecast, *wf*,

708 whereupon the weather forecast consumer behaviour resumes being that behaviour with its programmable attribute, d , being replaced by the received forecast suitably annotated;
 709 or the weather forecast consumer behaviour
 710 edits a display
 711 and resumes being a weather forecast consumer behaviour with the edited programmable attribute, d' .

value

```

701 wfc_beh: wfc:WFCI × (clki,wdii):WFCM × (LOC × ...) → D →
702   in ch_cp[clki,wfc],
703   in,out { ch_ic[wdii,wfc] | wfc:WFCI•wfc ∈ wfcis } Unit
704 wfc_beh(wfc,(clki,wdii),(loc,...))(d) ≡
705   let tf = tf1 ∩ tf2 ∩ ... ∩ tfn,
706       loc':LOC • loc'=loc ∨ loc'≠loc,
707       (t,ti):(TIME×TI) • ti≥0 in
708   let wf = (ch_ic[wdii,wfc] ! mkReqWF(tf,loc',(t,ti))) ; ch_ic[wdii,wfc] ? in
709   wfc_beh(wfc,(clki,wdii),(loc,...))((tf,loc',(t,ti)),wf) end end
710   ∩
711   let d':D { \EQ } rndr\_D(d,{ \DOTDOTDOT }) in
712   wfc_beh(wfc,(clki,wdii),(loc,...))(d') end
  
```

The choice of location may be that of the weather forecast consumer location, or it may be one different from that. The choice of time and time-interval is likewise a non-deterministic internal choice.

B.5 Conclusion

B.5.1 Reference to Similar Work

As far as I know there are no published literature nor, to our knowledge, institutional or private works on the subject of modelling weather data collection, interpretation and weather forecast delivery systems.

B.5.2 What Have We Achieved ?

TO BE WRITTEN

B.5.3 What Needs to be Done Next ?

TO BE WRITTEN

B.5.4 Acknowledgements

This technical cum experimental research report was begun in Bergen, Wednesday, November 9, 2016 – inspired by a presentation by Ms. Doreen Tuheirwe, Makerere University, Kampala, Uganda. I thank her, and Profs. Magne Haveraaen and Jaakko Järvi of BLDL: the Bergen Language Design Laboratory, Dept. of Informatics, University of Bergen (Norway), for their early comments, and Prof. Haveraaen for inviting me to give PhD lectures there in the week of Nov. 6–12, 2016.

C

Pipeline Systems

Summary



Fig. C.1. The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

- Named after Verdi's opera
- Gas pipeline
- 3300 kms
- 2011–2014, first gas flow: 2014; 2017–2019, more pipes
- 8 billion Euros
- Max flow: 31 bcm; billion cubic meters a year
- <http://www.nabucco-pipeline.com/>

C.1 Photos of Pipeline Units and Diagrams of Pipeline Systems

When combining joins and forks we can construct sitches. Figure C.7 on Page 307 shows some actual switches.

Figure C.8 on Page 308 diagrams a generic switch.

⁰ See http://en.wikipedia.org/wiki/Nabucco_Pipeline



Fig. C.2. The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

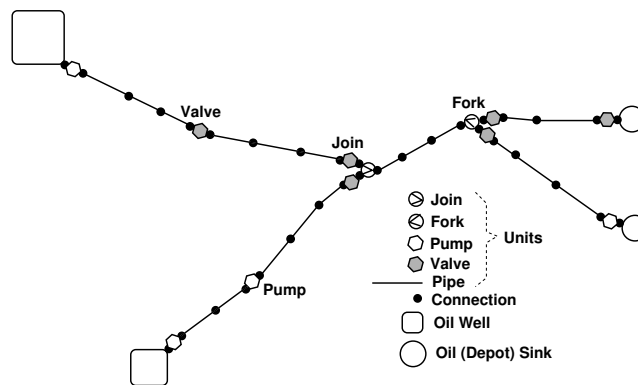


Fig. C.3. An oil pipeline system

C.2 Non-Temporal Aspects of Pipelines

These are some non-temporal aspects of pipelines. nets and units: wells, pumps, pipes, valves, joins, forks and sinks; net and unit attributes; and units states, but not state changes. We omit, in early (i.e., next) chapters, consideration of “pigs” and “pig”-insertion and “pig”-extraction units.

C.2.1 Nets of Pipes, Valves, Pumps, Forks and Joins

711 We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.

712 Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.

713 Units are explained in terms of disjoint types of Pipes, VALves, PUMps, FORks, JOins, WELls and SKs.¹

type

711 N, PI, VA, PU, FO, JO, WE, SK

712 $U = \Pi \mid V \mid P \mid F \mid J \mid S \mid W$

712 $\Pi == \text{mk}\Pi(\text{pi}:\text{PI})$

712 $V == \text{mk}V(\text{va}:\text{VA})$

712 $P == \text{mk}P(\text{pu}:\text{PU})$

712 $F == \text{mk}F(\text{fo}:\text{FO})$

¹ This is a mere specification language technicality.



Fig. C.4. Pipes



Fig. C.5. Valves

712 J == mkJ(jo:JO)
712 W == mkW(we:WE)
712 S == mkS(sk:SK)

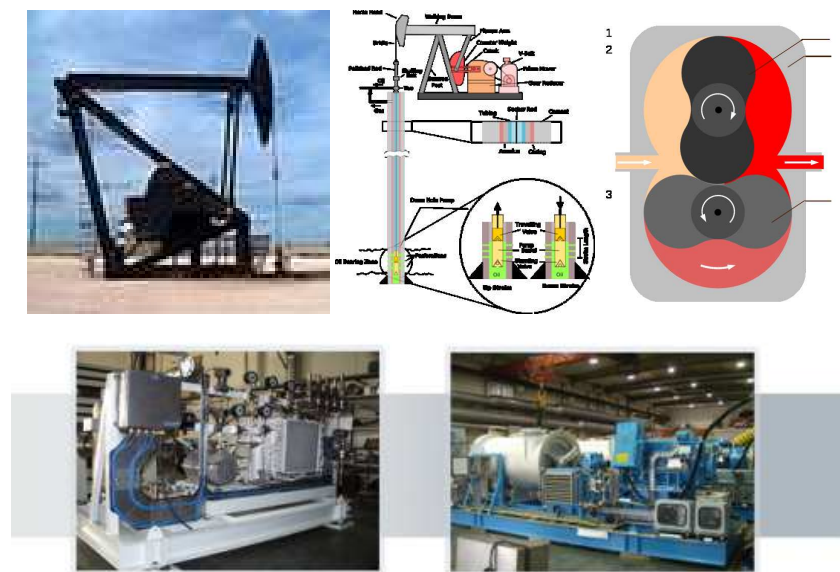


Fig. C.6. Oil Pumps and Gas Compressors

C.2.2 Unit Identifiers and Unit Type Predicates

714 We associate with each unit a unique identifier, $ui : UI$.

715 From a unit we can observe its unique identifier.

716 From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

type

714 UI

value

715 $obs_UI: U \rightarrow UI$

716 $is_PI: U \rightarrow \mathbf{Bool}$, $is_V: U \rightarrow \mathbf{Bool}$, ..., $is_J: U \rightarrow \mathbf{Bool}$

$is_PI(u) \equiv \mathbf{case\ } u \mathbf{ of\ } mkPI(_) \rightarrow \mathbf{true}, _ \rightarrow \mathbf{false\ } \mathbf{end}$

$is_V(u) \equiv \mathbf{case\ } u \mathbf{ of\ } mkV(_) \rightarrow \mathbf{true}, _ \rightarrow \mathbf{false\ } \mathbf{end}$

...

$is_S(u) \equiv \mathbf{case\ } u \mathbf{ of\ } mkS(_) \rightarrow \mathbf{true}, _ \rightarrow \mathbf{false\ } \mathbf{end}$

C.2.3 Unit Connections

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from “the other side”.

717 With a pipe, a valve and a pump we associate exactly one input and one output connection.

718 With a fork we associate a maximum number of output connections, m , larger than one.

719 With a join we associate a maximum number of input connections, m , larger than one.

720 With a well we associate zero input connections and exactly one output connection.

721 With a sink we associate exactly one input connection and zero output connections.



Fig. C.7. Oil and Gas Switches

value717 $\text{obs_InCs, obs_OutCs: } \Pi |V|P \rightarrow \{1:\mathbf{Nat}\}$ 718 $\text{obs_inCs: } F \rightarrow \{1:\mathbf{Nat}\}, \text{ obs_outCs: } F \rightarrow \mathbf{Nat}$ 719 $\text{obs_inCs: } J \rightarrow \mathbf{Nat}, \text{ obs_outCs: } J \rightarrow \{1:\mathbf{Nat}\}$ 720 $\text{obs_inCs: } W \rightarrow \{0:\mathbf{Nat}\}, \text{ obs_outCs: } W \rightarrow \{1:\mathbf{Nat}\}$ 721 $\text{obs_inCs: } S \rightarrow \{1:\mathbf{Nat}\}, \text{ obs_outCs: } S \rightarrow \{0:\mathbf{Nat}\}$ **axiom**718 $\forall f:F \cdot \text{obs_outCs}(f) \geq 2$ 719 $\forall j:J \cdot \text{obs_inCs}(j) \geq 2$

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then

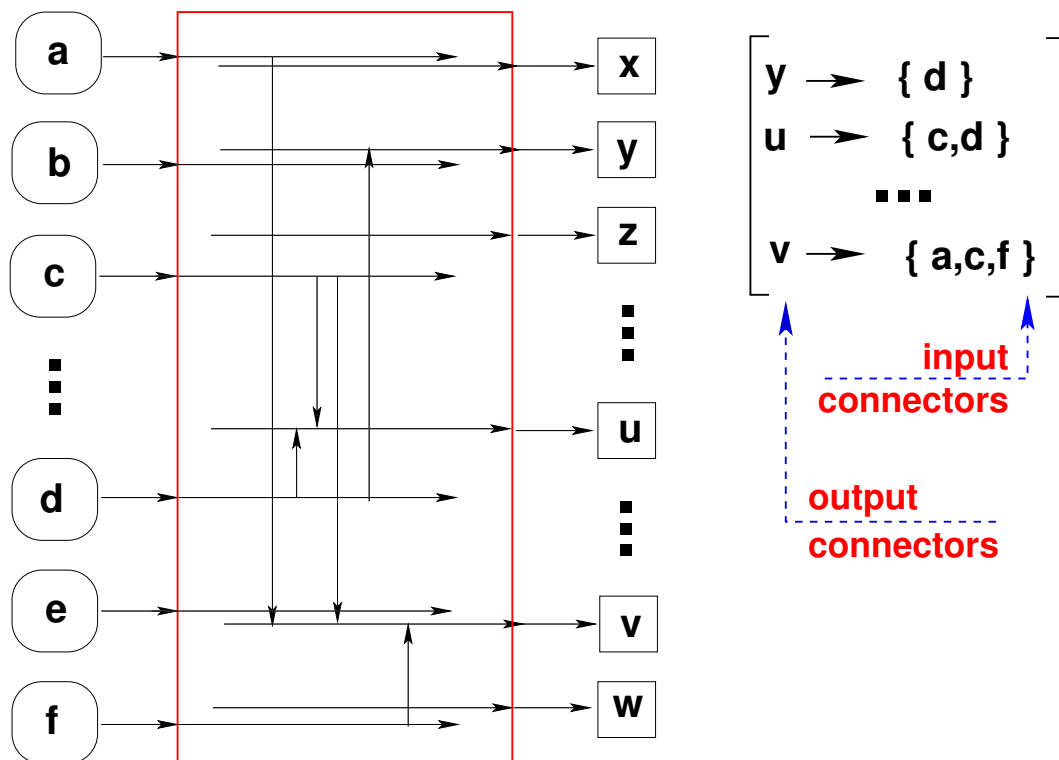


Fig. C.8. A Switch Diagram

it means that the fork input connector has been sealed. If a fork is output-connected to n units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: “the other way around”.

C.2.4 Net Observers and Unit Connections

722 From a net one can observe all its units.

723 From a unit one can observe the the pairs of disjoint input and output units to which it is connected:

- a Wells can be connected to zero or one output unit — a pump.
- b Sinks can be connected to zero or one input unit — a pump or a valve.
- c Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.
- d Forks, f , can be connected to zero or one input unit and to zero or n , $2 \leq n \leq \text{obs_Cs}(f)$ output units.
- e Joins, j , can be connected to zero or n , $2 \leq n \leq \text{obs_Cs}(j)$ input units and zero or one output units.

value

722 $\text{obs_Us}: N \rightarrow \text{U-set}$

723 $\text{obs_cUs}: U \rightarrow \text{UI-set} \times \text{UI-set}$

$\text{wf_Conns}: U \rightarrow \text{Bool}$

$\text{wf_Conns}(u) \equiv$

let $(\text{iuis}, \text{ouis}) = \text{obs_cUs}(u)$ **in** $\text{iuis} \cap \text{ouis} = \{\}$ **^**
case u **of**

723a $\text{mkW}(_) \rightarrow \text{card } \text{iuis} \in \{0\} \wedge \text{card } \text{ouis} \in \{0,1\}$,



Fig. C.9. To be treated in a later version of this report: Pig Launcher, Receiver and New and Old Pigs

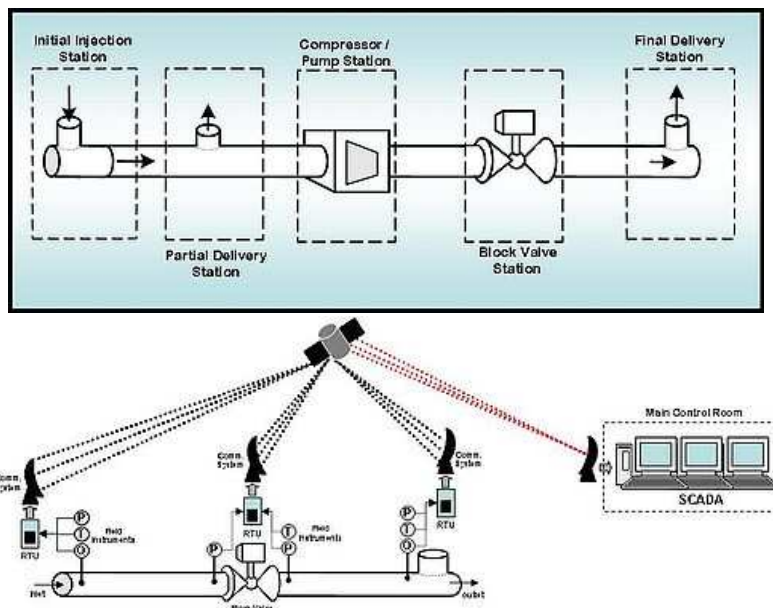


Fig. C.10. Pipeline Diagrams

- 723b $mkS(_) \rightarrow \mathbf{card\ iuis} \in \{0,1\} \wedge \mathbf{card\ ouis} \in \{0\}$,
- 723c $mkII(_) \rightarrow \mathbf{card\ iuis} \in \{0,1\} \wedge \mathbf{card\ ouis} \in \{0,1\}$,
- 723c $mkV(_) \rightarrow \mathbf{card\ iuis} \in \{0,1\} \wedge \mathbf{card\ ouis} \in \{0,1\}$,
- 723c $mkP(_) \rightarrow \mathbf{card\ iuis} \in \{0,1\} \wedge \mathbf{card\ ouis} \in \{0,1\}$,
- 723d $mkF(_) \rightarrow \mathbf{card\ iuis} \in \{0,1\} \wedge \mathbf{card\ ouis} \in \{0\} \cup \{2..obs_inCs(j)\}$,
- 723e $mkJ(_) \rightarrow \mathbf{card\ iuis} \in \{0\} \cup \{2..obs_inCs(j)\} \wedge \mathbf{card\ ouis} \in \{0,1\}$

end end

C.2.5 Well-formed Nets, Actual Connections

724 The unit identifiers observed by the `obs_cUls` observer must be identifiers of units of the net.

axiom

```

724  $\forall n:N, u:U \cdot u \in \text{obs\_Us}(n) \Rightarrow$ 
724 let (iuis,ouis) = obs_cUls(u) in
724  $\forall ui:UI \cdot ui \in \text{iuis} \cup \text{ouis} \Rightarrow$ 
724  $\exists u':U \cdot u' \in \text{obs\_Us}(n) \wedge u' \neq u \wedge \text{obs\_UI}(u') = ui$  end
```

C.2.6 Well-formed Nets, No Circular Nets

725 By a route we shall understand a sequence of units.

726 Units form routes of the net.

type

```
725  $R = UI^\omega$ 
```

value

```

726 routes:  $N \rightarrow R$ -infset
726 routes(n)  $\equiv$ 
726 let us = obs_Us(n) in
726 let rs =  $\{\langle u \rangle \mid u:U \cdot u \in \text{us}\} \cup \{\hat{r} \mid r,r':R \cdot \{r,r'\} \subseteq \text{rs} \wedge \text{adj}(r,r')\}$  in
726 rs end end
```

727 A route of length two or more can be decomposed into two routes

728 such that the least unit of the first route “connects” to the first unit of the second route.

value

```

727 adj:  $R \times R \rightarrow \text{Bool}$ 
727 adj(fr,lr)  $\equiv$ 
727 let (lu,fu) = (fr(len fr),hd lr) in
728 let (lui,fui) = (obs_UI(lu),obs_UI(fu)) in
728 let ((_,luis),(fuis,_)) = (obs_cUls(lu),obs_cUls(fu)) in
728 lui  $\in$  fuis  $\wedge$  fui  $\in$  luis end end end
```

729 No route must be circular, that is, the net must be acyclic.

value

```

729 acyclic:  $N \rightarrow \text{Bool}$ 
729 let rs = routes(n) in
729  $\sim \exists r:R \cdot r \in \text{rs} \Rightarrow \exists i,j:\text{Nat} \cdot \{i,j\} \subseteq \text{inds } r \wedge i \neq j \wedge r(i) = r(j)$  end
```

C.2.7 Well-formed Nets, Special Pairs, wfN_SP

- 730 We define a “special-pairs” well-formedness function.
- Fork outputs are output-connected to valves.
 - Join inputs are input-connected to valves.
 - Wells are output-connected to pumps.
 - Sinks are input-connected to either pumps or valves.

value

```

730 wfN_SP: N → Bool
730 wfN_SP(n) ≡
730   ∀ r:R • r ∈ routes(n) in
730     ∀ i:Nat • {i,i+1} ⊆ inds r ⇒
730       case r(i) of ∧
730a      mkF(⟦_⟧) → ∀ u:U•adj(⟦r(i)⟧,⟦u⟧) ⇒ is_V(u,⟦_⟧)→true end ∧
730       case r(i+1) of
730b      mkJ(⟦_⟧) → ∀ u:U•adj(⟦u⟧,⟦r(i)⟧) ⇒ is_V(u,⟦_⟧)→true end ∧
730       case r(1) of
730c      mkW(⟦_⟧) → is_P(r(2)),⟦_⟧→true end ∧
730       case r(len r) of
730d      mkS(⟦_⟧) → is_P(r(len r-1))∨is_V(r(len r-1)),⟦_⟧→true end

```

The **true** clauses may be negated by other **case** distinctions’ is_V or is_V clauses.

C.2.8 Special Routes, I

- 731 A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.
- 732 A simple pump-pump route is a pump-pump route with no forks and joins.
- 733 A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.
- 734 A simple pump-valve route is a pump-valve route with no forks and joins.
- 735 A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.
- 736 A simple valve-pump route is a valve-pump route with no forks and joins.
- 737 A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.
- 738 A simple valve-valve route is a valve-valve route with no forks and joins.

value

```

731-738 ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
       pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n ≥ 2

731 ppr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
732 sppr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ ppr(r) ∧ is_πr(ℓ)
733 pvr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
734 spvr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ ppr(r) ∧ is_πr(ℓ)
735 vpr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
736 spvr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ ppr(r) ∧ is_πr(ℓ)
737 vvr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
738 spvr(r:⟦fu⟧^ℓ^⟦lu⟧) ≡ ppr(r) ∧ is_πr(ℓ)

```

```

is_πfjr, is_πr: R → Bool
is_πfjr(r) ≡ ∀ u:U•u ∈ elems r ⇒ is_Π(u) ∨ is_F(u) ∨ is_J(u)
is_πr(r) ≡ ∀ u:U•u ∈ elems r ⇒ is_Π(u)

```

C.2.9 Special Routes, II

Given a unit of a route,
739 if they exist (\exists),
740 find the nearest pump or valve unit,
741 “upstream” and
742 “downstream” from the given unit.

value

```

739 ∃UpPoV: U × R → Bool
739 ∃DoPoV: U × R → Bool
741 find_UpPoV: U × R  $\xrightarrow{\sim}$  (P|V), pre find_UpPoV(u,r): ∃UpPoV(u,r)
742 find_DoPoV: U × R  $\xrightarrow{\sim}$  (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
739 ∃UpPoV(u,r) ≡
739 ∃ i,j Nat•{i,j} ⊆ inds r ∧ i ≤ j ∧ {is_V|is_P}(r(i)) ∧ u=r(j)
739 ∃DoPoV(u,r) ≡
739 ∃ i,j Nat•{i,j} ⊆ inds r ∧ i ≤ j ∧ u=r(i) ∧ {is_V|is_P}(r(j))
741 find_UpPoV(u,r) ≡
741 let i,j:Nat•{i,j} ⊆ inds r ∧ i ≤ j ∧ {is_V|is_P}(r(i)) ∧ u=r(j) in r(i) end
742 find_DoPoV(u,r) ≡
742 let i,j:Nat•{i,j} ⊆ inds r ∧ i ≤ j ∧ u=r(i) ∧ {is_V|is_P}(r(j)) in r(j) end

```

C.3 State Attributes of Pipeline Units

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units.

743 Oil flow, $\phi : \Phi$, is measured in volume per time unit.
744 Pumps are either pumping or not pumping, and if not pumping they are closed.
745 Valves are either open or closed.
746 Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
747 At any time any unit is sustaining a current input flow of oil (at its input(s)).
748 While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

type

```

743  $\Phi$ 
744 PΣ == pumping | not_pumping
744 VΣ == open | closed

```

value

```

-,+:  $\Phi \times \Phi \rightarrow \Phi$ , <,,>:  $\Phi \times \Phi \rightarrow \mathbf{Bool}$ 
744 obs_PΣ: P → PΣ
745 obs_VΣ: V → VΣ

```


746–748 $\text{obs_Lami}\Phi, \text{obs_Curr}\Phi, \text{obs_Leak}\Phi: U \rightarrow \Phi$
 $\text{is_Open}: U \rightarrow \mathbf{Bool}$
case u **of**
 $\text{mkII}(_) \rightarrow \mathbf{true}, \text{mkF}(_) \rightarrow \mathbf{true}, \text{mkJ}(_) \rightarrow \mathbf{true}, \text{mkW}(_) \rightarrow \mathbf{true}, \text{mkS}(_) \rightarrow \mathbf{true},$
 $\text{mkP}(_) \rightarrow \text{obs_P}\Sigma(u) = \text{pumping},$
 $\text{mkV}(_) \rightarrow \text{obs_V}\Sigma(u) = \text{open}$
end
 $\text{acceptable_Leak}\Phi, \text{excessive_Leak}\Phi: U \rightarrow \Phi$
axiom
 $\forall u:U \cdot \text{excess_Leak}\Phi(u) > \text{accept_Leak}\Phi(u)$

C.3.1 Flow Laws

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws.

749 When, in Item 747, for a unit u , we say that at any time any unit is sustaining a current input flow of oil, and when we model that by $\text{obs_Curr}\Phi(u)$ then we mean that $\text{obs_Curr}\Phi(u) - \text{obs_Leak}\Phi(u)$ represents the flow of oil from its outputs.

value

749 $\text{obs_in}\Phi: U \rightarrow \Phi$
749 $\text{obs_in}\Phi(u) \equiv \text{obs_Curr}\Phi(u)$
749 $\text{obs_out}\Phi: U \rightarrow \Phi$

law:

749 $\forall u:U \cdot \text{obs_out}\Phi(u) = \text{obs_Curr}\Phi(u) - \text{obs_Leak}\Phi(u)$

750 Two connected units enjoy the following flow relation:

a If

- | | | |
|----------------------------|----------------------------|-----------------------------|
| i two pipes, or | iv a valve and a valve, or | vii a pump and a pump, or |
| ii a pipe and a valve, or | v a pipe and a pump, or | viii a pump and a valve, or |
| iii a valve and a pipe, or | vi a pump and a pipe, or | ix a valve and a pump |

are immediately connected

b then

- i the current flow out of the first unit's connection to the second unit
- ii equals the current flow into the second unit's connection to the first unit

law:

750a $\forall u, u':U \cdot \{\text{is_II}, \text{is_V}, \text{is_P}, \text{is_W}\}(u'|u'') \wedge \text{adj}(\langle u \rangle, \langle u' \rangle)$
750a $\text{is_II}(u) \vee \text{is_V}(u) \vee \text{is_P}(u) \vee \text{is_W}(u) \wedge$
750a $\text{is_II}(u') \vee \text{is_V}(u') \vee \text{is_P}(u') \vee \text{is_S}(u')$
750b $\Rightarrow \text{obs_out}\Phi(u) = \text{obs_in}\Phi(u')$

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

C.3.2 Possibly Desirable Properties

- 751 Let r be a route of length two or more, whose first unit is a pump, p , whose last unit is a valve, v and whose intermediate units are all pipes: if the pump, p is pumping, then we expect the valve, v , to be open.
- 752 Let r be a route of length two or more, whose first unit is a pump, p , whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, p is pumping, then we expect pump p' , to also be pumping.
- 753 Let r be a route of length two or more, whose first unit is a valve, v , whose last unit is a pump, p and whose intermediate units are all pipes: if the valve, v is closed, then we expect pump p , to not be pumping.
- 754 Let r be a route of length two or more, whose first unit is a valve, v' , whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'' , to also be in the same state.

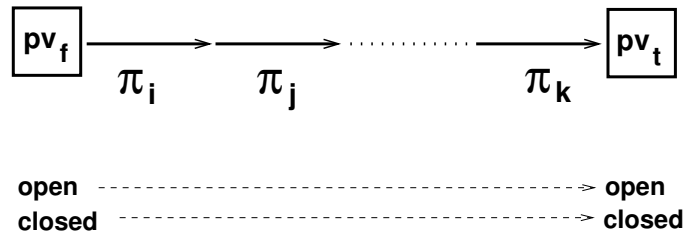


Fig. C.11. pv: Pump or valve, π : pipe

desirable properties:

$$751 \quad \forall r:R \cdot \text{spvr}(r) \wedge$$

$$751 \quad \text{spvr_prop}(r): \text{obs_P}\Sigma(\mathbf{hd} \ r)=\text{pumping} \Rightarrow \text{obs_P}\Sigma(r(\mathbf{len} \ r))=\text{open}$$

$$752 \quad \forall r:R \cdot \text{spvr}(r) \wedge$$

$$752 \quad \text{spvr_prop}(r): \text{obs_P}\Sigma(\mathbf{hd} \ r)=\text{pumping} \Rightarrow \text{obs_P}\Sigma(r(\mathbf{len} \ r))=\text{pumping}$$

$$753 \quad \forall r:R \cdot \text{svpr}(r) \wedge$$

$$753 \quad \text{svpr_prop}(r): \text{obs_P}\Sigma(\mathbf{hd} \ r)=\text{open} \Rightarrow \text{obs_P}\Sigma(r(\mathbf{len} \ r))=\text{pumping}$$

$$754 \quad \forall r:R \cdot \text{svvr}(r) \wedge$$

$$754 \quad \text{svvr_prop}(r): \text{obs_P}\Sigma(\mathbf{hd} \ r)=\text{obs_P}\Sigma(r(\mathbf{len} \ r))$$

C.4 Pipeline Actions

C.4.1 Simple Pump and Valve Actions

- 755 Pumps may be set to pumping or reset to not pumping irrespective of the pump state.
- 756 Valves may be set to be open or to be closed irrespective of the valve state.
- 757 In setting or resetting a pump or a valve a desirable property may be lost.

value

```
755 pump_to_pump, pump_to_not_pump: P → N → N
756 valve_to_open, valve_to_close: V → N → N
```

value

```
755 pump_to_pump(p)(n) as n'
755   pre p ∈ obs_Us(n)
755   post let p':P•obs_UI(p)=obs_UI(p') in
755         obs_PΣ(p')=pumping^else_equal(n,n')(p,p') end
755 pump_to_not_pump(p)(n) as n'
755   pre p ∈ obs_Us(n)
755   post let p':P•obs_UI(p)=obs_UI(p') in
755         obs_PΣ(p')=not_pumping^else_equal(n,n')(p,p') end
756 valve_to_open(v)(n) as n'
756   pre v ∈ obs_Us(n)
756   post let v':V•obs_UI(v)=obs_UI(v') in
756         obs_VΣ(v')=open^else_equal(n,n')(v,v') end
756 valve_to_close(v)(n) as n'
756   pre v ∈ obs_Us(n)
756   post let v':V•obs_UI(v)=obs_UI(v') in
756         obs_VΣ(v')=close^else_equal(n,n')(v,v') end
```

value

```
else_equal: (N×N) → (U×U) → Bool
else_equal(n,n')(u,u') ≡
  obs_UI(u)=obs_UI(u')
  ∧ u ∈ obs_Us(n) ∧ u' ∈ obs_Us(n')
  ∧ omit_Σ(u)=omit_Σ(u')
  ∧ obs_Us(n)\{u}=obs_Us(n)\{u'}
  ∧ ∀ u'':U•u'' ∈ obs_Us(n)\{u} ≡ u'' ∈ obs_Us(n')\{u'}
```

omit_Σ: U → U_{no_state} --- "magic" function

≡: U_{no_state} × U_{no_state} → Bool

axiom

```
∀ u,u':U•omit_Σ(u)=omit_Σ(u') ≡ obs_UI(u)=obs_UI(u')
```

C.4.2 Events

Unit Handling Events

758 Let n be any acyclic net.

758. If there exists p, p', v, v' , pairs of distinct pumps and distinct valves of the net,

758. and if there exists a route, r , of length two or more of the net such that

759 all units, u , of the route, except its first and last unit, are pipes, then

760 if the route "spans" between p and p' and the *simple desirable property*, $\text{sppr}(r)$, does not hold for the route, then we have a possibly undesirable event — that occurred as soon as $\text{sppr}(r)$ did not hold;

761 if the route "spans" between p and v and the *simple desirable property*, $\text{spvr}(r)$, does not hold for the route, then we have a possibly undesirable event;

762 if the route “spans” between v and p and the *simple desirable property*, $svpr(r)$, does not hold for the route, then we have a possibly undesirable event; and

763 if the route “spans” between v and v' and the *simple desirable property*, $svvr(r)$, does not hold for the route, then we have a possibly undesirable event.

events:

758 $\forall n:N \cdot \text{acyclic}(n) \wedge$
 758 $\exists p,p':P,v,v':V \cdot \{p,p',v,v'\} \subseteq \text{obs_Us}(n) \Rightarrow$
 758 $\wedge \exists r:R \cdot \text{routes}(n) \wedge$
 759 $\forall u:U \cdot u \in \text{elems}(r) \setminus \{\text{hd } r, r(\text{len } r)\} \Rightarrow \text{is_II}(i) \Rightarrow$
 760 $p=\text{hd } r \wedge p'=r(\text{len } r) \Rightarrow \sim \text{spvr_prop}(r) \wedge$
 761 $p=\text{hd } r \wedge v=r(\text{len } r) \Rightarrow \sim \text{svpr_prop}(r) \wedge$
 762 $v=\text{hd } r \wedge p=r(\text{len } r) \Rightarrow \sim \text{svvr_prop}(r) \wedge$
 763 $v=\text{hd } r \wedge v'=r(\text{len } r) \Rightarrow \sim \text{svvr_prop}(r)$

Foreseeable Accident Events

A number of foreseeable accidents may occur.

764 A unit ceases to function, that is,
 a a unit is clogged,
 b a valve does not open or close,
 c a pump does not pump or stop pumping.

765 A unit gives rise to excessive leakage.

766 A well becomes empty or a sunk becomes full.

767 A unit, or a connected net of units gets on fire.

768 Or a number of other such “accident”.

C.4.3 Well-formed Operational Nets

769 A well-formed operational net

770 is a well-formed net

- a with at least one well, w , and at least one sink, s ,
- b and such that there is a route in the net between w and s .

value

769 $\text{wf_OpN}: N \rightarrow \mathbf{Bool}$

769 $\text{wf_OpN}(n) \equiv$

770 $\text{satisfies axiom 724 on Page 310} \wedge \text{acyclic}(n): \text{Item 729 on Page 310} \wedge$

770 $\text{wfN_SP}(n): \text{satisfies flow laws, 749 on Page 313 and 750 on Page 313} \wedge$

770a $\exists w:W,s:S \cdot \{w,s\} \subseteq \text{obs_Us}(n) \Rightarrow$

770b $\exists r:R \cdot \langle w \rangle \hat{r} \langle s \rangle \in \text{routes}(n)$

C.4.4 Orderly Action Sequences

Initial Operational Net

771 Let us assume a notion of an initial operational net.

772 Its pump and valve units are in the following states

- a all pumps are not_pumping, and

b all valves are closed.

value

```

771 initial_OpN: N → Bool
772 initial_OpN(n) ≡ wf_OpN(n) ∧
772a  ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
772b  ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed

```

Oil Pipeline Preparation and Engagement

- 773 We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.
- We assume that the underlying net is operational wrt. w and s , that is, that there is a route, r , from w to s .
 - Now, an orderly action sequence for engaging route r is to “work backwards”, from s to w
 - setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable *sppr*, *spvr*, *svpr* and *svvr* properties. Finally, setting the pump adjacent to the (preceding) well starts the system.

value

```

773 prepare_and_engage: W × S → N → N
773 prepare_and_engage(w,s)(n) ≡
773a  let r:R • ⟨w⟩∧⟨s⟩ ∈ routes(n) in
773b  action_sequence(⟨w⟩∧⟨s⟩)(len⟨w⟩∧⟨s⟩)(n) end
773  pre ∃ r:R • ⟨w⟩∧⟨s⟩ ∈ routes(n)

773c action_sequence: R → Nat → N → N
773c action_sequence(r)(i)(n) ≡
773c  if i=1 then n else
773c  case r(i) of
773c    mkV(⟦) → action_sequence(r)(i-1)(valve_to_open(r(i))(n)),
773c    mkP(⟦) → action_sequence(r)(i-1)(pump_to_pump(r(i))(n)),
773c    _ → action_sequence(r)(i-1)(n)
773c  end end

```

C.4.5 Emergency Actions

- 774 If a unit starts leaking excessive oil
- then nearest up-stream valve(s) must be closed,
 - and any pumps in-between this (these) valves and the leaking unit must be set to *not_pumping* — following an orderly sequence.
- 775 If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold
- then — a ha !
 - Left as an exercise.

C.5 Connectors

The interface, that is, the possible “openings”, between adjacent units have not been explored. Likewise the for the possible “openings” of “begin” or “end” units, that is, units not having their input(s), respectively

their “output(s)” connected to anything, but left “exposed” to the environment. We now introduce a notion of connectors: abstractly you may think of connectors as concepts, and concretely as “fittings” with bolts and nuts, or “weldings”, or “plates” inserted onto “begin” or “end” units.

776 There are connectors and connectors have unique connector identifiers.

777 From a connector one can observe its unique connector identifier.

778 From a net one can observe all its connectors

779 and hence one can extract all its connector identifiers.

780 From a connector one can observe a pair of “optional” (distinct) unit identifiers:

- a An optional unit identifier is
- b either a unit identifier of some unit of the net
- c or a ‘nil’ “identifier”.

781 In an observed pair of “optional” (distinct) unit identifiers

- there can not be two ‘nil’ “identifiers”.
- or the possibly two unit identifiers must be distinct

type

776 K, KI

value

777 $obs_KI: K \rightarrow KI$

778 $obs_Ks: N \rightarrow K\text{-set}$

779 $xtr_KIS: N \rightarrow KI\text{-set}$

779 $xtr_KIs(n) \equiv \{obs_KI(k) \mid k:K \cdot k \in obs_Ks(n)\}$

type

780 $oUlp' = (UI \mid \{ \mid nil \}) \times (UI \mid \{ \mid nil \})$

780 $oUlp = \{ \mid ouip: oUlp' \cdot wf_oUlp(ouip) \mid \}$

value

780 $obs_oUlp: K \rightarrow oUlp$

781 $wf_oUlp: oUlp' \rightarrow \mathbf{Bool}$

781 $wf_oUlp(uon, uon') \equiv$

781 $uon = nil \Rightarrow uon' \neq nil \vee uon' = nil \Rightarrow uon \neq nil \vee uon \neq uon'$

782 Under the assumption that a fork unit cannot be adjacent to a join unit

783 we impose the constraint that no two distinct connectors feature the same pair of actual (distinct) unit identifiers.

784 The first proper unit identifier of a pair of “optional” (distinct) unit identifiers must identify a unit of the net.

785 The second proper unit identifier of a pair of “optional” (distinct) unit identifiers must identify a unit of the net.

axiom

782 $\forall n: N, u, u': U \cdot \{u, u'\} \subseteq obs_Us(n) \wedge adj(u, u') \Rightarrow \sim(is_F(u) \wedge is_J(u'))$

783 $\forall k, k': K \cdot obs_KI(k) \neq obs_KI(k') \Rightarrow$
case $(obs_oUlp(k), obs_oUlp(k'))$ **of**
 $((nil, ui), (nil, ui')) \rightarrow ui \neq ui'$,
 $((nil, ui), (ui', nil)) \rightarrow \mathbf{false}$,
 $((ui, nil), (nil, ui')) \rightarrow \mathbf{false}$,
 $((ui, nil), (ui', nil)) \rightarrow ui \neq ui'$,
 $_ \rightarrow \mathbf{false}$
end

```

 $\forall n:N, k:K \cdot k \in \text{obs\_Ks}(n) \Rightarrow$ 
  case  $\text{obs\_oUlp}(k)$  of
    784    $(ui, \text{nil}) \rightarrow \exists UI(ui)(n)$ 
    785    $(\text{nil}, ui) \rightarrow \exists UI(ui)(n)$ 
    784-785  $(ui, ui') \rightarrow \exists UI(ui)(n) \wedge \exists UI(ui')(n)$ 
  end
value
   $\exists UI: UI \rightarrow N \rightarrow \mathbf{Bool}$ 
   $\exists UI(ui)(n) \equiv \exists u:U \cdot u \in \text{obs\_Us}(n) \wedge \text{obs\_UI}(u) = ui$ 

```

C.6 On Temporal Aspects of Pipelines

The $\text{else_qual}(u, u')(n, n')$ function definition represents a gross simplification. It ignores the actual flow which changes as a result of setting alternate states, and hence the net state. We now wish to capture the dynamics of flow. We shall do so using the Duration Calculus — a continuous time, integral temporal logic that is semantically and proof system “integrated” with RSL:

Zhou ChaoChen and Michael Reichhardt Hansen
 Duration Calculus: A Formal Approach to Real-time Systems
 Monographs in Theoretical Computer Science
 The EATCS Series
 Springer 2004

C.7 A CSP Model of Pipelines

We recapitulate Sect. C.5 — now adding connectors to our model:

```

786 From an oil pipeline system one can observe units and connectors.
787 Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
788 Units and connectors have unique identifiers.
789 From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units
    that the connector connects.

```

```

type
786 OPLS, U, K
788 UI, KI
value
786  $\text{obs\_Us}: \text{OPLS} \rightarrow \mathbf{U\text{-set}}$ ,  $\text{obs\_Ks}: \text{OPLS} \rightarrow \mathbf{K\text{-set}}$ 
787  $\text{is\_WeU}$ ,  $\text{is\_PiU}$ ,  $\text{is\_PuU}$ ,  $\text{is\_VaU}$ ,
787  $\text{is\_JoU}$ ,  $\text{is\_FoU}$ ,  $\text{is\_SiU}: U \rightarrow \mathbf{Bool}$  [mutually exclusive]
788  $\text{obs\_UI}: U \rightarrow \text{UI}$ ,  $\text{obs\_KI}: K \rightarrow \text{KI}$ 
789  $\text{obs\_Ulp}: K \rightarrow (\text{UI} \setminus \{\text{nil}\}) \times (\text{UI} \setminus \{\text{nil}\})$ 

```

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

```

790 There is given an oil pipeline system,  $\text{opls}$ .
791 To every unit we associate a CSP behaviour.
792 Units are indexed by their unique unit identifiers.

```

- 793 To every connector we associate a CSP channel.
 Channels are indexed by their unique "k" onnector identifiers.
 794 Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.
 795 Channels, in this model, have no state.
 796 Unit behaviours communicate with neighbouring units — those with which they are connected.
 797 Unit functions, \mathcal{U}_i , change the unit state.
 798 The pipeline system is now the parallel composition of all the unit behaviours.

Editorial Remark: Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise. The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink. The literal **Unit**, in a function signature, before the \rightarrow "announces" that the function takes no argument.² The literal **Unit**, in a function signature, after the \rightarrow "announces", as used here, that the function never terminates.

value

790 opls:OPLS

channel

793 $\{ch[k_i] | k:K, k:K \cdot k \in obs_Ks(opls) \wedge ki=obs_Kl(k)\}$ M

value

798 pipeline_system: **Unit** \rightarrow **Unit**

798 pipeline_system() \equiv

791 $\parallel \{unit(ui)(u) | u:U \cdot u \in obs_Us(opls) \wedge ui=obs_Ul(u)\}$

792 unit: ui:Ul \rightarrow U \rightarrow

796 **in, out** $\{ch[k_i] | k:K, ki:Kl \cdot k \in obs_Ks(opls) \wedge ki=obs_Kl(k) \wedge$

796 $\quad \mathbf{let} (ui', ui'')=obs_Ulp(k) \mathbf{in} ui \in \{ui', ui''\} \setminus \{nil\} \mathbf{end}\}$ **Unit**

794 $unit(ui)(u) \equiv \mathbf{let} u' = \mathcal{U}_i(ui)(u) \mathbf{in} unit(ui)(u') \mathbf{end}$

797 $\mathcal{U}_i: ui:Ul \rightarrow$ U \rightarrow

797 **in, out** $\{ch[k_i] | k:K, ki:Kl \cdot k \in obs_Ks(opls) \wedge ki=obs_Kl(k) \wedge$

797 $\quad \mathbf{let} (ui', ui'')=obs_Ulp(k) \mathbf{in} ui \in \{ui', ui''\} \setminus \{nil\} \mathbf{end}\}$ U

C.8 Conclusion

We have shown draft sketches of aspects of gas/oil pipelines. From a comprehensive such domain description we can systematically "derive" a set of complementary or alternative requirements prescriptions for the monitoring and control of individual pipe units, as well as of consolidated pipelines. Etcetera !

² **Unit** is a type name; () is the only value of type **Unit**.

D

A Document System

We domain analyse and suggest a description of a domain of documents. We emphasize that the model is one of several possible. Common to these models is that we model “all” we can say about documents – irrespective of whether it can also be “implemented”! The model(s) are not requirements prescriptions – but we can develop such from our domain description.

Yiu may find that the model is overly detailed with respect to a number of “operations” and properties of documents. We find that these operations must be part of the very basis of a document domain in order to cope with documents such as they occur in, for example, public government, see Appendix sect. D.17, or in urban planning, see Appendix Sect. D.18.

D.1 Introduction

We analyse a notion of documents. Documents such as they occur in daily life. What can we say about documents – regardless of whether we can actually provide compelling evidence for what we say! That is: we model documents, not as electronic entities — which they are becoming, more-and-more, but as if they were manifest entities. When we, for example, say that “*this document was recently edited by such-and-such and the changes of that editing with respect to the text before is such-and-such*”, then we can, of course, always claim so, even if it may be difficult or even impossible to verify the claim. It is a fact, although maybe not demonstrably so, that there was a version of any document before an edit of that document. It is a fact that some handler did the editing. It is a fact that the editing took place at (or in) exactly such-and-such a time (interval), etc. We model such facts.

This research note unravels its analysis &¹ description in stages.

D.2 A System for Managing, Archiving and Handling Documents

The title of this section: *A System for Managing, Archiving and Handling Documents* immediately reveals the major concepts: That we are dealing with a *system* that **manages**, **archives** and **handles documents**. So what do we mean by **managing**, **archiving** and **handling** documents, and by **documents**? We give an ultra short survey. The survey relies on your prior knowledge of what you think documents are! **Management** decides² to direct **handlers** to work on **documents**. **Management** first directs the document archive to **create documents**. The document **archive creates documents**, as requested by **management**, and informs management of the **unique document identifiers** (by means of which handlers can handle

¹ We use the logogram & between two terms, A & B, when we mean to express one meaning.

² How these decisions come about is not shown in this research note – as it has nothing to do with the essence of document handling, but, perhaps, with ‘management’.

these documents). **Management** then **grants** its designated **handler(s)** **access rights** to **documents**, these access rights enable handlers to **edit, read** and **copy** documents. The **handlers'** **editing** and **reading** of **documents** is accomplished by the **handlers** “working directly” with the **documents** (i.e., synchronising and communicating with **document behaviours**). The **handlers'** **copying** of **documents** is accomplished by the **handlers** requesting **management**, in collaboration with the **archive** behaviour, to do so.

D.3 Principal Endurants

By an *endurant* we shall understand “an entity that can be observed or conceived and described as a ”complete thing” at no matter which given snapshot of time.” Were we to “freeze” time we would still be able to observe the entire enduring. This characterisation of what we mean by an ‘endurant’ is from [70, Manifest Domains: Analysis & Description]. We begin by identifying the principal enduring.

799 From document handling systems one can observe aggregates of handlers and documents.

We shall refer to ‘aggregates of handlers’ by M, for management, and to ‘aggregates of documents’ by A, for archive.

800 From aggregates of handlers (i.e., M) we can observe sets of handlers (i.e., H).

801 From aggregates of documents (i.e., A) we can observe sets of documents (i.e., D).

type

799 S, M, A

value

799 obs_M: S → M

799 obs_A: S → A

type

800 H, Hs = H-set

801 D, Ds = D-set

value

800 obs_Hs: M → Hs

801 obs_Ds: A → Ds

D.4 Unique Identifiers

The notion of unique identifiers is treated, at length, in [70, Manifest Domains: Analysis & Description].

802 We associate unique identifiers with aggregate, handler and document enduring.

803 These can be observed from respective parts³.

type

802 MI⁴, AI⁵, HI, DI

value

803 uid_MI⁶: M → MI

803 uid_AI⁷: A → AI

803 uid_HI: H → HI

803 uid_DI: D → DI

³ [70, Manifest Domains: Analysis & Description] explains how ‘parts’ are the discrete enduring with which we associate the full complement of properties: unique identifiers, mereology and attributes.

As reasoned in [70, Manifest Domains: Analysis & Description], the unique identifiers of endurant parts are indeed unique: No two parts, whether composite, as are the aggregates, or atomic, as are handlers and documents, can have the same unique identifiers.

D.5 Documents: A First View

A document is a written, drawn, presented, or memorialized representation of thought. The word originates from the Latin *documentum*, which denotes a “teaching” or “lesson”.⁸ We shall, for this research note, take a document in its written and/or drawn form. In this section we shall survey the concept a documents.

D.5.1 Document Identifiers

Documents have *unique identifiers*. If two or more documents have the same document identifier then they are the same, one (and not two or more) document(s).

D.5.2 Document Descriptors

With documents we associate *document descriptors*. We do not here stipulate what document descriptors are other than saying that when a document is **created** it is provided with a descriptor and this descriptor “remains” with the document and never changes value. In other words, it is a static attribute.⁹ We do, however, include, in document descriptors, that the document they describe was initially based on a set of zero, one or more documents – identified by their unique identifiers.

D.5.3 Document Annotations

With documents we also associate *document annotations*. By a document annotation we mean a programmable attribute, that is, an attribute which can be ‘augmented’ by document handlers. We think of document annotations as “incremental”, that is, as “adding” notes “on top of” previous notes. Thus we shall model document annotations as a repository: notes are added, i.e., annotations are augmented, previous notes are not edited, and no notes are deleted. We suggest that notes be time-stamped. The notes (of annotations) may be such which record handlers work on documents. Examples could be: “November 24, 2019: 13:16: This is version V.”, “This document was released on November 24, 2019: 13:16.”, “November 24, 2019: 13:16: Section X.Y.Z of version III was deleted.”, “November 24, 2019: 13:16: References to documents *doc_i* and *doc_j* are inserted on Pages *p* and *q*, respectively.” and “November 24, 2019: 13:16: Final release.”

D.5.4 Document Contents: Text/Graphics

The main idea of a document, to us, is the *written* (i.e., text) and/or *drawn* (i.e., graphics) *contents*. We do not characterise any format for this *contents*. We may wish to insert, in the *contents*, references to locations in the *contents* of other documents. But, for now, we shall not go into such details. The main operations on documents, to us, are concerned with: their **creation, editing, reading, copying** and **shredding**. The **editing** and **reading** operations are mainly concerned with document *annotations* and *text/graphics*.

⁴ We shall not, in this research note, make use of the (one and only) management identifier.

⁵ We shall not, in this research note, make use of the (one and only) archive identifier.

⁶ Cf. Footnote 4: hence we shall not be using the `uid_MI` observer.

⁷ Cf. Footnote 5: hence we shall not be using the `uid_AI` observer.

⁸ From: <https://en.wikipedia.org/wiki/Document>

⁹ You may think of a document descriptor as giving the document a title; perhaps one or more authors; perhaps a physical address (of, for example, these authors); an initial date; as expressing whether the document is a research, or a technical report, or other; who is issuing the document (a public institution, a private firm, an individual citizen, or other); etc.

D.5.5 Document Histories

So documents are **created**, **edited**, **read**, **copied** and **shredded**. These operations are initiated by the management (**create**), by the archive (**create**), and by handlers (**edit**, **read**, **copy**), and at specific times.

D.5.6 A Summary of Document Attributes

- 804 As separate attributes of documents we have document descriptors, document annotations, document contents and document histories.
- 805 Document annotations are lists of document notes.
- 806 Document histories are lists of time-stamped document operation designators.
- 807 A document operation designator is either a create, or an edit, or a read, or a copy, or a shred designator.
- 808 A create designator identifies
- a a handler and a time (at which the create request first arose), and presents
 - b elements for constructing a document descriptor, one which
 - i besides some further undefined information
 - ii refers to a set of documents (i.e., embeds reference to their unique identifiers),
 - c a (first) document note, and
 - d an empty document contents.
- 809 An edit designator identifies a handler, a time, and specifies a pair of edit/undo functions.
- 810 A read designator identifies a handler.
- 811 A copy designator identifies a handler, a time, the document to be copied (by its unique identifier, and a document note to be inserted in both the master and the copy document).
- 812 A shred designator identifies a handler.
- 813 An edit function takes a triple of a document annotation, a document note and document contents and yields a pair of a document annotation and a document contents.
- 814 An undo function takes a pair of a document note and document contents and yields a triple of a document annotation, a document note and a document contents.
- 815 Proper pairs of (edit,undo) functions satisfy some inverse relation.

There is, of course, no need, in any document history, to identify the identifier of that document.

type

804 DD, DA, DC, DH

value

804 attr_DD: $D \rightarrow DD$

804 attr_DA: $D \rightarrow DA$

804 attr_DC: $D \rightarrow DC$

804 attr_DH: $D \rightarrow DH$

type

805 DA = DN*

806 DH = (TIME × DO)*

807 DO == Crea | Edit | Read | Copy | Shre

808 Crea :: (HI × TIME) × (DI-set × Info) × DN × {"empty_DC"}

808(b)i Info = ...

value

808(b)ii embed_DIs.in_DD: DI-set × Info → DD

axiom

808d "empty_DC" ∈ DC

type

809 Edit :: (HI × TIME) × (EDIT × UNDO)

810 Read :: (HI × TIME) × DI

```

811 Copy :: (HI × TIME) × DI × DN
812 Shre :: (HI × TIME) × DI
813 EDIT = (DA × DN × DC) → (DA × DC)
814 UNDO = (DA × DC) → (DA × DN × DC)
axiom
815 ∇ mkEdit(⌊_,(e,u)):Edit •
815   ∇ (da,dn,dc):(DA×DN×DC) •
815     u(e(da,dn,dc))=(da,dn,dc)

```

D.6 Behaviours: An Informal, First View

In [70, Manifest Domains: Analysis & Description] we show that we can associate behaviours with parts, where parts are such discrete endurants for which we choose to model all its observable properties: unique identifiers, mereology and attributes, and where behaviours are sequences of actions, events and behaviours.

- The overall document handler system behaviour can be expressed in terms of the parallel composition of the behaviours
 - 816 of the system core behaviour,
 - 817 of the handler aggregate (the management) behaviour
 - 818 and the document aggregate (the archive) behaviour,
 - with the (distributed) parallel composition of
 - 819 all the behaviours of handlers and,
 - the (distributed) parallel composition of
 - 820 at any one time, zero, one or more behaviours of documents.
- To express the latter
 - 821 we need introduce two “global” values: an indefinite set of handler identifiers and an indefinite set of document identifiers.

value

```

821 his:HI-set, dis:DI-set

816 sys(...)
817 || mgmt(...)
818 || arch(...)
819 || ||{hdlri(...)}|i:HI•i∈his}
820 || ||{docui(dd)(da,dc,dh)}|i:DI•i∈dis}

```

For now we leave undefined the arguments, (...) etc., of these behaviours. The arguments of the document behaviour, (dd)(da,dc,dh), are the static, respectively the three programmable (i.e., dynamic) attributes: *document descriptor*, *document annotation*, *document contents* and *document history*. The above expressions, Items 817–820, do not define anything, they can be said to be “snapshots” of a “behaviour state”. Initially there are no document behaviours, docu_i(dd)(da,dc,dh), Item 820. Document behaviours are “started” by the archive behaviour (on behalf of the management and the handler behaviours). Other than mentioning the system (core) behaviour we shall not model that behaviour further.

D.7 Channels, A First View

Channels are means for behaviours to synchronise and communicate values (such as unique identifiers, mereologies and attributes).

- 822 The management behaviour, *mgmt*, need to (synchronise and) communicate with the archive behaviour, *arch*, in order, for the management behaviour, to request the archive behaviour
- to **create** (ab initio or due to **copying**)
 - or **shred** document behaviours, docu_j , and for the archive behaviour
 - to inform the management behaviour of the identity of the document(behaviour)s that it has created.

channel822 mgmt_arch_ch:MA

- 823 The management behaviour, *mgmt*, need to (synchronise and) communicate with all handler behaviours, hdlr_i and they, in turn, to (synchronised) communicate with the handler management behaviour, *mgmt*. The management behaviour need to do so in order
- to inform a handler behaviour that it is granted access rights to a specific document, subsequently these access rights may be modified, including revoked.

channel823 $\{\text{mgmt_hdlr_ch}[i]:\text{MH}|i:\text{HI}\cdot i \in \text{his}\}$

- 824 The document archive behaviour, *arch*, need (synchronise and) communicate with all document behaviours, docu_j and they, in turn, to (synchronise and) communicate with the archive behaviour, *arch*.

channel824 $\{\text{arch_docu_ch}[j]:\text{AD}|h:\text{DI}\cdot j \in \text{dis}\}$

- 825 Handler behaviours, hdlr_i , need (synchronise and) communicate with all the document behaviours, docu_j , with which it has operational allowance to so do so¹⁰, and document behaviours, docu_j , need (synchronise and) communicate with potentially all handler behaviours, hdlr_i , namely those handler behaviours, hdlr_i with which they have (“earlier” synchronised and) communicated.

channel825 $\{\text{hdlr_docu_ch}[i,j]:\text{HD}|i:\text{HI},j:\text{DI}\cdot i \in \text{his}\wedge j \in \text{dis}\}$

- 826 At present we leave undefined the type of messages that are communicated.

type

826 MA, MH, AD, HD

D.8 An Informal Graphical System Rendition

Figure D.1 on the next page is an informal rendition of the “state” of a number of behaviours: a single management behaviour, a single archive behaviour, a fixed number, n_h , of one or more handler behaviours, and a variable, initially zero number of document behaviours, with a maximum of these being n_d . The figure also indicates, again rather informally, the channels between these behaviours: one channel between the management and the archive behaviours; n_h channels (n_h is, again, informally indicated) between the management behaviour and the n_h handler behaviours; n_d channels (n_d is, again, informally indicated) between the archive behaviour and the n_d document behaviours; and $n_h \times n_d$ channels ($n_h \times n_d$ is, again, informally indicated) between the n_h handler behaviours and the n_d document behaviours

¹⁰ The notion of operational allowance will be explained below.

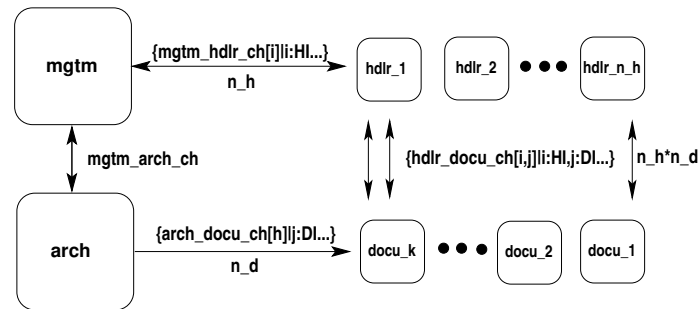


Fig. D.1. An Informal Snapshot of System Behaviours

D.9 Behaviour Signatures

- 827 The `mgmt` behaviour (synchronises and) communicates with the archive behaviour and with all of the handler behaviours, hdr_i .
- 828 The archive behaviour (synchronises and) communicates with the `mgmt` behaviour and with all of the document behaviours, docu_j .
- 829 The signature of the generic handler behaviours, hdr_i expresses that they [occasionally] receive “orders” from management, and otherwise [regularly] interacts with document behaviours.
- 830 The signature of the generic document behaviours, docu_j expresses that they [occasionally] receive “orders” from the archive behaviour and that they [regularly] interacts with handler behaviours.

value

- 827 `mgmt`: ... \rightarrow **in,out** `mgmt_arch_ch`, $\{\text{mgmt_hdr_ch}[i]|i:HI \cdot i \in \text{his}\}$ **Unit**
- 828 `arch`: ... \rightarrow **in,out** `mgmt_arch_ch`, $\{\text{arch_docu_ch}[j]|j:DI \cdot j \in \text{dis}\}$ **Unit**
- 829 hdr_i : ... \rightarrow **in** `mgmt_hdr_ch[i]`, **in,out** $\{\text{hdr_docu_ch}[i,j]|j:DI \cdot j \in \text{dis}\}$ **Unit**
- 830 docu_j : ... \rightarrow **in** `mgmt_arch_ch`, **in,out** $\{\text{hdr_docu_ch}[i,j]|i:HI \cdot i \in \text{his}\}$ **Unit**

D.10 Time

D.10.1 Time and Time Intervals: Types and Functions

- 831 We postulate a notion of time, one that covers both a calendar date (from before Christ up till now and beyond). But we do not specify any concrete type (i.e., format such as: YY:MM:DD, HH:MM:SS).
- 832 And we postulate a notion of (signed) time interval — between two times (say: $\pm YY:MM:DD:HH:MM:SS$).
- 833 Then we postulate some operations on time: Adding a time interval to a time obtaining a time; subtracting one time from another time obtaining a time interval, multiplying a time interval with a natural number; etc.
- 834 And we postulate some relations between times and between time intervals.

type

- 831 `TIME`
- 832 `TIME_INTERVAL`

value

- 833 `add`: `TIME_INTERVAL` \times `TIME` \rightarrow `TIME`

```

833 sub: TIME × TIME → TIME_INTERVAL
833 mpy: TIME_INTERVAL × Nat → TIME_INTERVAL
834 <,≤,=,≠,≥,>: ((TIME×TIME)|(TIME_INTERVAL×TIME_INTERVAL)) → Bool

```

D.10.2 A Time Behaviour and a Time Channel

- 835 We postulate a[n “ongoing”] time behaviour: it either keeps being a time behaviour with unchanged time, t , or – internally non-deterministically – chooses being a time behaviour with a time interval incremented time, $t+ti$, or – internally non-deterministically – chooses to [first] offer its time on a [global] channel, $time_ch$, then resumes being a time behaviour with unchanged time., t
- 836 The time interval increment, ti , is likewise internally non-deterministically chosen. We would assume that the increment is “infinitesimally small”, but there is no need to specify so.
- 837 We also postulate a channel, $time_ch$, on which the time behaviour offers time values to whoever so requests.

value

```

835 time: TIME → time_ch TIME Unit
835 time(t) ≡ (time(t) [] time(t+ti) [] time_ch!t ; time(t))
836 ti:TIME_INTERVAL ...

```

channel

```

837 time_ch:TIME

```

D.10.3 An Informal RSL Construct

The formal-looking specifications of this report appear in the style of the RAISE [132] Specification Language, RSL [131]. We shall be making use of an informal language construct:

- **wait** ti .

wait is a keyword; ti designates a time interval. A typical use of the wait construct is:

- ... ptA ; **wait** ti ; ptB ; ...

If at specification text point ptA we may assert that time is t , then at specification text point ptB we can assert that time is $t+ti$.

D.11 Behaviour “States”

We recall that the enduring parts, Management, Archive, Handlers, and Documents, have properties in the form of *unique identifiers*, *mereologies* and *attributes*. We shall not, in this research note, deal with possible mereologies of these enduring parts. In this section we shall discuss the enduring attributes of *mgtm* (management), *arch* (archive), *hdlrs* (handlers), and *docus* (documents). Together the values of these properties, notably the attributes, constitute states – and, since we associate behaviours with these enduring parts, we can refer to these states also as behaviour states. Some attributes are static, i.e., their value never changes. Other attributes are dynamic.¹¹ Document handling systems are rather conceptual, i.e., abstract in nature. The dynamic attributes, therefore, in this modeling “exercise”, are constrained to just the *programmable* attributes. Programmable attributes are those whose value is set by “their” behaviour. For a behaviour β we shall show the static attributes as one set of parameters and the programmable attributes as another set of parameters.

¹¹ We refer to Sect. 3.4 of [70], and in particular its subsection 3.4.4.

value β : Static \rightarrow Program \rightarrow ... **Unit**

- 838 For the management enduring/behaviour we focus on one programmable attribute. The management behaviour needs keep track of all the handlers it is charged with, and for each of these which zero, one or more documents they have been granted access to (cf. Sect. D.12.3 on the following page). Initially that management directory lists a number of handlers, by their identifiers, but with no granted documents.
- 839 For the archive behaviour we similarly focus on one programmable attribute. The archive behaviour needs keep track of all the documents it has used (i.e., created), those that are available (and not yet used), and of those it has shredded. Initially all these three archive directory sets are empty.
- 840 For the handler behaviour we similarly focus on one programmable attribute. The handler behaviour needs keep track of all the documents it has been charged with and its access rights to these.
- 841 Document attributes we mentioned above, cf. Items 804–807.

type

- 838 MDIR = HI \rightarrow_m (DI \rightarrow_m ANm-set)
 839 ADIR = avail:DI-set \times used:DI-set \times gone:DI-set
 840 HDIR = DI \rightarrow_m ANm-set
 841 SDATR = DD, PDATR = DA \times DC \times DH

axiom

- 839 \forall (avail,used,gone):ADIR \cdot avail \cap used = $\{\}$ \wedge gone \subseteq used

We can now “complete” the behaviour signatures. We omit, for now, static attributes.

value

- 827 mgmt: MDIR \rightarrow **in,out** mgmt_arch_ch, {mgmt_hdlr_ch[i]|i:HI*i* \in his} **Unit**
 828 arch: ADIR \rightarrow **in,out** mgmt_arch_ch, {arch_docu_ch[j]|j:DI*j* \in dis} **Unit**
 829 hdlr_i: HDIR \rightarrow **in** mgmt_hdlr_ch[i], **in,out** {hdlr_docu_ch[i,j]|j:DI*j* \in dis} **Unit**
 830 docu_j: SDATR \rightarrow PDATR \rightarrow **in** mgmt_arch_ch, **in,out** {hdlr_docu_ch[i,j]|i:HI*i* \in his} **Unit**

D.12 Inter-Behaviour Messages

Documents are not “fixed, innate” entities. They embody a “history”, they have a “past”. Somehow or other they “carry a trace of all the ”things” that have happened/occurred to them. And, to us, these things are the manipulations that management, via the archive and handlers perform on documents.

D.12.1 Management Messages with Respect to the Archive

- 842 Management **create** documents. It does so by requesting the archive behaviour to allocate a document identifier and initialize the document “state” and start a document behaviour, with initial information, cf. Item 808 on Page 324:
- the identity of the initial handler of the document to be created,
 - the time at which the request is being made,
 - a document descriptor which embodies a (finite) set of zero or more (used) document identifiers (dis),
 - a document annotation note dn, and
 - an initial, i.e., “empty” contents, "empty_DC".

type

808. Crea :: (HI \times TIME) \times (DI-set \times Info) \times DN \times {"empty_DC"} [cf. formula Item 808, Page 325]

843 The management behaviour passes on to the archive behaviour, requests that it accepts from handlers behaviours, for the copying of document:

843 Copy :: DI \times HI \times TIME \times DN [cf. Item 853 on the facing page]

844 Management **schreds** documents by informing the archive behaviour to do so.

type

844 Shred :: TIME \times DI

D.12.2 Management Messages with Respect to Handlers

845 Upon receiving, from the archive behaviour, the “feedback” the identifier of the created document (behaviour):

type

845. Create_Reply :: NewDocID(di:DI)

846 the management behaviour decides to **grant** access rights, acrs:ACRS¹², to a document handler, hi:HI.

type

846 Gran :: HI \times TIME \times DI \times ACRS

D.12.3 Document Access Rights

Implicit in the above is a notion of document access rights.

847 By document access rights we mean a set of action names.

848 By an action name we mean such tokens that indicate either of the document handler operations indicate above.

type

847 ACRS = ANm-set

848 ANm = {"edit","read","copy"}

D.12.4 Archive Messages with Respect to Management

To create a document management provides the archive with some initial information. The archive behaviour selects a document identifier that has not been used before.

849 The archive behaviour informs the management behaviour of the identifier of the created document.

type

849 NewDocID :: DI

D.12.5 Archive Message with Respect to Documents

850 To shred a document the archive behaviour must access the designated document in order to **stop** it. No “message”, other than a symbolic “stop”, need be communicated to the document behaviour.

type

850 Shred :: {"stop"}

¹² For the concept of access rights see Sect. D.12.3.

D.12.6 Handler Messages with Respect to Documents

Handlers, generically referred to by $hdlr_i$, may perform the following operations on documents: **edit**, **read** and **copy**. (Management, via the archive behaviour, **creates** and **shreds** documents.)

851 To perform an **edit** action handler $hdlr_i$ must provide the following:

- the document identity – in the form of a $(i:HI,j:DI)$ channel $hdlr_docu_ch$ index value,
- the handler identity, i ,
- the time of the edit request,
- and a pair of functions: one which performs the editing and one which un-does it !

type

851 Edit :: $DI \times HI \times TIME \times (EDIT \times UNDO)$

852 To perform a **read** action handler $hdlr_i$ must provide the following information:

- the document identity – in the form of a $di:DI$ channel $hdlr_docu_ch$ index value,
- the handler identity and
- the time of the read request.

type

852 Read :: $DI \times HI \times TIME$

D.12.7 Handler Messages with Respect to Management

853 To perform a **copy** action, a handler, $hdlr_i$, must provide the following information to the management behaviour, $mgtm$:

- the document identity,
- the handler identity – in the form of an $hi:HI$ channel $mgtm_hdlr_ch$ index value,
- the time of the copy request, and
- a document note (to be affixed both the master and the copy documents).

853 Copy :: $DI \times HI \times TIME \times DN$ [cf. Item 843 on the facing page]

How the handler, the management, the archive and the “named other” handlers then enact the copying, etc., will be outlined later.

D.12.8 A Summary of Behaviour Interactions

Figure D.2 on the next page summarises the sources, **out**, resp. !, and the targets, **in**, resp. ?, of the messages covered in the previous sections.

D.13 A General Discussion of Handler and Document Interactions

We think of documents being manifest. Either a document is in paper form, or it is in electronic form. In paper form we think of a document as being in only one – and exactly one – physical location. In electronic form a document is also in only one – and exactly one – physical location. No two handlers can access the same document at the same time or in overlapping time intervals. If your conventional thinking makes you think that two or more handlers can, for example, read the same document “at the same time”, then, in fact, they are reading either a master and a copy of that master, or they are reading two copies of a common master.

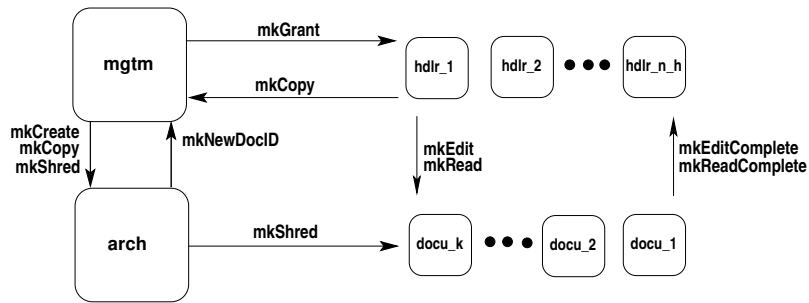


Fig. D.2. A Summary of Behaviour Interactions

D.14 Channels: A Final View

We can now summarize the types of the various channel messages first referred to in Items 822, 823, 824 and 825.

type

822 MA = Create (Item 842 on Page 329) | Shred (Item 842d on Page 330) | NewDocID (Item 849 on Page 330)

823 MH = Grant (Item 842c on Page 330) | Copy (Item 853 on the preceding page) |

824 AD = Shred (Item 850 on the previous page)

825 HD = Edit (Item 851 on the preceding page) | Read (Item 852 on the previous page) | Copy (Item 853 on the pre

D.15 An Informal Summary of Behaviours

D.15.1 The Create Behaviour: Left Fig. D.3 on the next page

854 [1] The management behaviour, at its own volition, initiates a create document behaviour. It does so by offering a create document message to the archive behaviour.

a [1.1] That message contains a meaningful document descriptor,

b [1.2] an initial document annotation,

c [1.3] an “empty” document contents and

d [1.4] a single element document history.

(We refer to Sect. D.12.1 on Page 329, Items 842–842e.)

855 [2] The archive behaviour offers to accept that management message. It then selects an available document identifier (here shown as k), henceforth marking k as used.

856 [3] The archive behaviour then “spawns off” document behaviour docu_k – here shown by the “dash-dotted” rounded edge square.

857 [4] The archive behaviour then offers the document identifier k message to the management behaviour. (We refer to Sect. D.12.4 on Page 330, Item 849.)

858 [5] The management behaviour then

a [5.1] selects a handler, here shown as i , i.e., hdlr_i ,

b [5.2] records that that handler is granted certain access rights to document k ,

c [5.3] and offers that granting to handler behaviour i .

(We refer to Sect. D.12.2 on Page 330, Item 846 on Page 330.)

859 [6] Handler behaviour i records that it now has certain access rights to document i .

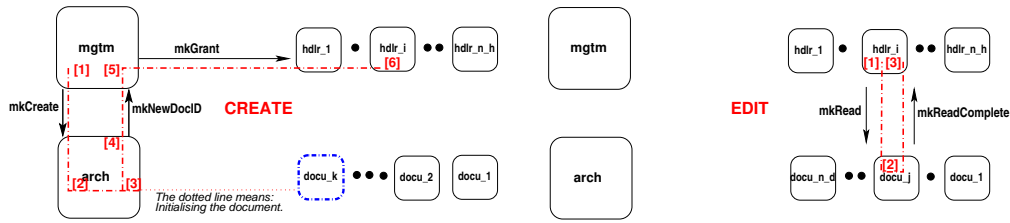


Fig. D.3. Informal Snapshots of Create and Edit Document Behaviours

D.15.2 The Edit Behaviour: Right Fig. D.3

- 1 Handler behaviour i , at its own volition, initiates an edit action on document j (where i has editing rights for document j). Handler i , optionally, provides document j with a(annotation) note. While editing document j handler i also “selects” an appropriate pair of *edit/undo* functions for document j .
- 2 Document behaviour j accepts the editing request, enacts the editing, optionally appends the (annotation) note, and, with handler i , completes the editing, after some time interval t_i .
- 3 Handler behaviour i completes its edit action.

D.15.3 The Read Behaviour: Left Fig. D.4 on the next page

- 1 Handler behaviour i , at its own volition, initiates a read action on document j (where i has reading rights for document j). Handler i , optionally, provides document j with a(annotation) note.
- 2 Document behaviour j accepts the reading request, enacts the reading by providing the handler, i , with the document contents, and optionally appends the (annotation) note, and, with handler i , completes the reading, after some time interval t_i .
- 3 Handler behaviour i completes its read action.

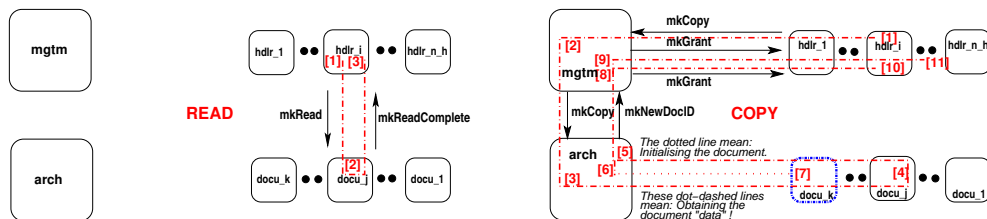


Fig. D.4. Informal Snapshots of Read and Copy Document Behaviours

D.15.4 The Copy Behaviour: Right Fig. D.4 on the following page

- 1 Handler behaviour i , at its own volition, initiates a copy action on document j (where i has copying rights for document j). Handler i , optionally, provides master document j as well as the copied document (yet to be identified) with respective (annotation) notes.
- 2 The management behaviour offers to accept the handler message. As for the create action, the management behaviour offers a combined *copy and create* document message to the archive behaviour.

- 3 The archive behaviour selects an available document identifier (here shown as k), henceforth marking k as used.
- 4 The archive behaviour then obtains, from the master document j its *document descriptor*, dd_j , its *document annotations*, da_j , its *document contents*, dc_j , and its *document history*, dh_j .
- 5 The archive behaviour informs the management behaviour of the identifier, k , of the (new) document copy,
- 6 while assembling the attributes for that (new) document copy: its *document descriptor*, dd_k , its *document annotations*, da_k , its *document contents*, dc_k , and its *document history*, dh_k , from these “similar” attributes of the master document j ,
- 7 while then “spawning off” document behaviour $docu_k$ – here shown by the “dash-dotted” rounded edge square.
- 8 The management behaviour accepts the identifier, k , of the (new) document copy, recording the identities of the handlers and their access rights to k ,
- 9 while informing these handlers (informally indicated by a “dangling” dash-dotted line) of their grants,
- 10 while also informing the master copy of the copy identity (etcetera).
- 11 The handlers granted access to the copy record this fact.

D.15.5 The Grant Behaviour: Left Fig. D.5

This behaviour has its

- 1 Item [1] correspond, in essence, to Item [9] of the copy behaviour – see just above – and
- 2 Item [2] correspond, in essence, to Item [11] of the copy behaviour.

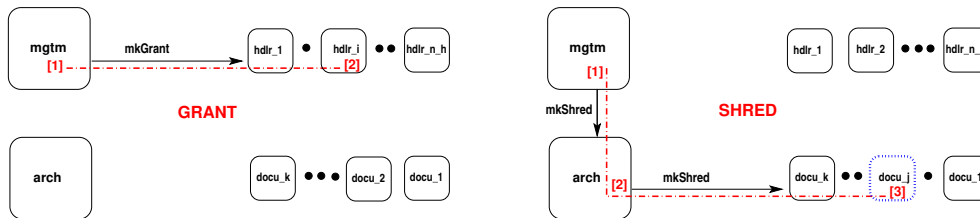


Fig. D.5. Informal Snapshots of Grant and Shred Document Behaviours

D.15.6 The Shred Behaviour: Right Fig. D.5 on the facing page

- 1 The management, at its own volition, selects a document, j , to be shredded. It so informs the archive behaviour.
- 2 The archive behaviour records that document j is to be no longer in use, but shredded, and informs document j 's behaviour.
- 3 The document j behaviour accepts the shred message and **stops** (indicated by the dotted rounded edge box).

D.16 The Behaviour Actions

To properly structure the definitions of the four kinds of (management, archive, handler and document) behaviours we single each of these out “across” the six behaviour traces informally described in

Sects. D.15.1–D.15.6. The idea is that if behaviour β is involved in τ traces, $\tau_1, \tau_2, \dots, \tau_\tau$, then behaviour β shall be defined in terms of τ non-deterministic alternative behaviours named $\beta_{\tau_1}, \beta_{\tau_2}, \dots, \beta_{\tau_\tau}$.

D.16.1 Management Behaviour

860 The management behaviour is involved in the following action traces:

- a **create**
- b **copy**
- c **grant**
- d **shred**

Fig. D.3 on Page 333 Left
 Fig. D.4 on the facing page Right
 Fig. D.5 on the preceding page Left
 Fig. D.5 on the facing page Right

value

```
860 mgtm: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
860 mgtm(mdir) ≡
860a   mgtm_create(mdir)
860b   [] mgtm_copy(mdir)
860c   [] mgtm_grant(mdir)
860d   [] mgtm_shred(mdir)
```

Management Create Behaviour: Left Fig. D.3 on Page 333

861 The **management create** behaviour

862 initiates a create document behaviour (i.e., a request to the archive behaviour),

863 and then awaits its response.

value

```
861 mgtm_create: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
861 mgtm_create(mdir) ≡
862 [1] let hi = mgtm_create_initiation(mdir) ; [Left Fig. D.3 on Page 333]
863 [5] mgtm_create_awaits_response(mdir)(hi) end [Left Fig. D.3 on Page 333]
```

The **management create initiation** behaviour

864 selects a handler on behalf of which it requests the document creation,

865 assembles the elements of the create message:

- by embedding a set of zero or more document references, dis, with some information, info, into a document descriptor, adding
- a document note, dn, and
- and initial, that is, empty document contents, "empty_DC",

866 offers such a create document message to the archive behaviour, and

867 yields the identifier of the chosen handler.

value

```
862 mgtm_create_initiation: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
862 mgtm_create_initiation(mdir) ≡
864   let hi:HI • hi ∈ dom mdir,
865 [1.2–.4] (dis,info):(DI-set×Info),dn:DN • is_meaningful(embed_DIs_in_DD(dis,info))(mdir) in
866 [1.1] mgtm_arch_ch ! mkCreate(embed_DIs_in_DD(ds,info),dn,"empty_DC")
867   hi end
```

865 is_meaningful: DD → MDIR → **Bool** [left further undefined]

The **management create awaits response** behaviour

- 868 starts by awaiting a reply from the archive behaviour with the identity, di , of the document (that that behaviour has created).
 869 It then selects suitable access rights,
 870 with which it updates its handler/document directory
 871 and offers to the chosen handler
 872 whereupon it resumes, with the updated management directory, being the management behaviour.

value

```

863 mgtm_create_awaits_response: MDIR → HI → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
863 mgtm_create_awaits_response(mdir) ≡
868 [5]      let mkNewDocID(di) = mgtm_arch_ch ? in
869 [5.1]    let acrs:ANm-set in
870 [5.2]    let mdir' = mdir † [hi ↦ [di ↦ acrs]] in
871 [5.3]    mgtm_hdlr_ch[hi] ! mkGrant(di,acrs)
872      mgtm(mdir') end end end

```

Management Copy Behaviour: Right Fig. D.4 on Page 334

- 873 The **management copy** behaviour
 874 accepts a copy document request from a handler behaviour (i.e., a request to the archive behaviour),
 875 and then awaits a response from the archive behaviour;
 876 after which it grants access rights to handlers to the document copy.

value

```

873 mgtm_copy: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
873 mgtm_copy(mdir) ≡
874 [2]  let hi = mgtm_accept_copy_request(mdir) in
875 [8]  let di = mgtm_awaits_copy_response(mdir)(hi) in
876 [9]  mgtm_grant_access_rights(mdir)(di) end end

```

- 877 The **management accept copy** behaviour non-deterministically externally (\square) awaits a copy request from a[ny] handler (i) behaviour –
 878 with the request identifying the master document, j , to be copied.
 879 The management accept copy behaviour forwards (!) this request to the archive behaviour –
 880 while yielding the identity of the requesting handler.

```

877. mgtm_accept_copy_request: MDIR → in,out mgtm_arch_ch, {mgtm_hdlr_ch[hi]|hi:HI•hi ∈ his} HI
877. mgtm_accept_copy_request(mdir) ≡
878.   let mkCopy(di,hi,t,dn) =  $\square$ {mgtm_hdlr_ch[i]?|i:HI•i ∈ his} in
879.   mgtm_arch_ch ! mkCopy(di,hi,t,dn) ;
879.   hi end

```

The **management awaits copy response** behaviour

- 881 awaits a reply from the archive behaviour as to the identity of the newly created copy (di) of master document j .
 882 The management awaits copy response behaviour then informs the ‘copying-requesting’ handler, hi , that the copying has been completed and the identity of the copy (di) –
 883 while yielding the identity, di , of the newly created copy.


```

860b. mgmtm_awaits_copy_response: MDIR → HI → in,out mgmtm_arch_ch, {mgmtm_hdlr_ch[hi]|hi:HI•hi ∈ his} DI
860b. mgmtm_awaits_copy_response(mdir)(hi) ≡
881. [8] let mkNewDocID(di) = mgmtm_arch_ch ? in
882.     mgmtm_hdlr_ch[hi] ! mkCopy(di) ;
883.     di end

```

The **management grants access rights** behaviour

884 selects suitable access rights for a suitable number of selected handlers.
885 It then offers these to the selected handlers.

```

876. mgmtm_grant_access_rights: MDIR → DI → in,out {mgmtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
876. mgmtm_grant_access_rights(mdir)(di) ≡
884.     let diarm = [hi→acrs|hi:HI,acrs:ANm-set• hi ∈ dom mdir∧acrs⊆(diarm(hi))(di)] in
885.     || {mgmtm_hdlr_ch[hi]!mkGrant(hi,time_ch?,di,acrs) |
885.     hi:HI,acrs:ANm-set•hi ∈ dom diarm∧acrs⊆(diarm(hi))(di)} end

```

Management Grant Behaviour: Left Fig. D.5 on Page 334

The **management grant** behaviour

886 is a variant of the mgmtm_grant_access_rights function, Items 884–885.
887 The management behaviour selects a suitable subset of known handler identifiers, and
888 for these a suitable subset of document identifiers from which
889 it then constructs a map from handler identifiers to subsets of access rights.
890 With this the management behaviour then issues appropriate grants to the chosen handlers.

type

$$\text{MDIR} = \text{HI} \rightarrow_{\text{m}} (\text{DI} \rightarrow_{\text{m}} \text{ANm-set})$$

value

```

886 mgmtm_grant: MDIR → in,out {mgmtm_hdlr_ch[hi]|hi:HI•hi ∈ his} Unit
886 mgmtm_grant(mdir) ≡
887     let his ⊆ dom dir in
888     let dis ⊆ ∪{dom mdir(hi)|hi:HI•hi ∈ his} in
889     let diarm = [hi→acrs|hi:HI,di:DI,acrs:ANm-set• hi ∈ his∧di ∈ dis∧acrs⊆(diarm(hi))(di)] in
890     || {mgmtm_hdlr_ch[hi]!mkGrant(di,acrs) |
890     hi:HI,di:DI,acrs:ANm-set•hi ∈ dom diarm∧di ∈ dis∧acrs⊆(diarm(hi))(di)}
886     end end end

```

Management Shred Behaviour: Right Fig. D.5 on Page 334

The **management shred** behaviour

891 initiates a request to the archive behaviour.
892 First the management shred behaviour selects a document identifier (from its directory).
893 Then it communicates a shred document message to the archive behaviour;
894 then it notes the (to be shredded) document in its directory
895 whereupon the management shred behaviour resumes being the management behaviour.

value

```

891 mgtm_shred: MDIR → out mgtm_arch_ch Unit
891 mgtm_shred(mdir) ≡
892   let di:DI • is_suitable(di)(mdir) in
893   [1] mgtm_arch_ch ! mkShred(time_ch?,di) ;
894   let mdir' = [hi→mdir(hi)\{di}|hi:HI•hi ∈ dom mdir] in
895   mgtm(mdir') end end

```

D.16.2 Archive Behaviour

896 The archive behaviour is involved in the following action traces:

- a **create**
- b **copy**
- c **shred**

Fig. D.3 on Page 333 Left
 Fig. D.4 on Page 334 Right
 Fig. D.5 on Page 334 Right

type

839 ADIR = avail:DI-set × used:DI-set × gone:DI-set

axiom

839 $\forall (avail,used,gone):ADIR \cdot avail \cap used = \{\} \wedge gone \subseteq used$

value

```

896 arch: ADIR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
896a arch(adir) ≡
896a   arch_create(adir)
896b   [] arch_copy(adir)
896c   [] arch_shred(adir)

```

The Archive Create Behaviour: Left Fig. D.3 on Page 333

The **archive create** behaviour

- 897 accepts a request, from the management behaviour to create a document;
 898 it then selects an available document identifier;
 899 communicates this new document identifier to the management behaviour;
 900 while initiating a new document behaviour, $docu_{di}$, with the document descriptor, dd , the initial document annotation being the singleton list of the note, an , and the initial document contents, dc – all received from the management behaviour – and an initial document history of just one entry: the date of creation, all
 901 in parallel with resuming the archive behaviour with updated programmable attributes.

```

896a. arch_create: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
896a. arch_create(avail,used,gone) ≡
897.   [2] let mkCreate((hi,t),dd,an,dc) = mgmt_arch_ch ? in
898.   let di:DI•di ∈ avail in
899.   [4] mgmt_arch_ch ! mkNewDocID(di) ;
900.   [3]  $docu_{di}(dd)(\langle an \rangle, dc, \langle (date\_of\_creation) \rangle)$ 
901.   || arch(avail\{di},used∪{di},gone)
896a. end end

```

The Archive Copy Behaviour: Right Fig. D.4 on Page 334

The **archive copy** behaviour

- 902 accepts a copy document request from the management behaviour with the identity, j , of the master document;
 903 it communicates (the request to obtain all the attribute values of the master document, j) to that document behaviour;
 904 whereupon it awaits their communication (i.e., (dd,da,dc,dh));
 905 (meanwhile) it obtains an available document identifier,
 906 which it communicates to the management behaviour,
 907 while initiating a new document behaviour, $docu_{di}$, with the master document descriptor, dd , the master document annotation, and the master document contents, dc , and the master document history, dh (all received from the master document),
 908 in parallel with resuming the archive behaviour with updated programmable attributes.

```

896b. arch_copy: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
896b. arch_copy(avail,used,gone) ≡
902. [3] let mkDocID(j,hi) = mgmt_arch_ch ? in
903.     arch_docu_ch[j] ! mkReqAttrs() ;
904.     let mkAttrs(dd,da,dc,dh) = arch_docu_ch[j] ? in
905.     let di:DI • di ∈ avail in
906.     mgmt_arch_ch ! mkCopyDocID(di) ;
907. [6,7] docu_{di}(augment(dd,"copy",j,hi),augment(da,"copy",hi),dc,augment(dh,("copy",date_and_time,j,hi)))
908.     || arch(avail\{di},used∪{di},gone)
896b.     end end end
  
```

where we presently leave the [overloaded] augment functions undefined.

The Archive Shred Behaviour: Right Fig. D.5 on Page 334

The **archive shred** behaviour

- 909 accepts a shred request from the management behaviour.
 910 It communicates this request to the identified document behaviour.
 911 And then resumes being the archive behaviour, noting however, that the shredded document has been shredded.

```

896c. arch_shred: AATTR → in,out mgmt_arch_ch, {arch_docu_ch[di]|di:DI•di ∈ dis} Unit
896c. arch_shred(avail,used,gone) ≡
909. [2] let mkShred(j) = mgmt_arch_ch ? in
910.     arch_docu_ch[j] ! mkShred() ;
911.     arch(avail,used,gone∪{j})
896c.     end
  
```

D.16.3 Handler Behaviours

- 912 The handler behaviour is involved in the following action traces:

- a **create**
- b **edit**
- c **read**
- d **copy**

- Fig. D.3 on Page 333 Left
- Fig. D.3 on Page 333 Right
- Fig. D.4 on Page 334 Left
- Fig. D.4 on Page 334 Right

e **grant**

Fig. D.5 on Page 334 Left

value912 $\text{hdlr}_{hi}: \text{HATTRS} \rightarrow \mathbf{in, out} \text{ mgtm_hdlr_ch}[hi], \{\text{hdlr_docu_ch}[hi, di] | di:DI \cdot di \in \text{dis}\} \mathbf{Unit}$ 912 $\text{hdlr}_{hi}(\text{hattr}) \equiv$ 912a $\text{hdlr_create}_{hi}(\text{hattr})$ 912b $\sqcap \text{hdlr_edit}_{hi}(\text{hattr})$ 912c $\sqcap \text{hdlr_read}_{hi}(\text{hattr})$ 912d $\sqcap \text{hdlr_copy}_{hi}(\text{hattr})$ 912e $\sqcap \text{hdlr_grant}_{hi}(\text{hattr})$ **The Handler Create Behaviour: Left Fig. D.3 on Page 333**913 The **handler create** behaviour offers to accept the granting of access rights, acrs, to document di.

914 It accordingly updates its programmable hattr attribute;

915 and resumes being a handler behaviour with that update.

912a $\text{hdlr_create}_{hi}: \text{HATTRS} \times \text{HHIST} \rightarrow \mathbf{in, out} \text{ mgtm_hdlr_ch}[hi] \mathbf{Unit}$ 912a $\text{hdlr_create}_{hi}(\text{hattr}, \text{hhist}) \equiv$ 913 **let** $\text{mkGrant}(di, \text{acrs}) = \text{mgtm_hdlr_ch}[hi] ? \mathbf{in}$ 914 **let** $\text{hattr}' = \text{hattr} \dagger [hi \mapsto \text{acrs}] \mathbf{in}$ 915 $\text{hdlr_create}_{hi}(\text{hattr}', \text{augment}(\text{hhist}, \text{mkGrant}(di, \text{acrs}))) \mathbf{end end}$ **The Handler Edit Behaviour: Right Fig. D.3 on Page 333**

916 The handler behaviour, on its own volition, decides to edit a document, di, for which it has editing rights.

917 The handler behaviour selects a suitable (...) pair of edit/undo functions and a suitable (annotation) note.

918 It then communicates the desire to edit document di with (e,u) (at time $t = \text{time_ch}?$).

919 Editing take some time, ti.

920 We can therefore assert that the time at which editing has completed is $t + ti$.

921 The handler behaviour accepts the edit completion message from the document handler.

922 The handler behaviour can therefore resume with an updated document history.

912b $\text{hdlr_edit}_{hi}: \text{HATTRS} \times \text{HHIST} \rightarrow \mathbf{in, out} \{\text{hdlr_docu_ch}[hi, di] | di:DI \cdot di \in \text{dis}\} \mathbf{Unit}$ 912b $\text{hdlr_edit}_{hi}(\text{hattr}, \text{hhist}) \equiv$ 916 [1] **let** $di:DI \cdot di \in \mathbf{dom} \text{ hattr} \wedge \text{"edit"} \in \text{hattr}(di) \mathbf{in}$ 917 [1] **let** $(e, u):(\text{EDIT} \times \text{UNDO}) \cdot \dots, n:AN \cdot \dots \mathbf{in}$ 918 [1] $\text{hdlr_docu_ch}[hi, di] ! \text{mkEdit}(hi, t = \text{time_ch}?, e, u, n) ;$ 919 [2] **let** $ti: \text{TIME_INTERVAL} \cdot \dots \mathbf{in}$ 920 [2] **wait** ti ; **assert:** $\text{time_ch} = t + ti$ 921 [3] **let** $\text{mkEditComplete}(ti', \dots) = \text{hdlr_docu_ch}[hi, di] ? \mathbf{in} \mathbf{assert} \text{ } ti' \cong ti$ 922 $\text{hdlr}_{hi}(\text{hattr}, \text{augment}(\text{hhist}, (di, \text{mkEdit}(hi, t, ti, e, u))))$ 912b **end end end end**

The Handler Read Behaviour: Left Fig. D.4 on Page 334

- 923 The **handler behaviour**, on its own volition, decides to read a document, di , for which it has reading rights.
- 924 It then communicates the desire to read document di with at time $t=time_ch?$ – with an annotation note (n).
- 925 Reading take some time, ti .
- 926 We can therefore assert that the time at which reading has completed is $t+ti$.
- 927 The handler behaviour accepts the read completion message from the document handler.
- 928 The handler behaviour can therefore resume with an updated document history.

```

912c  hdlr_edithi: HATTRS × HHIST → in,out {hdlr_docu_ch[hi,di]|di:DI•di∈dis} Unit
912c  hdlr_edithi(hattrs,hhist) ≡
923  [1] let di:DI • di ∈ dom hattrs ∧ "read" ∈ hattrs(di), n:N • ... in
924  [1] hdlr_docu_ch[hi,di] ! mkRead(hi,t=time_ch?,n) ;
925  [2] let ti:TIME_INTERVAL • ... in
926  [2] wait ti ; assert: time_ch? = t+ti
927  [3] let mkReadComplete(ti,...) = hdlr_docu_ch[hi,di] ? in
928      hdlrhi(hattrs,augment(hhist,(di,mkRead(di,t,ti))))
912c  end end end

```

The Handler Copy Behaviour: Right Fig. D.4 on Page 334

- 929 The **handler [copy] behaviour**, on its own volition, decides to copy a document, di , for which it has copying rights.
- 930 It communicates this copy request to the management behaviour.
- 931 After a while the handler [copy] behaviour receives acknowledgement of a completed copying from the management behaviour.
- 932 The handler [copy] behaviour records the request and acknowledgement in its, thus updated whereupon the handler [copy] behaviour resumes being the handler behaviour.

```

912d  hdlr_copyhi: HATTRS × HHIST → in,out mgmt_hdlr_ch[hi] Unit
912d  hdlr_copyhi(hattrs,hhist) ≡
929  [1] let di:DI • di ∈ dom hattrs ∧ "copy" ∈ hattrs(di) in
930  [1] mgmt_hdlr_ch[hi] ! mkCopy(di,hi,t=time_ch?) ;
931  [10] let mkCopyComplete(di',di) = mgmt_hdlr_ch[hi] ? in
932  [10] hdlrhi(hattrs,augment(hhist,time_ch?,(mkCopy(di,hi,,t),mkCopyComplete(di'))))
912d  end end

```

The Handler Grant Behaviour: Left Fig. D.5 on Page 334

- 933 The **handler [grant] behaviour** offers to accept grant permissions from the management behaviour.
- 934 In response it updates its handler attribute while resuming being a handler behaviour.

```

912e  hdlr_granthi: HATTRS × HHIST → in,out mgmt_hdlr_ch[hi] Unit
912e  hdlr_granthi(hattrs,hhist) ≡
933  [2] let mkGrant(di,acrs) = mgmt_hdlr_ch[hi] ? in
934  [2] hdlrhi(hattrs†[di→acrs],augment(hhist,time_ch?,mkGrant(di,acrs)))
912e  end

```

D.16.4 Document Behaviours

935 The document behaviour is involved in the following action traces:

- a **edit**
- b **read**
- c **shred**

Fig. D.3 on Page 333 Right

Fig. D.4 on Page 334 Left

Fig. D.5 on Page 334 Right

value

```

935 docudi: DD × (DA × DC × DH) → in,out arch_docu_ch[di], {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
935 docudi(dattrs) ≡
935a   docu_editdi(dd)(da,dc,dh)
935b   [] docu_readdi(dd)(da,dc,dh)
935c   [] docu_shreddi(dd)(da,dc,dh)

```

The Document Edit Behaviour: Right Fig. D.3 on Page 333

936 The **document [edit] behaviour** offers to accept edit requests from document handlers.

- a The document contents is edited, over a time interval of ti , with respect to the handlers edit function (e),
- b the document annotations are augmented with respect to the handlers note (n), and
- c the document history is augmented with the fact that an edit took place, at a certain time, with a pair of *edit/undo* functions.

937 The *edit* (etc.) function(s) take some time, ti , to do.

938 The handler behaviour is notified, mkEditComplete(...) of the completion of the edit, and

939 the document behaviour is then resumed with updated programmable attributes.

value

```

935a docu_editdi: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
935a docu_editdi(dd)(da,dc,dh) ≡
936 [2] let mkEdit(hi,t,e,u,n) = [] {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
936a [2] let dc' = e(dc),
936b   da' = augment(da,((hi,t),("edit",e,u),n)),
936c   dh' = augment(dh,((hi,t),("edit",e,u))) in
937   let ti = time_ch? - t in
938   hdlr_docu_ch[hi,di] ! mkEditComplete(ti,...) ;
939   docudi(dd)(da',dc',dh')
935a end end end

```

The Document Read Behaviour: Left Fig. D.4 on Page 334

940 The **document [read] behaviour** offers to receive a read request from a handler behaviour.

941 The reading takes some time to do.

942 The handler behaviour is advised on completion.

943 And the document behaviour is resumed with appropriate programmable attributes being updated.

value

```

935b docu_readdi: DD × (DA × DC × DH) → in,out {hdlr_docu_ch[hi,di]|hi:HI•hi∈his} Unit
935b docu_readdi(dd)(da,dc,dh) ≡
940 [2] let mkRead(hi,t,n) = {hdlr_docu_ch[hi,di]?|hi:HI•hi∈his} in
941 [2] let ti:TIME_INTERVAL • ... in

```

```

941 [2] wait ti ;
942 [2] hdlr_docu_ch[hi,di] ! mkReadComplete(ti,...) ;
943 [2] docudi(dd)(augment(da,n),dc,augment(dh,(hi,t,ti,"read")))
935b     end end

```

The Document Shred Behaviour: Right Fig. D.5 on Page 334

944 The **document [shred] behaviour** offers to accept a document shred request from the archive behaviour –
 945 whereupon it **stops** !

value

```

935c docu_shreddi: DD × (DA × DC × DH) → in,out arch_docu_ch[di] Unit
935c docu_shreddi(dd)(da,dc,dh) ≡
944 [3] let mkShred(...) = arch_docu_ch[di] ? in
945     stop
935c [3] end

```

D.16.5 Conclusion

This completes a first draft version of this document. The date time is: November 24, 2019: 13:16. Many things need to be done. First a careful checking of all types and functions: that all used names have been defined. The internal non-deterministic choices in formula Items 860 on Page 335, 896 on Page 338, 912 on Page 340 and 935 on the facing page, need be checked. I suspect there should, instead, be some mix of both internal and external non-deterministic choices. Then a careful motivation for all the other non-deterministic choices.

D.17 Documents in Public Gornment

Public government, in the spirit of *Charles-Louis de Secondat, Baron de La Brède et de Montesquieu* (or just *Montesquieu*), has three branches:

- the **legislative**,
- the **executive**, and
- the **judicial**.

Our interpretation of these, with respect to documents, are as follows.

- The **legislative** branch produces laws, i.e., documents. To do so many preparatory documents are created, edited, read, copied, etc. Committees, subcommittees, individual lawmakers and ministry law office staff handles these documents. Parliament staff and legislators are granted limited or unlimited access rights to these documents. Finally laws are put into effect, are amended, changed or abolished. The legislative branch documents refer to legislative, executive and judicial branch documents.
- The **executive** branch produces guide lines, i.e., documents. Instructions on interpretation and implementation of laws; directives to ministry services on how to handle the laws; etcetera. These executive branch documents refer to legislative, executive and judicial branch documents.
- The **judicial** branch produces documents. Police cite citizens and enterprises for breach of law. Citizens and enterprise sue other citizens and/or enterprises. Attorneys on behalf of the governments, or citizens or enterprises prepare statements. Court proceedings are recorded. Justices pass verdicts. The judicial branch documents refer to legislative, executive and judicial branch documents.

D.18 Documents in Urban Planning

A separate research note [95, Urban Planning Processes] analyses & describes a domain of urban planning. There are the geographical documents:

- geodetic,
- geotechnic,
- meteorological,
- and other types of geographical documents.

In order to perform an informed urban planning further documents are needed:

- auxiliary documents which
- requirements documents which

Auxiliary documents presents such information that “fill in” details concerning current ownership of the land area, current laws affecting this ownership, the use of the land, etcetera. Requirements documents express expectations about the (base) urban plans that should result from the base urban planning. As a first result of base urban planning we see the emergence of the following kinds of documents:

- base urban plans
- and ancillary notes.

The base urban plans deal with

- cadastral,
- cartographic and
- zoning

issues. The ancillary notes deal with such things as insufficiencies in the base plans, things that ought be improved in a next iteration base urban planning, etc. The base plans and ancillary notes, besides possible re-iteration of base urban planning, lead on to “derived urban planning” for

- light, medium and heavy industry zones,
- mixed shopping and residential zones,
- apartment building zones,
- villa zones,
- recreational zones,
- etcetera.

After these “first generation” derived urban plans are well underway, a “second generation” derived urban planning can start:

- transport infrastructure,
- water and waste resource management,
- electricity, natural gas, etc., infrastructure,
- etcetera.

And so forth. Literally “zillions upon zillions” of strongly and crucially interrelated documents accrue.

Urban planning evolves and revolves around documents.

Documents are the only “tangible” results of urban planning.¹³

¹³ Once urban plans have been agreed upon by all relevant authorities and individuals, then urban development (“build”) and, finally, “operation” of the developed, new urban “landscape”. For development, the urban plans form one of the “tangible” inputs. Others are of financial and human and other resource nature.

E

Urban Planning

We examine concepts of urban planning. There is *the urban space* which we treat as a *part* and as a *behaviour*. There are n distinct urban space *analysers*, distinctly named (i.e., indexed) $\{a_1, a_2, \dots, a_n\}$, treated as [parts and] *behaviours*. There is one *master planner*, treated as a [part and as a] *behaviour*. There are p distinctly named *derived [urban] planners*, distinctly named (i.e., indexed) $\{d_1, d_2, \dots, d_p\}$ and treated as [parts and] *behaviours*. .

To serve the one master and the p derived planners there are $1+p$ distinctly named *input argument servers* $\{m, d_1, d_2, \dots, d_p\}$, one *output result server*, and one *derived planner index generator*. All of these are also treated as *parts* and *behaviours*. .

The behaviours (synchronise and) communicate via *channels*. An array of channels communicate *urban space attribute values* to requesting analysers. The analysers provide *analyses* to all planners. The planners obtain *input arguments* from “their” servers. The planners provide *result values* to the common output result server. And the derived planner index generator provide possibly empty sets of *derived planner indices* to all planners. .

Emphasis, in this research note, is on the *information* (abstract “data”) and *functions* and *behaviours* of urban space analysis and planning – and their *interaction*. We separate *urban space analysis* from *urban planning*. *Urban space analysers* analyse [existing] urban spaces and produce *analyses*. *Urban planners* analyse the analysis results and, in case of the master planning, also the urban space [itself] – and produce plans and other information. The *master [urban] planner* produces a master plan [and other information]. The *derived [urban] planners* produce derivative [urban] plans [and other information]. That is, we thus distinguish between the two kinds of urban planning: the *master*, ‘ab initio’, behaviour of determining “the general layout of the land (!)”, and the *derived*, ‘follow-up’, behaviours focused on social and technological infrastructures. Master urban planning applies to descriptions of “the land”: *geographic, that is, geodetic, cadastral, geotechnical, meteorological, socio-economic and rules & regulations*. Examples of derived urban plannings are such which are focused on humans and on social and technological artifacts: *industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas, etc., and health care, schools, transport, electricity, water, waste, etc.* This research note also discusses issues of urban planning project management, cf. Sect. E.16.4, and urban planning document management, cf. Sect. E.16.2. The overall aim of this paper is to suggest a formal foundation for urban planning. We must emphasize that all that is conceivable and describable in the domain can be described. We shall return to this remark, in this report, again-and-again.

E.1 Introduction

“Urban planning is a technical and political process concerned with the development and use of land, planning permission, protection and use of the environment, public welfare, and the design of the urban

environment, including air, water, and the infrastructure passing into and out of urban areas, such as transportation, communications, and distribution networks.”¹

In this research note we shall try to understand two of the aspects of the domain underlying urban planning, (i) namely those of the “input” information to and “output” plans (etc.) from urban planning, and (ii) that of some possible urban planning (development) functions and processes. We are trying to understand and describe a domain, not requirements for IT for that domain and certainly not the IT (incl. its software). And: We are certainly not constructing any general or any specific urban plan!

The overall aim of this case study is to suggest a formal foundation for urban planning.

Another, secondary aim of this case study is to suggest that a number of requirements must be satisfied before a fully professional urban development project can be commenced.

E.1.1 On Urban Planning

We search for answers to the question: “What is Urban Planning?”. First we identify “planning areas”. Then we sketch element of a first domain model for Urban Planning.

Urban planning seems to be also be about *infrastructure planning*. So we examine these terms. First the latter, then the former.

Infrastructures

The term ‘infrastructure’ has gained currency in the last 80 years.² It is more frequently used in socio-economic than in scientific, let alone computing science, contexts. According to the World Bank, ‘infrastructure’ is an umbrella term for many activities referred to as ‘social overhead capital’ by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users). We take a more technical view, and see infrastructures as concerned with supporting other systems or activities. Software for infrastructures is likely to be distributed and concerned in particular with supporting communication of data, people and/or materials. Hence issues of openness, timeliness, security, lack of corruption and resilience are often important.

Examples of infrastructures, or, more precisely, infrastructure components, are:

- transport systems (roads, railways, air traffic, canals/rivers/lake/ocean , etc.);
- water and sewage;
- telecommunications;
- postal service (physical letters, packages etc.);
- power: electricity, gas, oil, wind (generation, distribution); etc.
- the financial industry (banking, insurance, securities, clearing, etc.);
- documents (creation, editing, formatting, etc.);
- ministry of finance (taxation, budget, treasury, etc.);
- health care (private physicians, clinics, hospitals, pharmacies, etc.);
- education (kindergartens, pre-schools, primary schools, secondary schools, high schools, colleges, universities);
- manufacturing industry;
- et cetera.

Wikipedia: https://en.wikipedia.org/wiki/Urban_planning

“*Urban planning is a technical and political process concerned with the development and use of land planning permission, protection and use of the environment, public well-fare, and the design of the urban*

¹ https://en.wikipedia.org/wiki/Urban_planning

² Winston Churchill is quoted to have said, in the House of Commons, in 1936: . . . *the young Labourite speaker, that we just heard, obviously wishes to impress his constituency with the fact that he has attended Eton and Oxford when he uses such modern terms as ‘infrastructure’ . . .*

environment, including **air, water, and the infrastructure** passing into and out of urban areas, such as **transportation, communications, and distribution networks** [947].”

“Urban planning is also referred to as urban and regional planning, regional planning, town planning, city planning, rural planning or some combination in various areas worldwide. It takes many forms and it can share perspectives and practices with urban design [946].”

“Urban planning guides orderly development in urban, suburban and rural areas. Although predominantly concerned with the planning of settlements and communities, urban planning is also responsible for the **planning and development of water use and resources, rural and agricultural land, parks and conserving areas of natural environmental** significance. Practitioners of urban planning are concerned with research and analysis, strategic thinking, architecture, urban design, public consultation, policy recommendations, implementation and management [948].”

“Urban planners work with the cognate fields of architecture, landscape architecture, civil engineering, and public administration to achieve strategic, policy and sustainability goals. Early urban planners were often members of these cognate fields. Today urban planning is a separate, independent professional discipline. The discipline is the broader category that includes different sub-fields such as land-use planning, zoning, economic development, environmental planning, and transportation planning [949].”

Theories of Urban Planning

“Planning theory is the body of scientific concepts, definitions, behavioral relationships, and assumptions that define the body of knowledge of urban planning. There are eight procedural theories of planning that remain the principal theories of planning procedure today: the rational-comprehensive approach, the incremental approach, the trans-active approach, the communicative approach, the advocacy approach, the equity approach, the radical approach, and the humanist or phenomenological approach [950].”

Technical aspects

Technical aspects of urban planning involve applying scientific, technical processes, considerations and features that are involved in planning for land use, urban design, natural resources, transportation, and infrastructure. Urban planning includes techniques such as: predicting population growth, zoning, geographic mapping and analysis, analyzing park space, surveying the water supply, identifying transportation patterns, recognizing food supply demands, allocating health-care and social services, and analyzing the impact of land use.

Urban planners

An urban planner is a professional who works in the field of urban planning for the purpose of optimizing the effectiveness of a community’s land use and infrastructure. They formulate plans for the development and management of urban and suburban areas, typically analyzing land use compatibility as well as economic, environmental and social trends. In developing the plan for a community (whether commercial, residential, agricultural, natural or recreational), urban planners must also consider a wide array of issues such as sustainability, air pollution, traffic congestion, crime, land values, legislation and zoning codes.

The importance of the urban planner is increasing throughout the 21st century, as modern society begins to face issues of increased population growth, climate change and unsustainable development. An urban planner could be considered a green collar professional.[clarification needed]

References

946. “What is Urban Planning” (retrieved April 24, 2015)
<https://mcgill.ca/urbanplanning/planning>

“Modern urban planning emerged as a profession in the early decades of the 20th century, largely as a response to the appalling **sanitary, social, and economic** conditions of rapidly-growing industrial cities. Initially the disciplines of architecture and civil engineering provided the nucleus of concerned professionals. They were joined by **public health** specialists, economists, sociologists, lawyers, and geographers, as the complexities of managing cities came to be more fully understood. Contemporary urban and regional planning techniques for survey, analysis, design, and implementation developed from an interdisciplinary synthesis of these fields. Today, urban planning can be described as a technical and political process concerned with the welfare of people, control of the use of land, design of the urban environment including transportation and communication networks, and protection and enhancement of the natural environment.”

947. Van Assche, K., Beunen, R., Duineveld, M., & de Jong, H. (2013). Co-evolutions of planning and design: Risks and benefits of design perspectives in planning systems. *Planning Theory*, 12(2), 177-198.
948. Taylor, Nigel (2007). *Urban Planning Theory since 1945*, London, Sage.
949. <https://www.planning.org/aboutplanning/whatisplanning.htm>: “What Is Planning?”. www.planning.org. Retrieved 2015-09-28.
950. <https://www.planetizen.com/node/73570/how-planners-use-planning-theory>: How Planners Use Planning Theory. Andrew Whitmore of the University of North Carolina Department of Urban and Regional Planning identifies planning theory in everyday practice.

E.1.2 On the Form of This Research Note

The present form of this research note, as of November 24, 2019: 13:16, is that of recording a development. The development is that of *trying to come to grips with what urban planning is*. We have made the decision, from an early start, that urban planning “as a whole” is a collection of one master and an evolving number of (initially zero) derived urban planning behaviours. Here we have made the choice to model the various behaviours of a complex of urban planning functions.

E.1.3 On the Structure of this Research Note

The page references in the items below refer to the first page of the part, section or subsections listed.

- It is always a good idea to study the contents listing. The author have made some effort in structuring the presentation. And the result of this effort is obviously reflected in the contents listing.
- Section E.3 [Page 351] can be skipped in any reading by those familiar with *tritych* approach to software development, *formal methods*, my work on *domain science & engineering*, etc. – topics that are otherwise covered in Sect. E.4. Sect. E.5 reviews the changes of my *domain analysis & description calculus*, wrt. [70]. These changes take effect in our treatments of parts E.6 and E.11.

The next two parts are concerned with the [research &] development of a **model** of urban analysis and planning.

- Section E.6 [Page 355] treats the endurants of urban analysis and planning. It unfolds the model in four stages:
 - ⊗ Sect. E.7 [Page 356] analyses & describes the universe of discourse, the structures and the (atomic) parts;
 - ⊗ Sect. E.8 [Page 359] analyses & describes the unique identifiers of all atomic parts;
 - ⊗ Sect. E.9 [Page 364] analyses & describes the mereologies of all atomic parts;
 - ⊗ Sect. E.10 [Page 367] analyses & describes the attributes of all atomic parts.
- Section E.11 treats the perdurants of urban analysis and planning. It further unfolds the model in four stages:
 - ⊗ Sect. E.12 [Page 377] calculates behaviours from parts;

- ⊗ Sect. E.13 [Page 379] analyses & describes channels by means of which the behaviours can synchronise & communicate;
- ⊗ Sect. E.14 [Page 383] calculate the basics of all atomic behaviours and define these behaviours; and
- ⊗ Sect. E.15 [Page 396] finally suggests an initial composition of the atomic behaviours.
- Section E.16 collects a number of “loose” ends:
 - ⊗ Subsect. E.16.1 [Page 398] laments over the lack of assertions related to liveness and deadlock freeness of the defined behaviours and their initialisation;
 - ⊗ Subsect. E.16.2 [Page 399] points out that documents, their distribution and sharing, play a central rôle in urban analysis and planning;
 - ⊗ Subsect. E.16.3 [Page 399] muses over issues of validation and verification of the proposed model of urban analysis and planning; and
 - ⊗ Subsect. E.16.4 [Page 400] points out that the model of urban analysis and planning implies a number of issues with respect to the organisation and management of urban analysis and planning projects.

E.2 An Urban Planning System

E.2.1 A First Iteration Overview

We think of urban planning to be “dividable” into master urban planning, `master_planner`, and derived urban plannings, `derived_planneri`, where sub-index i indicate that there may be several, i.e., $i \in \{d_1, d_2, \dots, d_n\}$, such derived urban plannings.

We think of master urban planning to “convert” physical (geographic, that is, geodetic, cadastral, geo-technical, meteorological, etc.) information about the land area to be developed into a master plan, that is, cartographic, cadastral and other such information (zoning, etc.). And we think of derived urban planning to “convert” master plans into societal and/or technological plans. Societal and technological urban planning concerns are typical such as *industry zones*, *commercial (i.e., office and shopping) zones*, *residential zones*, *recreational areas*, etc., and *health care*, *schools*, *transport*, *electricity*, *water*, *waste*, etc.

Each urban planning *behaviour*, whether ‘master’ or ‘derived’, is seen as a *sequence* of the applications of “the same” urban planning *function*, – but possibly to different goals so that each application (of “the same” urban planning *action*) resolves a sub-goal. Each urban planning action takes a number of information *arguments* and yield information *results*. The master urban planning behaviour may **start** one or more derived urban planning behaviours, `derived_planneri`, at the end of “completion” of a master urban planning *action*. Let $\{d_1, d_2, \dots, d_n\}$ index separate derived urban plannings, each concerned with a distinct, i.e., reasonably delineated technological and/or societal urban planning concern. During master urban planning actions may start any of these derived urban plannings once.

Thus we think of urban planning as a system of a single master urban planning process (i.e., behaviour), `master_planner`, which “spawns” zero, one or more (but a definite number of) derived urban planning processes (i.e., behaviours), `derived_plannerj`. Derived urban planning processes, `derived_plannerj`, may themselves start other derived urban planning processes, `derived_planneri`, `derived_plannerk`, ..., `derived_plannerl`.

Figure E.1 is intended to illustrate the following: At time t_0 a master urban planning is started. At time t_1 the master urban planning initiates a number of derived urban development, D_1, \dots, D_i . At time t_2 the master urban planning initiates the D_j derived urban planning. At time t_3 the derived urban planning D_i initiates two derived urban plannings, D_k and D_l . At time t_4 the master urban planning ends. And at time t_5 all urban plannings have ended. .

Urban planning actions are provided with “input” in the form of either geographic, geodetic, geo-technical, meteorological, etc., information, `tusm:TUSm`, or auxiliary information, `m_aux:mAUX`³, or re-

³ The `m_` value prefixes and the `m` type prefixes shall designate master urban planning entities.

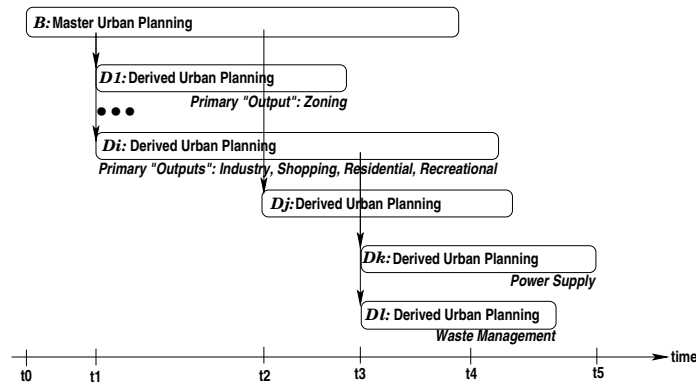


Fig. E.1. An Urban Planning Development

quirements information, $m_req:mREQ$. We shall detail issues of the urban space, auxiliary and requirements information later.

E.2.2 A Visual Rendition of Urban Planning Development

We examine concepts of urban analysis and planning. We refer to Fig. E.2 [Page 351]. There is *the urban space*: $tus:TUS$, which we treat as a *part* and as a *behaviour*. There are a distinct urban space analysers, distinctly named (i.e., indexed) $\{a_1, a_2, \dots, a_a\}$, treated as [parts and] *behaviours*. There is one *master planner*, treated as a [part and as a] *behaviour*. There are p distinctly named *derived [urban] planners*, distinctly named (i.e., indexed) $\{d_1, d_2, \dots, d_p\}$ and treated as [parts and] *behaviours*.

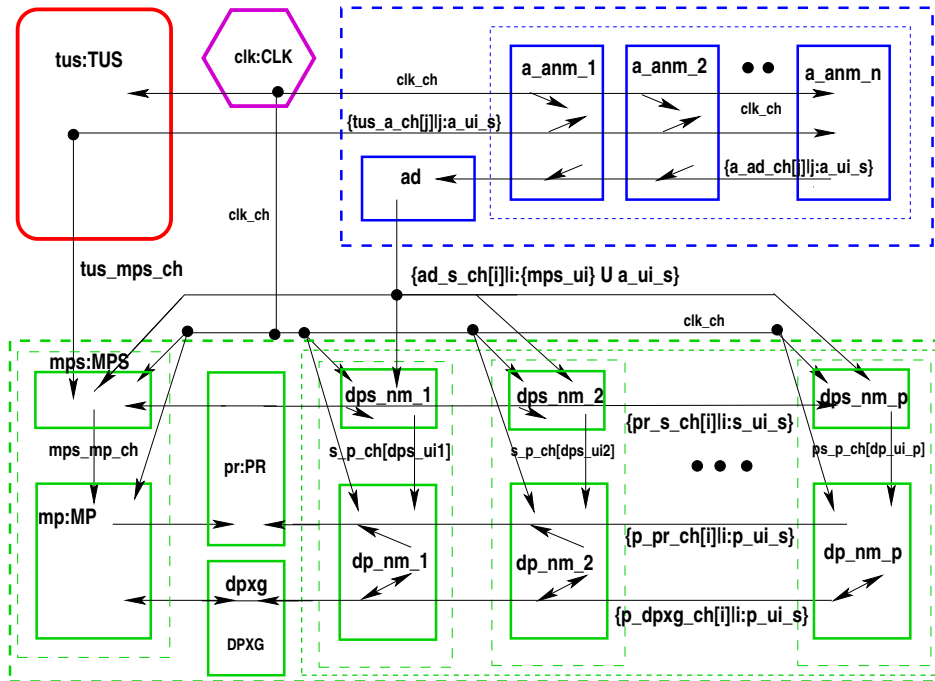


Fig. E.2. An Urban Analysis and Planning System

The behaviours (synchronise and) communicate via *channels*. An array of channels communicate *urban space attribute values* to requesting analysers. The analysers provide *analyses* to all planners. The planners obtain *input arguments* from “their” servers. The planners provide *result values* to the common output result server. And the derived planner index generator provide possibly empty sets of *derived planner indices* to all planners.

Emphasis, in this research note, is on the *information* (abstract “data”) and *functions* and *behaviours* of urban space analysis and planning – and their *interaction*. We separate *urban space analysis* from *urban planning*. *Urban space analysers* analyse [existing] urban spaces and produce *analyses*. *Urban planners* analyse the analysis results and, in case of the master planning, also the urban space [itself] – and produce plans and other information. The *master [urban] planner* produces a master plan [and other information]. The *derived [urban] planners* produce derivative [urban] plans [and other information].

That is, we thus distinguish between the two kinds of urban planning: the *master*, ‘ab initio’, behaviour of determining “the general layout of the land (!)”, and the *derived*, ‘follow-up’, behaviours focused on social and technological infrastructures. Master urban planning applies to descriptions of “the land”: *geographic, that is, geodetic, cadastral, geotechnical, meteorological, socio-economic and rules & regulations*. Examples of derived urban plannings are such which are focused on humans and on social and technological artifacts: *industry zones, commercial (i.e., office and shopping) zones, residential zones, recreational areas, etc., and health care, schools, transport, electricity, water, waste, etc.*

E.3 METHOD

Several factors necessitated this part of this case study.

- In Sect. E.4, [“Prelude”] we briefly present basic issues of formal development.
- In Sect. E.5 [“Review & Refinement of the Method”] we then
 - ∞ review, in Sect. E.5.1 [“Review of Manifest Domains: Analysis & Description”] the specific approach basically taken when we describe manifest domains [70] and,
 - ∞ as a result of a number of recent (2016–2017) *experimental research & engineering* work, [63, 67, 95, 72, 75, 69],
 - ∞ we refine the approach described in [70], Sect. E.5.2 [“Refinement of the Method”].

E.4 Prelude

E.4.1 A Triptych of Software Development

Before hardware and software systems can be designed and coded we must have a reasonable grasp of “its” requirements; before requirements can be prescribed we must have a reasonable grasp of “the underlying” domain. To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

By a domain description we understand a collection of pairs of narrative and of commensurate formal texts, where each pair describes either aspects of an enduring entity (i.e., information) or aspects of a perdurant entity (i.e., an action, event or behaviour).

E.4.2 On Formality

We consider software programs to be formal, i.e., mathematical, quantities — rather than of social/psychological interest. We wish to be able to reason about software, whether programs, or program specifications, or requirements prescriptions, or domain descriptions. Although we shall only try to understand some facets

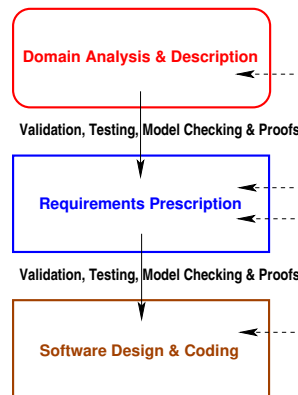


Fig. E.3. The Triptych of Software Development

of the domain of urban planning we shall eventually let such an understanding, in the form of a precise, formal, mathematical, although non-deterministic, i.e., “multiple choice”, description be the basis for subsequent requirements prescriptions for software support, and, again, eventually, “the real software itself”, that is, tools, for urban planners. We do so, so that we can argue, eventually prove formally, that the software *is correct* with respect to the (i.e., its) formally prescribed requirements, and that the software *meets customer*, i.e., domain users’ *expectations* – as expressed in the formal domain description.

E.4.3 On Describing Domains

If we can describe some domain phenomenon in logical statements and if these can be transcribed into some form of mathematical logic and set theory then we may have to describe it: narratively and formally. That is, even though it may be humanly or even technologically very cumbersome or even impossible to implement what is described we may find it necessary to describe it. *As to when we have to describe something – that is another matter!*⁴ Let us give an example: The example is that of the domain of *documents*. Documents may be *created, edited, read, copied, referred to, and shredded*. We may talk, meaningfully, that is, rationally, logically, about the previous *version* of a document, and hence we may be obliged to model document versions as from their first creation, *who created, who edited, who read, who copied, and who shredded* (sic!) a document, including, perhaps, the *location* and *time* of these operations, *how they were edited*, etc., etc. Let us take another example. As for the meteorological properties of any specific geographic area, these properties, like temperature, humidity, wind, etc., vary, in reality, continuously over time, from location to location, including altitude. In modeling meteorological properties we may be well-served when modeling exactly their continuous, however “sporadic” nature. To a first approximation we do not have to bother as to whether we can actually “implement” the recording of such continuous, “sporadic” “behaviours”. In that sense the domain analyser cum describer is expected to be like the physicists,⁵ certainly not like programmers. That is: *the domain analyser cum describer are not necessarily describing computable domains*.

⁴ We may find occasions in this document to discuss this “other matter”!

⁵ It is written above: that domain descriptions are based on mathematical logic and set theory. Yes, unfortunately! To properly describe domains involving continuity we need “mix” logic with classical calculus: differential equations, integrals, etc. And here we have nothing to say: the ability, in an informed ways, to blend mathematical logic and set theoretic descriptions with differential equations, integrals, etc., is almost non-existent as of 2017/2018!

E.4.4 Reiterating Domain Modeling

Any domain description is an approximation. One cannot ever hope to have described all facets of any domain. So, in setting out to analyse & describe a domain one is not trying to produce a definitive, final, model; one is merely studying and recording (some) results of that study. One is prepared to reiterate the study and produce alternative models. From such models one can develop requirements, [66], for software that in one way or another support activities of the domain. If you are to seriously develop software in this way, for example for the support of urban planners, then you must be prepared to “restart” the process, to develop, from scratch, a domain model. You have a basis from which to start, namely this report [95]. But do not try to simply modify it. Study [95] in depth, but rethink that basis. A description, any description, can be improved. Perhaps the emphasis should be refocused. For the example of software (incl. IT) support for the keeping, production, editing, etc., of the very many documents that are needed during urban planning, you may, in addition to refocusing the present report’s focus on the documents of the very many document categories that are presumed, introduced and further elaborated upon in the present report, also study [72]. A principle guiding us in the reformulation of a domain model to be the basis for a specific software product is that we must strive to document all the assumptions about the context in which this software is to serve – otherwise we cannot hope to achieve a product that meets its customers expectations.

E.4.5 Partial, Precise, and Approximate Descriptions

By a *partial description* we mean a description which covers only a fraction of the domain as a group of people working in that domain, that is, professionals, would otherwise talk about. Descriptions are here taken to describe *behaviours*: first “do this”, then “do that”! By a *precise description* we mean a description which in whatever behaviour it describes, partially or fully, does so precisely, that is, it is precisely as described, no more, no less. By an *approximate description* we mean a description which in whatever behavior it describes, partially or fully, even when precisely so, allows for a set of interpretations.

We shall then avail ourselves of two forms of ‘approximation’: *internal non-determinism* and *external non-determinism*. By *internal non-deterministic behaviour* we shall mean a behaviour whose “next step, next move” is “determined” by some “own flipping a coin”. By *external non-deterministic behaviour* we shall mean a behaviour whose “next step, next move” is “determined” by some “outside demon”! In describing urban planning we shall allow for: partial descriptions: not all is described and what has been selected for description has been so, perhaps rather arbitrarily, by us, i.e., me, and both forms of ‘approximation’. We shall endeavour to indicate where and why we present only partial descriptions, and deploy ‘approximation’.

E.4.6 On Formal Notations

To be able to *prove formal correctness* and *meeting customer expectations* we avail ourselves of some formal notation. In this research note we use the RAISE [132] Specification Language, RSL, [131]. Other formal notations, such as Alloy [156], Event B [1], VDM-SL citevdm or Z [260] could be used. We choose RSL since it, to our taste, nicely embodies Hoare’s concept of *Communicating Sequential Processes*, CSP [148]

E.5 Review & Refinement of the Method

The basis for the kind of domain analysis & description of this case study is [70]. It was submitted 19 Dec. 2014 and (paper) published in March 2017. Between those dates and in particular since March 2017 a number of *experimental engineering cum research* took place. We mention some of these. A *credit card system* modeling, [63, May 2016]. A *weather forecast system* modeling, [67, Nov. 2017]. The first phase, March 2017–July 2017, of this *urban planning* project [95]. A *document system*, [72, July 2017]. A *clarification* of concepts so-called *implicit/explicit semantics*, [75, Oct. 2017]. A *swarms of drones* modeling experiment, [69, Nov.–Dec.].

E.5.1 Review of Manifest Domains: Analysis & Description

We refer to [70, submitted 19 Dec. 2014, published March 2017] We present a terse, itemised summary of the method outlined in that paper:

- First we analyse & describe *endurants*:
 - ⊗ the *form* of *parts*, *components* and *materials*.
 - ⊗ then the *qualities* of *parts*, *components* and *materials*, that is:
 - ⊗ the *unique identifiers* of *parts* and *components*, then
 - ⊗ the *mereology* of *parts*, and finally
 - ⊗ the *attributes* of *parts* and *materials*.
- Then we analyse & describe *perdurants*:
 - ⊗ the notion of *domain states*,
 - ⊗ the *actions*, then
 - ⊗ the *events*, and finally
 - ⊗ the *behaviours*.
- As part of the description of behaviours we analyse & describe
 - ⊗ *channels*

We can summarise this in the *ontology diagram*, cf. Fig. E.4 [Page 354] of [70].

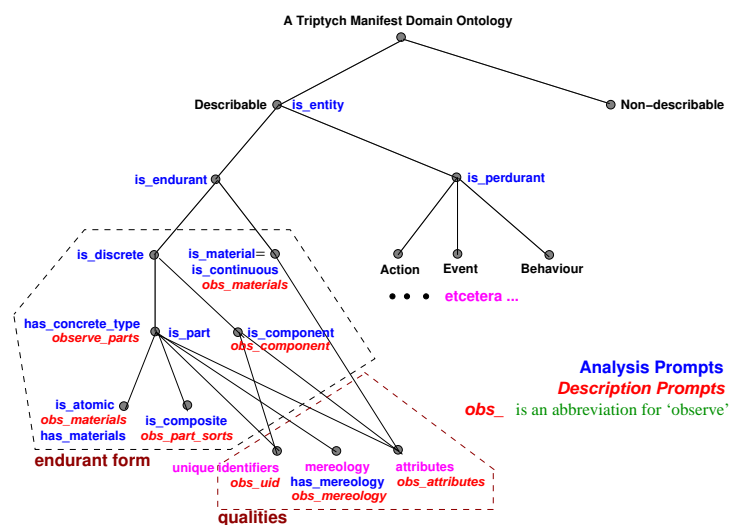


Fig. E.4. The Previous Upper Ontology

E.5.2 Refinement of the Method

- First we analyse & describe *endurants*:
 - ⊗ the *form* of **structures**, *parts*, *components* and *materials*. The refinement of the *manifest domain analysis & description* approach is the addition of *endurant structures*. Structures are “abstract composite parts”, though with no *qualities*,
 - ⊗ then we analyse & describe *qualities* of *parts*, *components* and *materials*, that is:
 - ⊗ the *unique identifiers* of *parts* and *components*, then
 - ⊗ the *mereology* of *parts*, and finally

- ∞ the *attributes* of *parts* and *materials*.
- Then we analyse & describe *perdurants*:
 - ∞ the notion of *domain states* and
 - ∞ the **channels**. We observe that this item has been “moved” to “before” analysis & description of subsequent analyses & descriptions.
 - ∞ The *behaviours*.
 - ∞ As part of the description of behaviours we analyse & describe
 - ∞ the *actions* and
 - ∞ the *events*

We can summarise this in a revised *ontology diagram*, cf. Fig. E.5 [Page 355].

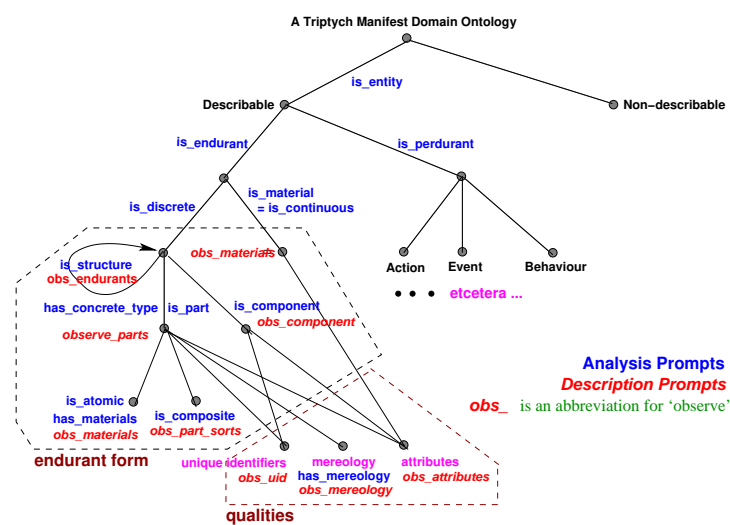


Fig. E.5. The Refined Upper Ontology

E.6 ENDURANTS

By an *entity* we shall understand a phenomenon, i.e., something that can be observed, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity. We further demand that an entity can be objectively described

By an *endurant* we shall understand an entity that can be observed or conceived and described as a “complete thing” at no matter which given snapshot of time. Were we to “freeze” time we would still be able to observe the entire endurant.

By a *discrete endurant* we shall understand an endurant which is separate, individual or distinct in form or concept.

By a *part* we shall understand a discrete endurant which the domain engineer chooses to endow with *internal qualities*⁶ such as *unique identification*, *mereology*, and one or more *attributes*. We shall define these three categories in Sects. E.8, E.9, respectively Sect. E.10. We refer in general to [70].

In this, a major section of this case study, we shall cover

⁶ – where by *external qualities* of an endurant we mean whether it is discrete or *continuous*, whether it is a *part*, or a *component* – such as these are defined in [70].

- Sect. E.7: Parts,
- Sect. E.8: Unique Identifiers,
- Sect. E.9: Mereology, and
- Sect. E.10: Attributes.

E.7 Structures and Parts

From an epistemological⁷ point of view a study of the parts of a universe of discourse is often the way to understand “*who the players*” of that domain are. From the point of view of [70] *knowledge about parts lead to knowledge about behaviours*. This is the reason, then, for our interest in parts.

E.7.1 The Urban Space, Clock, Analysis & Planning Complex

The domain-of-interest, i.e., the universe of discourse for this case study is that of *the urban space analysis & planning complex* – where the ampersand, ‘&’, shall designate that we consider this complex as ‘one’ !

951. The *universe of discourse*, UoD, is here seen as a *structure* of four elements:
- a *clock*, CLK,
 - the urban space*, TUS,
 - an *analyser aggregate*, AA,
 - the *planner aggregate*, PA,

type

951 UoD, CLK, TUS, AAG, PA

value

951a obs_CLK: UoD \rightarrow CLK
 951b obs_TUS: UoD \rightarrow TUS
 951c obs_AAG: UoD \rightarrow AAG
 951d obs_PA: UoD \rightarrow PA

The clock and the urban space are here considered *atomic*, the *analyser aggregate*, AA, and the the *planner aggregate*, PA, are here seen as *structures*.

E.7.2 The Analyser Structure and Named Analysers

952. The *analyser structure* consists of
- a *structure*, AC, which consists of two elements:
 - a *structure* of an indexed set, hence named *analysers*,
 - $A_{anm_1}, A_{anm_2}, \dots, A_{anm_n}$,
- and
953. an *atomic analysis depository*, AD.

There is therefore defined a set, ANms, of

954. *analyser names*: $\{anm_1, anm_2, \dots, anm_n\}$, where $n \geq 0$.

⁷ Epistemology is the branch of philosophy concerned with the theory of knowledge.

type

952 AA, AC, A, AD

952(a)i $A = A_{anm_1} \mid A_{anm_2} \mid \dots \mid A_{anm_n}$ 954 $ANms = \{anm_1, anm_2, \dots, anm_n\}$ **value**952a obs_AC: $AA \rightarrow AC$ 952(a)ii obs_AC $_{anm_i}$: $AC \rightarrow A_{anm_i}, i: [1..n]$ 953 obs_AD: $AA \rightarrow AD$

Analysers and the *analysis depository* are here seen as atomic parts.

E.7.3 The Planner Structure

955. The composite *planner structure* part, consists of
- a. a *master planner structure*, MPA, which consists of
 - i. an atomic *master planner server*, MPS, and
 - ii. an atomic *master planner*, MP, and
 - b. a *derived planner structure*, DPA, which consists of
 - i. a *structure* in the form of an indexed set of (hence named) *derived planner structures*, $DPC_{nm_j}, j: [1..p]$, which each consists of
 1. a atomic *derived planner servers*, $DPS_{nm_j}, j: [1..p]$, and
 2. a atomic *derived planners*, $DP_{nm_j}, j: [1..p]$;
 - c. an atomic *plan repository*, PR, and
 - d. an atomic *derived planner index generator*, DPXG.

type955 PA, MPA, MPS, MP, DPA, DPC_{nm_j} , DPS_{nm_j} , $DP_{nm_j}, i: [1..p]$ **value**955a obs_MPA: $PA \rightarrow MPA$ 955(a)i obs_MPS: $MPA \rightarrow MPS$ 955(a)ii obs_MP: $MPA \rightarrow MP$ 955b obs_DPA: $PA \rightarrow DPA$ 955(b)i obs_DPC $_{nm_j}$: $DPA \rightarrow DPC_{nm_j}, i: [1..p]$ 955(b)i1 obs_DPS $_{nm_j}$: $DPC_{nm_j} \rightarrow DPS_{nm_j}, i: [1..p]$ 955(b)i2 obs_DP $_{nm_j}$: $DPC_{nm_j} \rightarrow DP_{nm_j}, i: [1..p]$ 955c obs_PR: $PA \rightarrow PR$ 955d obs_DPXG: $\rightarrow DPXG$

We have chosen to model as *structures* what could have been modeled as *composite* parts. If we were to domain analyse & describe *management & organisation* facets of the urban space analysis & planning domain then we might have chosen to model some of these *structures* instead as *composite parts*.

E.7.4 Atomic Parts

The following are seen as atomic parts:

- *clock*,
- *urban space*,
- *analysis deposit*,
- each *analyser* in the indexed set of $analyser_{anm_i}$ s,
- *master planner server*,
- *master planner*,

- each *server* in the indexed set of *derived planner* $server_{nm_j}S$,
- each *planner* in the indexed set of *derived planner* $planner_{nm_j}S$,
- *derived planner index generator*.
- *plan repository* and

We shall return to these atomic part sorts when we explore their properties: *unique identifiers*, *merologies* and *attributes*.

E.7.5 Preview of Structures and Parts

Let us take a preview of the parts, see Fig. E.6 [Page 358].

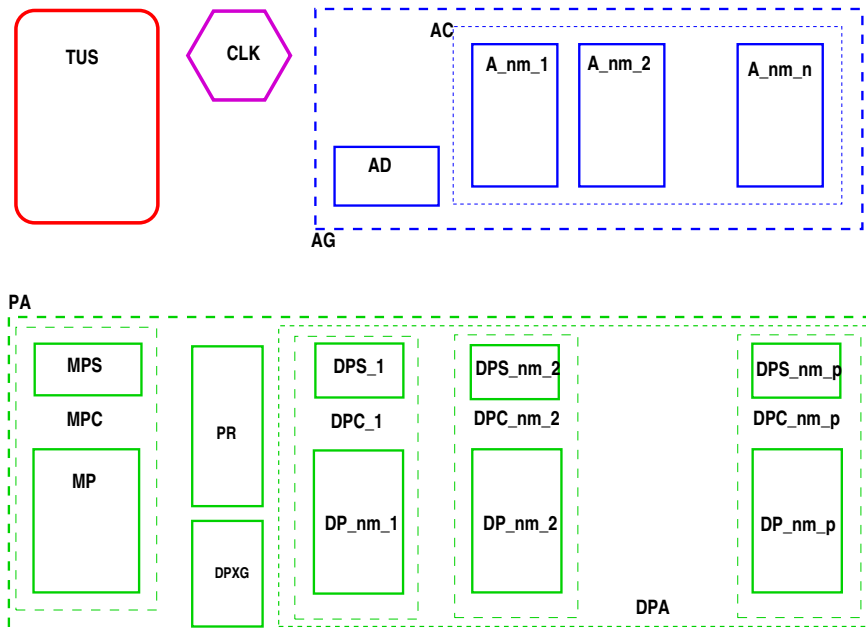


Fig. E.6. The Urban Analysis and Planning System: Structures and Atomic Parts

E.7.6 Planner Names

956. There is therefore defined identical sets of *derived planner aggregate names*, *derived planner server names*, and *derived planner names*: $\{dnm_1, dnm_2, \dots, dnm_p\}$, where $g \geq 0$.

type

956 DNms = $\{|dnm_1, dnm_2, \dots, dnm_p|\}$ uod-084

E.7.7 Individual and Sets of Atomic Parts

In this closing section of Sect. E.7.7 we shall identify individual and sets of atomic parts.

957. We postulate an arbitrary *universe of discourse*, $uod:UoD$ and let that be a constant value from which we calculate a number of individual and sets of atomic parts.
958. There is the *clock*, $clk:CLK$,
959. the *urban space*, $tus:TUS$,
960. the set of *analysers*, $a_{ann_i}:A_{ann_i}$, $i:[1..n]$,
961. the *analysis depository*, ad ,
962. the *master planner server*, $mps:MPS$,
963. the *master planner*, $mp:MP$,
964. the set of *derived planner servers*, $\{dps_{nm_i}:DPS_{nm_i} \mid i:[1..p]\}$,
965. the set of *derived planners*, $\{dp_{nm_i}:DP_{nm_i} \mid i:[1..p]\}$,
966. the *derived plan index generator*, $dpxg$,
967. the *plan repository*, pr , and
968. the set of pairs of *derived server and derived planners*, sps .

value

```

957  uod : UoD
958  clk : CLK = obs_CLK(uod)
959  tus : TUS = obs_TUS(uod)
960  ans : Aanni-set, i:[1..n] =
960      { obs_Aanni(aa) | aa∈(obs_AA(uod)), i:[1..n] }
961  ad : AD = obs_AD(obs_AA(uod))
962  mps : MPS = obs_MPS(obs_MPA(uod))
963  mp : MP = obs_MP(obs_MPA(uod))
964  dps : DPSnmi-set, i:[1..p] =
964      { obs_DPSnmi(dpcnmi) |
964          dpcnmi:DPCnmi•dpcnmi∈obs_DPCSnmi(obs_DPA(uod)), i:[1..p] }
965  dps : DPnmi-set, i:[1..p] =
965      { obs_DPnmi(dpcnmi) |
965          dpcnmi:DPCnmi•dpcnmi∈obs_DPCSnmi(obs_DPA(uod)), i:[1..p] }
966  dpxg : DPXG = obs_DPXG(uod)
967  pr : PR = obs_PR(uod)
968  spsps : (DPSnmi × DPnmi)-set, i:[1..p] =
968      { (obs_DPSnmi(dpcnmi), obs_DPnmi(dpcnmi)) |
968          dpcnmi:DPCnmi•dpcnmi∈ obs_DPCSnmi(obs_DPA(uod)), i:[1..p] }

```

E.8 Unique Identifiers

We introduce a notion of unique identification of parts. We assume (i) that all parts, p , of any domain P , have unique identifiers, (ii) that unique identifiers (of parts $p:P$) are abstract values (of the unique identifier, π , sort Π_{UI} of parts $p:P$), (iii) such that distinct part sorts, P_i and P_j , have distinctly named unique identifier sorts, say Π_{UI_i} and Π_{UI_j} , (iv) that all $\pi:\Pi_{UI_i}$ and $\pi_j:\Pi_{UI_j}$ are distinct, and (v) that the observer function uid_P applied to p yields the unique identifier, say $\pi:\Pi_{UI}$, of p .

The analysis & description of unique identification is a prerequisite for talking about mereologies of universes of discourse, and the analysis & description of mereologies are a means for understanding how parts relate to one another.

Since we model as *structures* what elsewhere might have been modeled as *composite parts* we shall only deal with *unique identifiers of atomic parts*.

E.8.1 Urban Space Unique Identifier

969. The urban space has a unique identifier.

type

969 TUS_UI

value

969 uid_TSU: TSU \rightarrow TUS_UI

E.8.2 Analyser Unique Identifiers

970. Each analyser has a unique identifier.

971. The analysis depository has a unique identifier.

type

970 A_UI = A_UI_{anm₁} | A_UI_{anm₂} | ... | A_UI_{anm_n}

971 AD_UI

value

970 uid_A: A_{nm_i} \rightarrow A_UI_{nm_i}, $i : [1..n]$

971 uid_AD: AD \rightarrow AD_UI

axiom

970 $\forall a_{nm_i} : A_{nm_i} \cdot$

970 **let** a_ui_{nm_i} = uid_A(a_{nm_i}) **in** a_ui_{nm_i} \simeq nm_i **end**

The mathematical symbol \simeq (in this case study) denotes *isomorphy*.

E.8.3 Master Planner Server Unique Identifier

972. The *unique identifier* of the *master planner server*.

type

972 MPS_UI

value

972 uid_MPS: MPS \rightarrow MPS_UI

E.8.4 Master Planner Unique Identifier

973. The *unique identifier* of the *master planner*.

type

973 MP_UI

value

973 uid_MP: MP \rightarrow MP_UI

E.8.5 Derived Planner Server Unique Identifier

974. The *unique identifiers of derived planner servers*.

```

type
974  DPS_UI = DPS_UInm1 | DPS_UInm2 | ... | DPS_UInmp
value
974  uid_DPS: DPSnmi → DPS_UInmi, i : [1..p]
axiom
974  ∀ dpsnmi:DPSnmi •
974    let dps_uinmi = uid_DPS(dpsnmi) in dps_uinmi ≈ nmi end

```

E.8.6 Derived Planner Unique Identifier

975. The *unique identifiers of derived planners*.

```

type
975  DP_UI = DP_UInm1 | DP_UInm2 | ... | DP_UInmp
value
975  uid_DP: DPnmi → DP_UInmi, i : [1..p]
axiom
975  ∀ dpnmi:DPnmi •
975    let dp_uinmi = uid_DP(dpnmi) in dp_uinmi ≈ nmi end

```

E.8.7 Derived Plan Index Generator Identifier

976. The *unique identifier of derived plan index generator*:

```

type
976  DPXG_UI
value
976  uid_DPXG: DPXG → DPXG_UI

```

E.8.8 Plan Repository

977. The *unique identifier of plan repository*:

```

type
977  PR_UI
value
977  uid_PR: PR → PR_UI

```

E.8.9 Uniqueness of Identifiers

- 978. The identifiers of all analysers are distinct.
- 979. The identifiers of all derived planner servers are distinct.
- 980. The identifiers of all derived planners are distinct.
- 981. The identifiers of all other atomic parts are distinct.
- 982. And the identifiers of all atomic parts are distinct.

978 $\text{card } ans = \text{card } a_{uis}$
 979 $\text{card } dpss = \text{card } dp_{uis}$
 980 $\text{card } dps = \text{card } dp_{uis}$
 981 $\text{card}\{clk_{ui}, tus_{ui}, ad_{ui}, mps_{ui}, mp_{ui}, dp_{xg_{ui}}, plas_{ui}\} = 7$
 982 $\cap(ans, dpss, dps, \{clk_{ui}, tus_{ui}, ad_{ui}, mps_{ui}, mp_{ui}, dp_{xg_{ui}}, plas_{ui}\}) = \{\}$

E.8.10 Indices and Index Sets

It will turn out to be convenient, in the following, to introduce a number of index sets.

983. There is the *clock* identifier, $clk_{ui}:\text{CLK_UI}$.
 984. There is the *urban space* identifier, $tus_{ui}:\text{TUS_UI}$.
 985. There is the set, $a_{uis}:\text{A_UI-set}$, of the identifiers of all *analysers*.
 986. The *analysis depository* identifier, ad_{ui} .
 987. There is the *master planner server* identifier, $mps_{ui}:\text{MPS_UI}$.
 988. There is the *master planner* identifier, $mp_{ui}:\text{MP_UI}$.
 989. There is the set, $dp_{uis}:\text{DPS_UI-set}$, of the identifiers of all *derived planner servers*.
 990. There is the set, $dp_{uis}:\text{DP_UI-set}$, of the identifiers of all *derived planners*.
 991. There is the *derived plan index generator* identifier, $dp_{xg_{ui}}:\text{DPXG_UI}$.
 992. And there is the *plan repository* identifier, $pr_{ui}:\text{PR_UI}$.

value

983 $clk_{ui} : \text{CLK_UI} = \text{uid_CLK}(uod)$
 984 $tus_{ui} : \text{TUS_UI} = \text{uid_TUS}(uod)$
 985 $a_{uis} : \text{A_UI-set} = \{\text{uid_A}(a) \mid a:A \cdot a \in ans\}$
 986 $ad_{ui} : \text{AD_UI} = \text{uid_AD}(ad)$
 987 $mps_{ui} : \text{MPS_UI} = \text{uid_MPS}(mps)$
 988 $mp_{ui} : \text{MP_UI} = \text{uid_MP}(mp)$
 989 $dp_{uis} : \text{DPS_UI-set} = \{\text{uid_DPS}(dps) \mid dps:\text{DPS} \cdot dps \in dpss\}$
 990 $dp_{uis} : \text{DP_UI-set} = \{\text{uid_DP}(dp) \mid dp:\text{DP} \cdot dp \in dps\}$
 991 $dp_{xg_{ui}} : \text{DPXG_UI} = \text{uid_DPXG}(dp_{xg})$
 992 $pr_{ui} : \text{PR_UI} = \text{uid_PR}(pr)$

993. There is also the set of identifiers for all servers: $ps_{uis}:(\text{MPS_UI} \mid \text{DPS_UI})\text{-set}$,
 994. there is then the set of identifiers for all planners: $ps_{uis}:(\text{MP_UI} \mid \text{DP_UI})\text{-set}$,
 995. there is finally the set of pairs of paired *derived planner server* and *derived planner* identifiers.
 996. there is a map from the unique derived server identifiers to their “paired” unique derived planner identifiers, and
 997. there is finally the reverse map from planner to server identifiers.

value

993 $s_{uis} : (\text{MPS_UI} \mid \text{DPS_UI})\text{-set} = \{mps_{ui}\} \cup dp_{uis}$
 994 $p_{uis} : (\text{MP_UI} \mid \text{DP_UI})\text{-set} = \{mp_{ui}\} \cup dp_{uis}$
 995 $sips : (\text{DPS_UI} \times \text{DP_UI})\text{-set} = \{(\text{uid_DPS}(dps), \text{uid_DP}(dp)) \mid (dps, dp):(\text{DPS} \times \text{DP}) \cdot (dps, dp) \in sps\}$
 996 $si_pi_m : \text{DPS_UI} \xrightarrow{m} \text{DP_UI} = [\text{uid_DPS}(dps) \mapsto \text{uid_DP}(dp) \mid (dps, dp):(\text{DPS} \times \text{DP}) \cdot (dps, dp) \in sps]$
 997 $pi_si_m : \text{DP_UI} \xrightarrow{m} \text{DPS_UI} = [\text{uid_DP}(dp) \mapsto \text{uid_DPS}(dps) \mid (dps, dp):(\text{DPS} \times \text{DP}) \cdot (dps, dp) \in sps]$

E.8.11 Retrieval of Parts from their Identifiers

998. Given the global set $dpss$, cf. 964 [Page 359], i.e., the set of all derived servers, and given a unique planner server identifier, we can calculate the derived server with that identifier.
999. Given the global set dps , cf. 965 [Page 359], the set of all derived planners, and given a unique derived planner identifier, we can calculate the derived planner with that identifier.

value

```

998  c_s: dpss → DPS_UI → DPS
998  c_s(dpss)(dps_ui) ≡ let dps:DPS•dps ∈ dpss∧uid_DPS(dps)=dps_ui in dps end
999  c_p: dps → DP_UI → DP
999  c_p(dps)(dp_ui) ≡ let dp:DP•dp ∈ dps∧uid_DPS(dp)=dp_ui in dp end

```

E.8.12 A Bijection: Derived Planner Names and Derived Planner Identifiers

We can postulate a unique relation between the names, $dn:DNm$ -set, i.e., the names $dn \in DNms$, and the unique identifiers of the named planners:

1000. We can claim that there is a function, $extr_DNm$, from the unique identifiers of derived planner servers to the names of these unique identifiers.
1001. Similarly can claim that there is a function, $extr_DNm$, from the unique identifiers of derived planners to the names of these unique identifiers.

value

```

1000 extr_Nm: DPS_UI → DNm
1000 extr_Nm(dps_ui) ≡ ...
1001 extr_Nm: DP_UI → DNm
1001 extr_Nm(dp_ui) ≡ ...

```

axiom

```

1000 ∀ dps_ui1,dps_ui2:DPS_ui • dps_ui1≠dps_ui2 ⇒ extr_Nm(dps_ui1) ≠ extr_Nm(dps_ui2)
1001 ∀ dp_ui1,dp_ui2:DP_ui • dp_ui1≠dp_ui2 ⇒ extr_Nm(dp_ui1) ≠ extr_Nm(dp_ui2)

```

1002. Let $dps_ui_dnm:DPS_UI_DNm$, $dp_ui_dnm:DP_UI_DNm$ stand for maps from derived planner server, respectively derived planner unique identifiers to derived planner names.
1003. Let $nm_dp_ui:Nm_DP_UI$, $nm_dp_ui:Nm_DP_UI$ stand for the reverse maps.
1004. These maps are bijections.

type

```

1002 DPS_UI_DNm: DPS_UI →m DP_Nm
1002 DP_UI_DNm: DP_UI →m DP_Nm
1003 DNm_DPS_UI: DP_Nm →m DP_UI
1003 DNm_DP_UI: DP_Nm →m DP_UI

```

axiom

```

1004 ∀ dps_ui_dnm:DPS_UI_DNm • dps_ui_dnm-1•dps_ui_dnm = λx.x
1004 ∀ dp_ui_dnm:DP_UI_DNm • dp_ui_dnm-1•dp_ui_dnm = λx.x
1004 ∀ dnm_dps_ui:DNm_DPS_UI • dnm_dps_ui-1•dnm_dps_ui = λx.x
1004 ∀ dnm_dp_ui:DNm_DP_UI • dp_ui_dnm-1•dnm_dp_ui = λx.x

```

that is:

```

1004 ∀ dps_ui_dnm:DPS_UI_DNm, dp_ui_dnm:DP_UI_DNm, dps_ui:DPS_UI •
1004     dps_ui ∈ dom dps_ui_dnm ⇒ dp_ui_dnm(dps_ui_dnm(dps_ui)) = dps_ui
    et cetera !

```

1005. The function `mk_DNm_DUI` takes the set of all derived planner servers, respectively derived planners and produces bijective maps, `dnm_dps_ui`, respectively `dnm_dp_ui`.

1006. Let $dnm_dps_ui:DNm_DPS_UI$ and

1007. $dnm_dp_ui:DNm_DP_UI$

stand for such [global] maps.

value

1005 $mk_Nm_DPS_UI: DPS_{nm_i}\text{-set} \rightarrow DNm_DPS_UI$

1005 $mk_Nm_DPS_UI(dps) \equiv [uid_DPS(dps) \mapsto extr_Nm(uid_DPS(dps)) | dps:DPS \cdot dps \in dps]$

1005 $mk_Nm_DP_UI: DP_{nm_i}\text{-set} \rightarrow DNm_DP_UI$

1005 $mk_Nm_DP_UI(dps) \equiv [uid_DP(dp) \mapsto extr_Nm(uid_DP(dp)) | dp:DP \cdot dps \in dps]$

1006 $nm_dps_ui:Nm_DPS_UI = mk_Nm_DPS_UI(dps)$

1007 $nm_dp_ui:Nm_DP_UI = mk_Nm_DP_UI(dps)$

E.9 Mereologies

Mereology (from the Greek $\mu\epsilon\rho\omicron\varsigma$ ‘part’) is the *theory of part-hood relations: of the relations of part to whole and the relations of part to part within a whole*⁸.

Part mereologies inform of how parts relate to other parts. As we shall see in the section on *perdurants*, mereologies are the basis for analysing & describing communicating between part behaviours.

Again: since we model as *structures* what is elsewhere modeled as *composite parts* we shall only consider *mereologies of atomic parts*.

E.9.1 Clock Mereology

1008. The clock is related to all those parts that create information, i.e., documents of interest to other parts.

Time is then used to *time-stamp* those documents. These other parts are: the *urban space*, the *analysers*, the *planner servers* and the *planners*.

type

1008 $CLK_Mer = TSU_UI \times A_UI\text{-set} \times MPS_UI \times MP_UI \times DPS_UI\text{-set} \times DP_UI\text{-set}$

value

1008 $mereo_CLK: CLK \rightarrow Clk_Mer$

axiom

1008 $mereo_CLK(uod) = (tus_{ui}, a_{uis}, mps_{ui}, mp_{ui}, dps_{uis}, dp_{uis})$

E.9.2 Urban Space Mereology

The urban space stands in relation to those parts which consume urban space information: the *clock* (in order to time stamp urban space information), the *analysers* and the *master planner server*.

1009. The mereology of the urban space is a triple of the clock identifier, the identifier of the master planner server and the set of all analyser identifiers. all of which are provided with urban space information.

1010. The constraint here is expressed in the ‘the’: for the universe of discourse it must be the master planner aggregate unique identifier and the set of exactly all the analyser unique identifiers for that universe.

⁸ Achille Varzi: Mereology, <http://plato.stanford.edu/entries/mereology/> 2009 and [104].

type
 1009 TUS_Mer = CLK_UI × A_UI-set × MPS_UI
value
 1009 mereo_TUS: TUS → TUS_Mer
axiom
 1010 mereo_TUS(tus) = (clk_{ui}, a_{uis}, mps_{ui})

E.9.3 Analyser Mereology

1011. The mereology of a[ny] *analyser* is that of a triple: the *clock identifier*, the *urban space identifier*, and the *analysis depository identifier*.

type
 1011 A_Mer = CLK_UI × TUS_UI × AD_UI
value
 1011 mereo_A: A → A_Mer

E.9.4 Analysis Depository Mereology

1012. The mereology of the *analysis depository* is a triple: the *clock identifier*, the *master planner server identifier*, and the set of *derived planner server identifiers*.

type
 1012 AD_Mer = CLK_UI × MPS_UI × DPS_UI-set
value
 1012 mereo_AD: AD → AD_Mer

E.9.5 Master Planner Server Mereology

1013. The *master planner server* mereology is a quadruplet of the clock identifier (time is used to time stamp input arguments, prepared by the server, to the planner), the urban space identifier, the analysis depository and the master planner identifier.

1014. And for all universes of discourse these must be exactly those of that universe.

type
 1013 MPS_Mer = CLK_UI × TUS_UI × AD_UI × MP_UI
value
 1013 mereo_MPS: MPS → MPS_Mer
axiom
 1014 mereo_MPS(mps) = (clk_{ui}, tus_{ui}, ad_{ui}, mp_{ui})

E.9.6 Master Planner Mereology

1015. The mereology of the *master planner* is a triple of: the clock identifier⁹, master server identifier¹⁰, derived planner index generator identifier¹¹, and the plan repository identifier¹².

⁹ From the clock the planners obtain the time with which they stamp all information assembled by the planner.

¹⁰ from which the master planner obtains essential input arguments

¹¹ in collaboration with which the master planner obtains a possibly empty set of derived planning indices

¹² with which it posits and from which it obtains summaries of all urban planning plans produced so far.

type1015 $MP_Mer = CLK_UI \times MPS_UI \times DPXG_UI \times PR_UI$ **value**1015 mereo_MP: $MP \rightarrow MP_Mer$ **axiom**1015 $mereo_MP(mp) = (clk_{ui}, mps_{ui}, dpxg_{ui}, pr_{ui})$ **E.9.7 Derived Planner Server Mereology**1016. The *derived planner server* mereology is a quadruplet of:

the clock identifier¹³, the set of all analyser identifiers¹⁴, the plan repository identifier,¹⁵ and the derived planner identifier¹⁶.

type1016 $DPS_Mer = CLK_UI \times AD_UI \times PLAS_UI \times DP_UI$ **value**1016 mereo_DPS: $DPS \rightarrow DPS_Mer$ **axiom**

1016 $\forall (dps, dp): (DPS \times DP) \cdot (dps, dp) \in sps \Rightarrow$
 1016 $mereo_DPS(dps) = (clk_{ui}, ad_{ui}, plas_{ui}, uid_DP(dp))$

E.9.8 Derived Planner Mereology1017. The *derived planner* mereology is a quadruplet of:

the clock identifier, the derived plan server identifier, the derived plan index generator identifier, and the plan repository identifier.

type1017 $DP_Mer = CLK_UI \times DPS_UI \times DPXG_UI \times PR_UI$ **value**1017 mereo_DP: $DP \rightarrow DP_Mer$ **axiom**

1017 $\forall (dps, dp): (DPS \times DP) \cdot (dps, dp) \in sps \Rightarrow$
 1017 $mereo_DP(dp) = (clk_{ui}, uid_DPS(dps), dpxg_{ui}, pr_{ui})$

¹³ From the clock the servers obtain the time with which they stamp all information assembled by the servers.

¹⁴ From the analysers the servers obtain analyses.

¹⁵ In collaboration with the plan repository the planners deposit plans etc. and obtains summaries of all urban planning plans produced so far

¹⁶ The server provides its associated planner with appropriate input arguments.

E.9.9 Derived Planner Index Generator Mereology

1018. The mereology of the derived planner index generator is the set of all planner identifiers: master and derived.

type

1018 DPXG_Mer = (MP_UI|DP_UI)-set

value

1018 mereo_DPXG: DPXG \rightarrow DPXG_Mer

axiom

1018 mereo_DPXG(dp_xg) = $ps_{ui}S$

E.9.10 Plan Repository Mereology

1019. The plan repository mereology is the set of all planner identifiers: master and derived.

1019 PR_Mer = (MP_UI|DP_UI)-set

value

1019 mereo_PR: PR \rightarrow PR_Mer

axiom

1019 mereo_PR(pr) = $ps_{ui}S$

E.10 Attributes

Parts are typically recognised because of their spatial form and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts and components) or continuous (as are materials), are physical, tangible, in the sense of being spatial (or being abstractions, i.e., concepts, of spatial endurants), attributes are intangible: cannot normally be touched, or seen, but can be objectively measured. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective “attributes”: feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. A formal concept, that is, a type, consists of all the entities which all have the same qualities. Thus removing a quality from an entity makes no sense: the entity of that type either becomes an entity of another type or ceases to exist (i.e., becomes a non-entity)

E.10.1 Clock Attribute

Time and Time Intervals and their Arithmetic

1020. Time is modeled as a continuous entity.

1021. One can subtract two times and obtain a time interval.

1022. There is an “infinitesimally” smallest time interval, $\delta t: T$.

1023. Time intervals are likewise modeled as continuous entities.

1024. One can add or subtract a time interval to, resp. from a time and obtain a time.

1025. One can compare two times, or two time intervals.

1026. One can add and subtract time intervals.

1027. One can multiply time intervals with real numbers.

```

type
1020 T
1021 TI
value
1021 sub:  $T \times T \rightarrow TI$ 
1022  $\delta t$ : TI
1024 add,sub:  $TI \times T \rightarrow T$ 
1025  $\langle, \leq, =, \geq, \rangle$ :  $((T \times T) | (TI \times TI)) \rightarrow \mathbf{Bool}$ 
1026 add,sub:  $TI \times TI \rightarrow TI$ 
1027 mpy:  $TI \times \mathbf{Real} \rightarrow TI$ 

```

The Attribute

1028. The only attribute of a clock is time. It is a programmable attribute.

```

type
1028 T
value
1028 attr_T:  $CLK \rightarrow T$ 
axiom
1028  $\forall clk: CLK \cdot$ 
1028   let (t,t') = (attr_CLK(clk);attr_CLK(clk)) in
1028    $t \leq t'$  end

```

The ';' in an expression (a;b) shall mean that first expression a is evaluated, then expression b.

E.10.2 Urban Space Attributes

The Urban Space

1029. We shall assume a notion of *the urban space*, tus:TUS, from which we can observe the attribute:
 1030. an infinite, compact Euclidean set of points.
 1031. By a *point* we shall understand a further undefined atomic notion.
 1032. By an *area* we shall understand a concept, related to the urban space, that allows us to speak of “a point being in an area” and “an area being equal to or properly within another area”.
 1033. To an[y] *urban space* we can associate an area; we may think of an area being an *attribute* of the urban space.

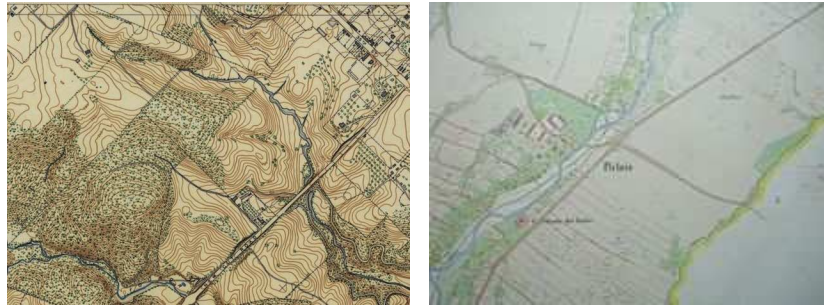
```

type
1029 TUS
1030 PtS = Pt-infsetsac-11
value
1029 attr_PtS:  $TUS \rightarrow Pt\text{-infset}$ 
type
1031 Ptsac00
1032 Areasac10
value
1033 attr_Area:  $TUS \rightarrow Area$ 
1032 is_Pt_in_Area:  $Pt \times (TUS|Area) \rightarrow \mathbf{Bool}$ 
1032 is_Area_within_Area:  $Area \times (TUS|Area) \rightarrow \mathbf{Bool}$ 

```


The Urban Space Attributes

By *urban space attributes* we shall here mean the facts by means of which we can characterize that which is subject to urban planning: the land, what is in and on it: its geodetics, its cadastra¹⁷, its meteorology, its socio-economics, its rule of law, etc. As such we shall consider ‘the urban space’ to be a *part* in the sense of [70]. And we shall consider the *geodetic, cadastral, geotechnical, meteorological, “the law”* (i.e., *state, province, city and district ordinances*) and *socio-economic* properties as *attributes*.



Left: geodetic map, right: cadastral map.

Main Part and Attributes

One way of observing *the urban space* is presented: to the left, in the framed box, we **narrate** the story; to the right, in the framed box, we **formalise** it.

1034. The Urban Space (TUS) has the following	tus000tus000tus000tus000tus000tus000
a. PointSpace attributes,	
b. Geodetic attributes,	
c. Cadastre attributes,	
d. Geotechnical attributes,	value
e. Meteorological attributes,	1034a attr_Pts: TUS → PtS
f. Law attributes,	1034b attr_GeoD: TUS → GeoD
g. Socio-Economic attributes, etcetera.	1034c attr_Cada: TUS → Cada
	1034d attr_GeoT: TUS → GeoT
	1034e attr_Met: TUS → Met
type	1034f attr_Law: TUS → Law
1034 TUS, PtS, GeoD, Cada, GeoT, Met, Law, SocEco	1034g attr_SocEco: TUS → SocEco

The $\text{attr}_A: P \rightarrow A$ is the **signature** of a postulated *attribute (observer) function*. From parts of type P it **observes** attributes of type A. attr_A are postulated functions. They express that we can always observe attributes of type A of parts of type P.

Urban Space Attributes – Narratives and Formalisation

We describe attributes of the domain of urban spaces. As they are, in real life. Not as we may record them or represent them (on paper or within the computer). We can “freely” model that reality as we think it is. If we can talk about and describe it, then it is so ! For meteorological attributes it means that we describe precipitation, evaporation, humidity and atmospheric pressure as these physical phenomena “really” are: continuous over time ! Similar for all other attributes. Etcetera.

¹⁷ Cadastra: A Cadastra is normally a parcel based, and up-to-date land information system containing a record of interests in land (e.g. rights, restrictions and responsibilities). It usually includes a geometric description of land parcels linked to other records describing the nature of the interests, the ownership or control of those interests, and often the value of the parcel and its improvements. See <http://www.fig.net/>

General Form of Attribute Models

1035. We choose to model the *General Form of Attributes*, such as geodetical, cadastral, geotechnical, meteorological, socio-economic, legal, etcetera, as [continuous] functions from time to maps from points or areas to the specific properties of the attributes.
1036. The points or areas of the properties maps must be in, respectively within, the area of the urban space whose attributes are being specified.

```

type
1035  GFA = T → ((Pt|Area) →m Properties)gfoam00
value
1036  wf_GFA: GFA × TUS → Bool
1036  wf_GFA(gfa,tus) ≡
1036    let area = attr_Area(tus) in
1036    ∀ t:T • t ∈  $\mathcal{D}$  gfa ⇒
1036      ∀ pt:Pt • pt ∈ dom gfa(t) ⇒ is_Pt_in_Area(pt,area)
1036      ∧ ∀ ar:Area • ar ∈ dom gfa(t) ⇒ is_within_Area(ar,area)
1036    end

```

\mathcal{D} is a hypothesized function which applies to continuous functions and yield their domain !

Geodetic Attribute[s]

1037. Geodetic attributes map points to
- land elevation and what kind of land it is; and (or) to
 - normal and current water depths and what kind of water it is.
1038. Geodetic attributes also includes road nets and what kind of roads;
1039. etcetera,

```

type
1037  GeoD = T → (Pt →m ((Land|Water) × RoadNet × ...))
1037a  Land = Elevation × (Farmland|Urban|Forest|Wilderness|Meadow|Swamp|...)
1037b  Water = (NormDepth × CurrDepth) × (Spring|Creek|River|Lake|Dam|Sea|Ocean|...)
1038  RoadNet = ...
1039  ...

```

Cadastral Attribute[s]

A cadastre is a public register showing details of ownership of the real property in a district, including boundaries and tax assessments.

1040. Cadastral maps shows the boundaries and ownership of land parcels. Some cadastral maps show additional details, such as survey district names, unique identifying numbers for parcels, certificate of title numbers, positions of existing structures, section or lot numbers and their respective areas, adjoining and adjacent street names, selected boundary dimensions and references to prior maps.
1041. Etcetera.

```

type
1040  Cada = T → (Area →m (Owner × Value × ...))
1041  ...

```

Geotechnical Attribute[s]

1042. Geotechnical attributes map points to
- top and lower layer soil etc. composition, by depth levels,
 - ground water occurrence, by depth levels,
 - gas, oil occurrence, by depth levels,
 - etcetera.

type

- 1042 $\text{GeoT} = (\text{Pt} \rightarrow_{\text{h}} \text{Composition})$
 1042a $\text{Composition} = \text{VerticalScaleUnit} \times \text{Composite}^*$
 1042b $\text{Composite} = (\text{Soil}|\text{GroundWater}|\text{Sand}|\text{Gravel}|\text{Rock}|\dots|\text{Oil}|\text{Gas}|\dots)$
 1042c $\text{Soil,Sand,Gravel,Rock},\dots,\text{Oil,Gas},\dots = [\text{chemical analysis}]$
 1042d ...

Meteorological Attribute[s]

1043. Meteorological information records, for points (of an area) precipitation, evaporation, humidity, etc.;
- precipitation: the amount of rain, snow, hail, etc.; that has fallen at a given place and at the time-stamped moment¹⁸, expressed, for example, in milimeters of water;
 - evaporation: the amount of water evaporated (to the air);
 - atmospheric pressure;
 - air humidity;
 - etcetera.

- 1043 $\text{Met} = \text{T} \rightarrow (\text{Pt} \rightarrow_{\text{h}} (\text{Precip} \times \text{Evap} \times \text{AtmPress} \times \text{Humid} \times \dots))$
 1043a $\text{Precip} = \text{MMs} [\text{milimeters}]$
 1043b $\text{Evap} = \text{MMs} [\text{milimeters}]$
 1043c $\text{AtmPress} = \text{MB} [\text{milibar}]$
 1043d $\text{Humid} = \text{Percent}$
 1043e ...

Socio-Economic Attribute[s]

1044. Socio-economic attributes include time-stamped area sub-attributes:
- income distribution;
 - housing situation, by housing category: apt., etc.;
 - migration (into, resp. out of the area);
 - social welfare support, by citizen category;
 - health status, by citizen category;
 - etcetera.

type

- 1044 $\text{SocEco} = \text{T} \rightarrow (\text{Area} \rightarrow_{\text{h}} (\text{Inc} \times \text{Hou} \times \text{Mig} \times \text{SoWe} \times \text{Heal} \times \dots))$
 1044a $\text{Inc} = \dots$
 1044b $\text{Hou} = \dots$
 1044c $\text{Mig} = \{[\text{"in"}, \text{"out"}]\} \rightarrow_{\text{h}} (\{[\text{"male"}, \text{"female"}]\} \rightarrow_{\text{h}} (\text{Agegroup} \times \text{Skills} \times \text{HealthSumm} \times \dots))$
 1044d $\text{SoWe} = \dots$
 1044e $\text{CommHeal} = \dots$
 1044f ...

¹⁸ – that is within a given time-unit

Law Attribute[s]: State, Province, Region, City and District Ordinances

1045. By the law we mean any state, province, region, city, district or other 'area' ordinance¹⁹.

1046. ...

type

1045 Law

value

1045 attr_Law: TUS \rightarrow Law

type

1045 Law = Area \xrightarrow{m} Ordinances

1046 ...

Industry and Business Economics

TO BE WRITTEN

Etcetera

TO BE WRITTEN

The Urban Space Attributes – A Summary

Summarising we can model the aggregate of urban space attributes as follows.

1047. Each of these attributes can be given a name.

1048. And the aggregate can be modelled as a map (i.e., a function) from names to appropriately typed attribute values.

type

1047 TUS_Attr_Nm = {"pts", "ged", "cad", "get", "law", "eco", ...}

1048 TUSm = TUS_Attr_Nm \xrightarrow{m} TUS_Attr

axiom

1048 \forall tusm:TUSm \cdot \forall nm:TUS_Attr_Nm \cdot nm \in **dom** tusm \Rightarrow

1048 **case** (nm, mtusm(nm)) **of**

1048 ("pts", v) \rightarrow is_PtS(v), ("ged", v) \rightarrow is_GeoD(v), ("cad", v) \rightarrow is_CaDa(v),

1048 ("get", v) \rightarrow is_GeoT(v), ("law", v) \rightarrow is_Law(v), ("eco", v) \rightarrow is_Eco(v), ...

1048 **end**

Discussion

TO BE WRITTEN

¹⁹ Ordinance: a law set forth by a governmental authority; specifically a municipal regulation: for ex.: *A city ordinance forbids construction work to start before 8 a.m.*

E.10.3 Scripts

The concept of *scripts* is relevant in the context of *analysers* and *planners*.

By a *script* we shall understand the structured, almost, if not outright, formally expressed, wording of a procedure on how to proceed, one that may have legally binding power, that is, which may be contested in a court of law.

Those who *contract* urban analyses and urban plannings may wish to establish that some procedural steps are taken. Examples are: the vetting of urban space information, the formulation of requirements to what the analysis must contain, the vetting of that and its “quality”, the order of procedural steps, etc. We refer to [76, 83].

A[ny] *script*, as implied above, is “like a program”, albeit to be “computed” by humans.

Scripts may typically be expressed in some notation that may include: graphical renditions, like that of Fig. E.2 [Page 351], that illustrate that two or more independent groups of people, are expected to perform a number of named and more-or-less loosely described actions, expressed in, for example, the technical (i.e., domain) language of urban analysis, respectively urban planning.

The design of urban analysis and of urban planning scripts is an experimental research project with fascinating prospects for further understanding *what urban analysis and urban planning is*.

E.10.4 Urban Analysis Attributes

1049. Each *analyser* is characterised by a script, and
 1050. the set of master and/or derived planner server identifiers – meaning that their “attached” planners might be interested in its analysis results.

type

1049 $A_Script = A_Script_{anm_1} \mid A_Script_{anm_2} \mid \dots \mid A_Script_{anm_n}$ uaa-000uaa-000

1050 $A_Mer = (MPS_UI \mid DPS_UI)\text{-set}$ uaa-010

value

1049 $attr_A_Script: A \rightarrow A_Scripts$

1050 $attr_A_Mer: A \rightarrow A_Mer$

axiom

1050 $\forall a:A \cdot a \in ans \Rightarrow attr_A_Mer(a) \subseteq ps_{uis}$

E.10.5 Analysis Depository Attributes

The purpose of the *analysis depository* is to *accept*, *store* and *distribute* collections of *analyses*; it *accepts* these analysis from the analysers. it *stores* these analyses “locally”; and it *distributes* aggregates of these analyses to *plan servers*.

1051. The *analysis depository* has just one attribute, AHist. It is modeled as a map from *analyser names* to *analysis histories*.
 1052. An *analysis history* is a time-ordered sequence, of time stamped analyses, most recent analyses first.

type

1051 $AHist = ANm \xrightarrow{h} (s_T:T \times s_Anal:Anal_{anm_i})^*$ ada-000

value

1051 $attr_AHist: AD \rightarrow AHist$

axiom

1052 $\forall ah:AHist, anm:ANm \cdot anm \in \mathbf{dom} \ ah \Rightarrow$

1052 $\quad \forall i:\mathbf{Nat} \cdot \{i, i+1\} \subseteq \mathbf{inds} \ ah(anm) \Rightarrow$

1052 $\quad \quad s_T((ah(nm))[i]) > s_T((ah(nm))[i+1])$

E.10.6 Master Planner Server Attributes

The *planner servers*, whether for *master planners* or *derived planners*, assemble arguments for their associated (i.e., ‘paired’) planners. These arguments include information *auxiliary* to other arguments, such as urban space information for the master planner, and analysis information for all planners; in addition the server also provides *requirements* that are resulting planner plans are expected to satisfy. For every iteration of the planner behaviour the pair of *auxiliary* and *requirements* information is to be renewed and the renewed pairs must somehow “fit” the previously issued pairs.

1053. The *programmable* attributes of the master planner server are those of aux:AUXiliaries and req:REQUIREments.

1054. We postulate a predicate function, fit_mAux_mReq, which takes a pair of pairs auxiliary and requirements arguments, and yields a truth value.

type

1053 mAUX, mREQmplaser-000mplaser-000

value

1053 attr_mAUX: MPS \rightarrow mAUX

1053 attr_mREQ: MPS \rightarrow mREQ

1054 fit_mAUX_mReq: (mAUX \times mREQ) \times (mAUX \times mREQ) \rightarrow **Bool**

1054 fit_mAUX_mReq(arg_prev, arg_new) \equiv ...

E.10.7 Master Planner Attributes

The *master planner* has the following attributes:

1055. a *master planner script* which is a *static attribute*;

1056. an aggregate of *script “counters”*, a *programmable attribute*; the aggregate designates *pointers* in the *master script* where resumption of *master planning* is to take place in a resumed planning;

1057. a set of *names* of the *analysers* whose analyses the master planner is, or may be interested in, a *static attribute*; and

1058. a set of *identifiers* of the *derived planners* which the master planner may initiate *static attribute*.

type

1055 MP_Script mpa-000

1056 MP_Script_Pt mpa-000

1056 MP_Script_Pts = MP_Script_pt-set mpa-000

1057 ANms = ANm-set mpa-010

1058 DPUs = DP_UI-set mpa-020

value

1055 attr_MP_Script: MP \rightarrow MP_Script

1056 attr_Script_Pts: MP \rightarrow MP_Script_Pts

1057 attr_ANms: MP \rightarrow ANms

1058 attr_DPUs: MP \rightarrow DPUs

axiom

1057 attr_ANms(*mp*) \subseteq ANms

1058 attr_DPNms(*mp*) \subseteq DNms

E.10.8 Derived Planner Server Attributes

1059. The *programmable* attributes, of the derived planner servers are those of aux:AUXiliaries and req:REQUIREments, one each of an indexed set.

1060. We postulate an indexed predicate function, fit_mAux_mReq, which takes a pair of pairs auxiliary and requirements arguments, and yields a truth value.

type

1053 dAUX = dAUX_{dnm₁} | dAUX_{dnm₂} | ... | dAUX_{dnm_p} mplaser-000mplaser-000

1053 dREQ = dREQ_{dnm₁} | dREQ_{dnm₂} | ... | dREQ_{dnm_p} mplaser-000mplaser-000

value

1059 $\text{attr_dAUX}_{dnm_i}: \text{MPS}_{dnm_i} \rightarrow \text{dAUX}_{dnm_i}$
 1059 $\text{attr_dREQ}_{dnm_i}: \text{MPS}_{dnm_i} \rightarrow \text{dREQ}_{dnm_i}$
 1060 $\text{fit_dAUX_dReq}_{dnm_i_dReq}_{dnm_i}: (\text{dAUX}_{dnm_i} \times \text{dREQ}_{dnm_i}) \times (\text{dAUX}_{dnm_i} \times \text{dREQ}_{dnm_i}) \rightarrow \mathbf{Bool}$
 1060 $\text{fit_dAUX_dReq}_i(\text{arg_prev}_{dnm_i}, \text{arg_new}_{dnm_i}) \equiv \dots$

E.10.9 Derived Planner Attributes

1061. a *derived planner script* which is a *static attribute*;
 1062. an aggregate of *script “counters”*, a *programmable attribute*; the aggregate designates *points* in the *derived planner script* where resumption of *derived planning* is to take place in a resumed planning;
 1063. a set of *identifiers* of the *analysers* whose analyses the master planner is, or may be interested in, a *static attribute*; and
 1064. a set of *identifiers* of the *derived planners* which any specific derived planner may initiate, a *static attribute*.

type		1061	$\text{attr_MP_Script}: \text{MP} \rightarrow \text{MP_Script}$
1061	DP_Scriptdpa-000	1062	$\text{attr_Script_Pts}: \text{MP} \rightarrow \text{Script_Pts}$
1062	$\text{DP_Script_ptdpa-005}$	1063	$\text{attr_ANms}: \text{MP} \rightarrow \text{ANms}$
1062	$\text{DP_Script_Pts} = \text{DP_Script_pt}^* \text{dpa-005}$	1064	$\text{attr_DNms}: \text{MP} \rightarrow \text{DNms}$
1063	ANmsdpa-010	axiom	
1064	DNmsdpa-020	1063	$\text{attr_AUIs}(mp) \subseteq \text{ANms}$
value		1064	$\text{attr_DPUIs}(mp) \subseteq \text{DNms}$

E.10.10 Derived Planner Index Generator Attributes

The *derived planner index generator* has two attributes:

1065. the set of all derived planner identifiers (a static attribute), and
 1066. a set of already used planner identifiers (a programmable attribute).

type	
1065	$\text{All_DPUIs} = \text{DP_UI-setdpiga-000}$
1066	$\text{Used_DPUIs} = \text{DP_UI-setdpiga-010}$
value	
1065	$\text{attr_All_DPUIs}: \text{DPXG} \rightarrow \text{All_DPUIs}$
1066	$\text{attr_Used_DPUIs}: \text{DPXG} \rightarrow \text{Used_DPUIs}$
axiom	
1065	$\text{attr_All_DPUIs}(dpxg) = dp_{uis}$
1066	$\text{attr_Used_DPUIs}(dpxg) \subseteq dp_{uis}$

E.10.11 Plan Repository Attributes

The rôle of the *plan repository* is to keep a record of all master and derived plans. There are two plan repository attributes.

1067. A bijective map between derived planner identifiers and names, and
 1068. a pair of a list of time-stamped master plans and a map from derived planner names to lists of time-stamped plans, where the lists are sorted in time order, most recent time first.

```

type
1067 NmUIm = DNm  $\rightarrow_{\mathcal{M}}$  DP_UIpra-000
1068 PLANS = ((MP_UI|DP_UI)  $\rightarrow_{\mathcal{M}}$  (s_t:T  $\times$  s_pla:PLA)*)pra-010
value
1067 attr_NmUIm: PR  $\rightarrow$  NmUIm
axiom
1067  $\forall$  bm:NmUIm  $\cdot$  bm-1(bm)  $\equiv$   $\lambda$ x.x
value
1067 attr_PLANS: PR  $\rightarrow$  PLANS
axiom
1068 let plans = attr_PLANS(pr) in
1068 dom plans  $\subseteq$  {mpui}  $\cup$  dpuis
1068  $\forall$  pui:(MP_UI|DP_UI)  $\cdot$  pui  $\in$  {mpui}  $\cup$  dpuis  $\Rightarrow$  time_ordered(plans(pui))
1068 end
value
1068 time_ordered: (s_t:T  $\times$  s_pla:PLA)*  $\rightarrow$  Bool
1068 time_ordered(tsl)  $\equiv$   $\forall$  i:Nat  $\cdot$  {i,i+1}  $\subseteq$  inds tsl  $\Rightarrow$  s_t(sl(i)) > s_t(tsl(i+1))

```

E.10.12 A System Property of Derived Planner Identifiers

Let there be given the set of derived planners *dps*.

1069. The function reachable identifiers is the one that calculates all derived planner identifiers reachable from a given such identifier, *dp_ui*:DP_UI, in *dps*.
- We calculate the derived planner, *dp*:DP, from *dp_ui*.
 - We postulate a set of unique identifiers, *uis*, initialised with those that can be in the attr_DPUIs(*dp*) attribute.
 - Then we recursively calculate the derived planner identifiers that can be reached from any identifier, *ui*, in *uis*.
 - The recursion reaches a fix-point when there are no more identifiers “added” to *uis* in an iteration of the recursion.
1070. A derived planner must not “circularly” refer to itself.

```

value
1069 reachable_identifiers: DP-set  $\times$  DP_UI  $\rightarrow$  DP_UI-set
1069 (dps)(dp_ui)  $\equiv$ 
1069a let dp = c_p(dps)(dp_ui) in
1069b let uis = attr_DPUIs(dp)  $\cup$ 
1069c {ui|ui:DP_UI  $\cdot$  ui  $\in$  uis  $\wedge$  ui  $\in$  reachable_identifiers(dps)(ui)}
1069d in uis end end

1070  $\forall$  ui:DP_UI  $\cdot$  ui  $\in$  dpuis  $\Rightarrow$  ui  $\notin$  names(dps)(ui)

```

The seeming “endless recursion” ends when an iteration of the *dns* construction and its next does not produce new names for *dns* — a least fix-point has been reached.

E.11 PERDURANTS

By a *perdurant* we shall an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the *perdurant*

This is the second major part of this case study. The first major part is Part E.6. In a number of subsections we shall cover

- Sect. E.12: the recursive *definition*
 - ⊗ of the *compilation of structures*, and *composite parts*
 - ⊗ into *translator* invocations;
- Sect. E.13: the *declaration of channels*; and
- Sect. E.14: the *definition of the translation of atomic parts* into
 - ⊗ *behaviour signatures* and
 - ⊗ *behaviour definition bodies*.

We observe that the term *train* can have the following “meanings”: the *train*, as an *endurant*, parked at the railway station platform, i.e., as a *composite part*; the *train*, as a *perdurant*, as it “speeds” down the railway track, i.e., as a *behaviour*; the *train*, as an *attribute*, say in a time-table.

This observation motivates that we “magically”, as it were, introduce a **COMPILER** function, cf. [70, Sect. 4] We shall refer to this “magic” as a **transcendental interpretation**²⁰ of *parts as behaviours*.

E.12 The Structure COMPILERS

E.12.1 A UNIVERSE OF DISCOURSE COMPILER

In this section, i.e., all of Sect. E.12.1, we omit complete typing of behaviours.

1071. The universe of discourse, *uod*, **COMPILES** and **TRANSLATES** into the of its four elements:
- a. the translation of the atomic clock, see Item E.14.1 [Page 383],
 - b. the translation of the atomic urban space, see Item E.14.2 [Page 383],
 - c. the compilation of the analyser structure, see Item E.12.2 [Page 377],
 - d. the compilation of planner structure. see Item E.12.3 [Page 378],

value

```

1071  COMPILER.UoD(uod) ≡
1071a  TRANSLATE_CLK(clk),
1071b  TRANSLATE_TUS(tus),
1071c  COMPILER_AA(obs_AA(uod)),
1071d  COMPILER_PA(obs_PA(uod))

```

The **COMPILER** apply to, as here, *structures*, or composite parts. The **TRANSLATOR** apply to atomic parts. In this section, i.e., Sect. E.12.1, we will explain the obvious meaning of these functions: we will not formalise their type, and we will make some obvious short-cuts.

²⁰ By *transcendental* we shall mean:

- (1) BEYOND THE CONTINGENT AND ACCIDENTAL IN HUMAN EXPERIENCE, BUT NOT BEYOND ALL HUMAN KNOWLEDGE,
- (2) PERTAINING TO, BASED UPON, OR CONCERNED WITH A PRIORI ELEMENTS IN EXPERIENCE, WHICH CONDITION HUMAN KNOWLEDGE [(2–3) <http://www.dictionary.com/browse/transcendental>],
- (3) OF OR RELATING TO EXPERIENCE AS DETERMINED BY THE MIND’S MAKEUP [(3) Merriam Webster, <https://www.merriam-webster.com/dictionary/transcendental>]

E.12.2 The ANALYSER STRUCTURE COMPILER

1072. Compiling the analyser structure results in an RSL-**Text** which expresses the separate
- translation of each of its n analysers, see Item E.14.3 [Page 385], and
 - the translation of the analysis depository, see Item E.14.4 [Page 386].

1072 **COMPILE_AA**(aa) \equiv
 1072a $\{ \text{TRANSLATE_A}_{ann_i}(\text{obs_A}_{ann_i}(\text{aa})) \mid i:[1..n] \}$,
 1072b **TRANSLATE_AD**(obs_AD(aa))

E.12.3 The PLANNER STRUCTURE COMPILER

1073. The *planner structure*, pa:PA, compiles into four elements:
- the compilation of the *master planner structure*, see Item E.12.3 [Page 378],
 - the translation of the *derived server index generator*, see Item E.14.5 [Page 387],
 - the translation of the *plan repository*, see Item E.14.6 [Page 388], and
 - the compilation of the *derived server structure*, see Item E.12.3 [Page 378].

1073 **COMPILE_PA**(pa) \equiv
 1073a **COMPILE_MPA**(obs_MPA(pa)),
 1073b **TRANSLATE_DPXG**(obs_DPXG(pa)),
 1073c **TRANSLATE_PR**(obs_PR(pa)),
 1073d **COMPILE_DPA**(obs_DPA(pa))

The MASTER PLANNER STRUCTURE COMPILER

1074. Compiling the *master planner structure* results in an RSL-**Text** which expresses the separate translations of the
- atomic *master planner server*, see Item E.14.7 [Page 389] and
 - atomic *master planner*, see Item E.14.8 [Page 391].

1074 **COMPILE_MPA**(mpa) \equiv
 1074a **TRANSLATE_MPS**(obs_MPS(mpa)),
 1074b **TRANSLATE_MP**(obs_MP(mpa))

The DERIVED PLANNER STRUCTURE COMPILER

1075. The compilation of the *derived planner structure* results in some RSL-**Text** which expresses the set of separate compilations of each of the *derived planner pair structures*, see Item E.12.3 [Page 378].

1075 **COMPILE_DPA**(dpa) $\equiv \{ \text{COMPILE}(\text{obs_DPC}_{nm_j}(\text{pa})) \mid j:[1..p] \}$

The DERIVED PLANNER PAIR STRUCTURE COMPILER

1076. The compilation of the *derived planner pair structure* results in some RSL-**Text** which expresses
- the results of translating the *derived planner server*, see Item E.14.9 [Page 393] and
 - the results of translating the *derived planner*, see Item E.14.10 [Page 394].

1076 **COMPILE_DPC** $_{nm_j}(\text{dpc}_{nm_j})$, $i:[1..p]$ \equiv
 1076a **TRANSLATE_DPS** $_{nm_j}(\text{obs_DPS}_{nm_j}(\text{dpc}_{nm_j}))$,
 1076b **TRANSLATE_DP** $_{nm_j}(\text{obs_DP}_{nm_j}(\text{dpc}_{nm_j}))$

E.13 Channel Analysis and Channel Declarations

The *transcendental interpretation* of parts as behaviours implies existence of means of communication & synchronisation of between and of these behaviours. We refer to Fig. E.7 [Page 379] for a summary of the channels of the urban space analysis and urban planning system.

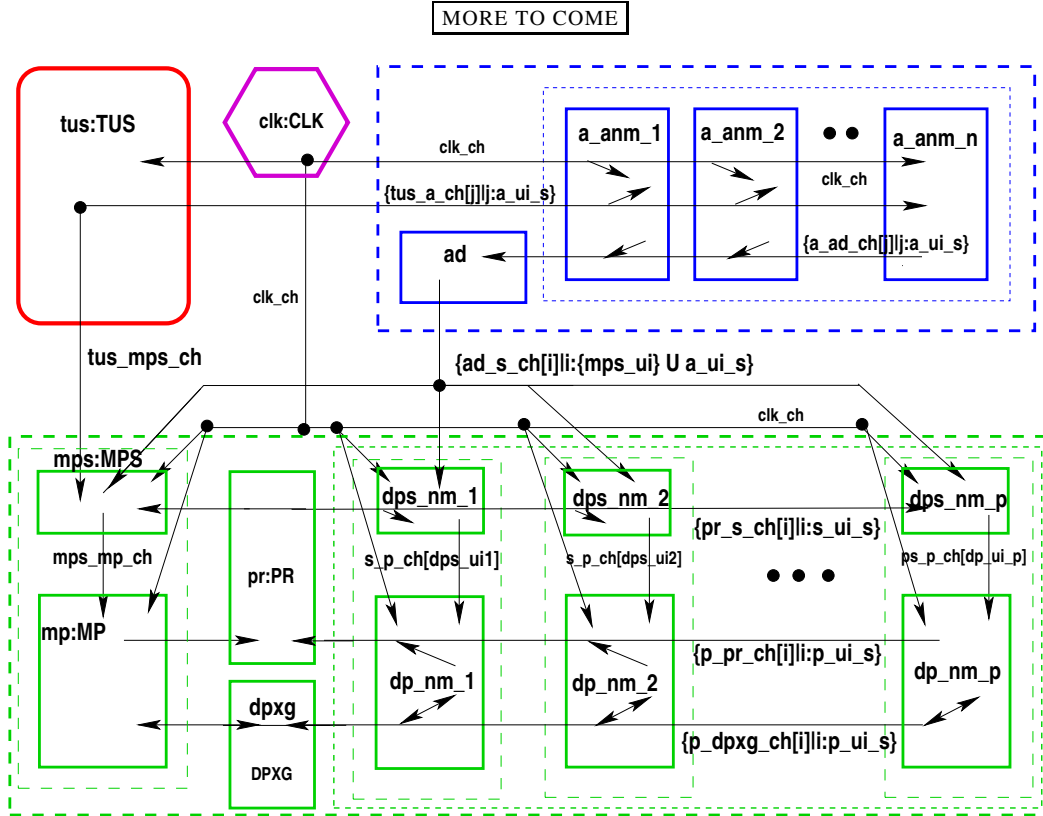


Fig. E.7. The Urban Space and Analysis Channels and Behaviours

E.13.1 The `clk_ch` Channel

The purpose of the `clk_ch` channel is, for the clock, to propagate Time to such entities who inquire. We refer to Sects. E.9.1 [Page 364], E.9.2 [Page 364], E.9.3 [Page 365], E.9.5 [Page 365], E.9.6 [Page 365], E.9.7 [Page 366] and E.9.8 [Page 366] for the mereologies that help determine the indices for the `clk_ch` channel.

- 1077. There is declared a (single) channel `clk_ch`
- 1078. whose messages are of type `CLK_MSG` (for Time).

The `clk_ch` is single. There is no need for enquirers to provide their identification. The clock “freely” dispenses of “its” time.

type

1077 `CLK_MSG = T`

channel

1078 `clk_ch:CLK_MSG`

E.13.2 The `tus_a_ch` Channel

The purpose of the `tus_a_ch` channel is, for the the urban space, to propagate urban space attributes to analysers. We refer to Sects. E.9.2 and E.9.3 for the mereologies that help determine the indices for the `tus_a_ch` channel.

1079. There is declared an array channel `tus_a_ch` whose messages are of

1080. type `TUS_MSG` (for a *time stamped* aggregate of *urban space attributes*, `TUSm`, cf. Item 1048 [Page 372]).

type

1080 `TUS_MSG = T × TUSm`

channel

1079 `{tus_a_ch[a-ui]:TUS_MSG|a-ui:A-UI•a-ui ∈ a_{uis} }`

The `tus_a_ch` channel is to offer urban space information to all analysers. Hence it is an array channel over indices `ANms`, cf. Item 954 [Page 357].

E.13.3 The `tus_mps_ch` Channel

The purpose of the `tus_mps_ch` channel is, for the the urban space, to propagate urban space attributes to the master planner server. We refer to Sects. E.9.2 and E.9.5 for the mereologies that help determine the indices for the `tus_mps_ch` channel.

1081. There is declared a channel `tus_mps_ch` whose messages are of

1080 type `TUS_MSG` (for a *time stamped* aggregate of *urban space attributes*, `TUSm`, cf. Item 1048 [Page 372]).

type

1080 `TUS_MSG = T × TUSm`

channel

1081 `tus_mps_ch:TUS_MSG`

The `tus_s_ch` channel is to offer urban space information to just the master server. Hence it is a single channel.

E.13.4 The `a_ad_ch` Channel

The purpose of the `a_ad_ch` channel is, for analysers to propagate analysis results to the analysis depository. We refer to Sects. E.9.3 and E.9.4 for the mereologies that help determine the indices for the `a_ad_ch` channel.

1082. There is declared a channel `a_ad_ch` whose *time stamped* messages are of

1083. type `A_MSG` (for *analysis message*).

type

1083 `A_MSGanmi = (s_T:T × s_A:Analysisanmi), i:[1:n]`

1083 `A_MSG = A_MSGanm1|A_MSGanm2|...|A_MSGanmn`

channel

1082 `{a_ad_ch[a-ui]:A_MSG|a-ui:A-UI•a-ui ∈ a_{uis} }`

E.13.5 The `ad_s_ch` Channel

The purpose of the `ad_s_ch` channel is, for the analysis depository to propagate histories of analysis results to the server. We refer to Sects. E.9.4, E.9.5 and E.9.7 for the mereologies that help determine the indices for the `ad_s_ch` channel.

1084. There is declared a channel `ad_s_ch` whose messages are of

1085. type `AD_MSG` (defined as `A_Hist` for a *histories of analyses*), see Item 1051 [Page 373].

type

1085 `AD_MSG = A_Hist`

channel

1084 $\{\text{ad_s_ch}[s_ui]|s_ui:(\text{MPS_UI}|\text{DPS_UI})\cdot s_ui \in \{mps_ui\} \cup \{dps_uis\}\}:\text{AD_MSG}$

The `ad_s_ch` channel is to offer urban space information to the *master* and *derived servers*. Hence it is an array channel.

E.13.6 The `mps_mp_ch` Channel

The purpose of the `mps_mp_ch` channel is for the master server to propagate comprehensive master planner input to the master planner. We refer to Sects. E.9.5 and E.9.6 for the mereologies that help determine the indices for the `mps_mp_ch` channel.

1086. There is declared a channel `mps_mp_ch` whose messages are of

1087. type `MPS_MSG` which are quadruplets of time stamped urban space information, `TUS_MSG`, see Item 1080 [Page 380], analysis histories, `A_Hist`, see Item 1085 [Page 381], *master planner auxiliary information*, `mAUX`, and *master plan requirements*, `mREQ`.

type

1087 `MPS_MSG = TUS_MSG × AD_MSG × mAUX × mREQ`

channel

1086 `mps_mp_ch:MPS_MSG`

The `mps_mp_ch` channel is to offer `MPS_MSG` information to just the *master server*. Hence it is a single channel.

E.13.7 The `p_pr_ch` Channel

The purpose of the `p_pr_ch` channel is, for master and derived planners to deposit and retrieve master and derived plans to the plan repository. We refer to Sects. E.9.6 and E.9.10 for the mereologies that help determine the indices for the `p_pr_ch` channel.

1088. There is declared a channel `p_pr_ch` whose messages are of

1089. type `PLAN_MSG` – for *time stamped master plans*.

type

1089 `PLAN_MSG = T × PLANS`

channel

1088 $\{\text{p_pr_ch}[p_ui]:\text{PLAN_MSG}|p_ui:(\text{MP_UI}|\text{DP_UI})\cdot p_ui \in p_uis\}$

The `p_pr_ch` channel is to offer comprehensive records of all current plans to all the the *planners*. Hence it is an array channel.

E.13.8 The p_dpxg_ch Channel

The purpose of the p_dpxg_ch channel is, for planners to request and obtain derived planner index names of, respectively from the derived planner index generator. We refer to Sects. E.9.6 and E.9.9 for the mereologies that help determine the indices for the mp_dpxg_ch channel.

1090. There is declared a channel p_dpxg_ch whose messages are of

1091. type DPXG_MSG. DPXG_MSG messages are

- a. either *request* from the *planner* to the *index generator* to provide zero, one or more of an indicated set of *derived planner names*,
- b. or to accept such a (*response*) set from the *index generator*.

type

1091 DPXG_MSG = DPXG_Req | DPXG_Rsp

1091a DPXG_Req :: DNm-set

1091b DPXG_Rsp :: DNm-set

channel

1090 {p_dpxg_ch[ui]:DPXG_MSG|ui:(MP_UI|DP_UI)•ui ∈ p_{uis}}

E.13.9 The pr_s_ch Channel

The purpose of the pr_s_ch channel is, for the plan repository to provide master and derived plans to the derived planner servers. We refer to Sects. E.9.10 and E.9.7 for the mereologies that help determine the indices for the pr_dps_ch channel.

1092. There is declared a channel pr_dps_ch whose messages are of

1093. type PR_MSGd, defined as PLAp, cf. Item 1068 [Page 375].

type

1093 PR_MSG = PLANS

channel

1092 {pr_s_ch[ui]:PR_MSGd|ui:(MPS_UI|DPS_UI)•ui ∈ s_{uis}}

E.13.10 The dps_dp_ch Channel

The purpose of the dps_dp_ch channel is, for derived planner servers to provide input to the derived planners. We refer to Sects. E.9.7 and E.9.8 for the mereologies that help determine the indices for the dps_dp_ch channel.

1094. There is declared a channel dps_dp_ch[ui_nm_j], one for each *derived planner* pair.

1095. The channel messages are of type DPS_MSG_{nm_j}. These DPS_MSG_{nm_i} messages are quadruplets of *analysis* aggregates, AD_MSG, *urban plan* aggregates, PLANS, *derived planner auxiliary information*, dAUX_{nm_j}, and *derived plan requirements*, dAUX_{nm_j}.

type

1095 DPS_MSG_{nm_j} = AD_MSG × PLANS × dAUX_{nm_j} × dREQ_{nm_j}, j: [1..p]

channel

1094 {dps_dp_ch[ui]:DPS_MSG_{nm_j}|ui:DPS_UI•ui ∈ dps_{uis}}

E.14 The Atomic Part TRANSLATORS

E.14.1 The CLOCK TRANSLATOR

We refer to Sect. E.10.1 for the attributes that play a rôle in determining the clock signature.

The TRANSLATE_CLK Function

1096. The `TRANSLATE_CLK(clk)` results in three text elements:

- a. the **value** keyword,
- b. the *signature* of the clock definition,
- c. and the *body* of that definition.

The clock signature contains the *unique identifier* of the clock; the *mereology* of the clock, cf. Item E.9.1 [Page 364]; and the *attributes* of the clock, in some form or another: the programmable time attribute and the channel over which the clock offers the time.

value

```
1096 TRANSLATE_CLK(clk) ≡
1096a " value
1096b clock: T → out clk_ch Unit
1096c clock(uid_CLK(clk),mereo_CLK(clk))(attr_T(clk)) ≡ ... "
```

The clock Behaviour

The purpose of the clock is to show the time. The “players” that need to know the time are: the urban space when informing requestors of aggregates of urban space attributes, the analysers when submitting analyses to the analysis depository, the planners when submitting plans to the plan repository.

1097. We see the clock as a behaviour.
1098. It takes a programmable input, the *current* time, *t*.
1099. It repeatedly emits the some *next* time on channel `clk_ch`.
1100. Each iteration of the clock it non-deterministically, internally increments the *current* time by either nothing or an infinitesimally small time interval δt_i , cf. Item 1022 [Page 367].
1101. In each iteration of the clock it either offers this *next* time, or skips doing so;
1102. whereupon the clock resumes being the clock albeit with the new, i.e., *next* time.

value

```
1099  $\delta t_i: T_I = \dots$  cf. Item 1022 [Page 367]
1097 clock: T → out clk_ch Unit
1098 clock(uid_clk,mereo_clk)(t) ≡
1100 let  $t' = (t + \delta t_i) \sqcap t$  in
1101 skip  $\sqcap$  clk_ch! $t'$ ;
1102 clock(uid_clk,mereo_clk)( $t'$ ) end
1102 pre: uid_clk =  $clk_{ui} \wedge$ 
1102 mereo_clk = ( $tus_{ui}, a_{ui}s, mps_{ui}, mp_{ui}, dps_{ui}s, dp_{ui}s$ )
```

E.14.2 The URBAN SPACE TRANSLATOR

We refer to Sect. E.10.2 for the attributes that play a rôle in determining the urban space signature.

The TRANSLATE_TUS Function

1103. The `TRANSLATE_TUS(tus)` results in three text elements:
- the **value** keyword
 - the *signature* of the `urb_spa` definition,
 - and the *body* of that definition.

The urban space signature contains the *unique identifier* of the urban space, the *mereology* of the urban space, cf. Item E.9.2 [Page 364], the static point space *attribute*.

value

```
1103 TRANSLATE_TUS(tus) ≡
1103a " value
1103b     urb_spa: TUS_UI × TUS_Mer → Pts →
1103b     out ... Unit
1103c     urb_spa(uid_TUS(tus),mereo_TUS(tus))(attr_Pts(tus)) ≡ ... "
```

We shall detail the `urb_spa` signature and the `urb_spa` body next.

The urb_spa Behaviour

The urban space can be seen as a behaviour. It is “visualized” as the rounded edge box to the left in Fig. E.8 [Page 384]. It is a “prefix” of Fig. E.2 [Page 351]. In this section we shall refer to many other elements of our evolving specification. To grasp the seeming complexity of the urban space, its analyses and its urban planning functions, we refer to Fig. E.2 [Page 351].

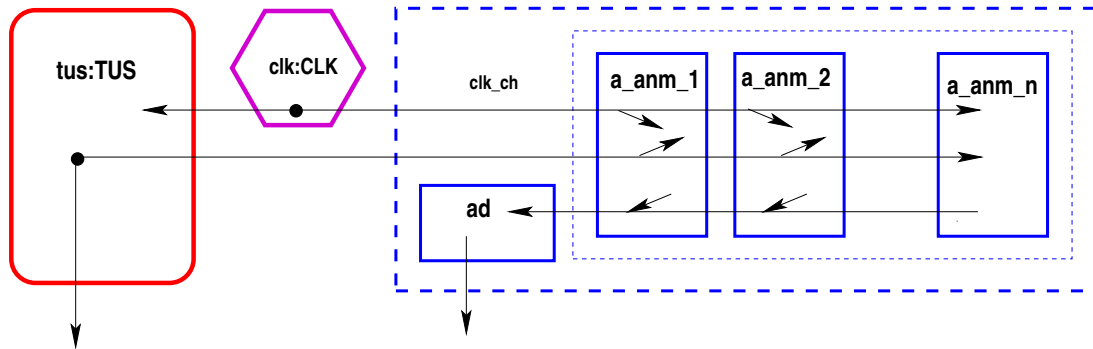


Fig. E.8. The Urban Space and Analysis Behaviours

1104. To every observable part, like `tus:TUS`, there corresponds a behaviour, in this case, the `urb_spa`.
1105. The `urb_spa` behaviour has, for this report, just one static attribute, the point space, `Pts`.
1106. The `urb_spa` behaviour has the following biddable and programmable attributes, the Cadastral, the Law and the SocioEconomic attributes. The biddable and programmable attributes “translate” into behaviour parameters.
1107. The `urb_spa` behaviour has the following dynamic, non-biddable, non-programmable attributes, the GeoDetic, GeoTechnic and the Meterological attributes. The non-biddable, non-programmable dynamic attributes “translate”, in the conversion from parts to behaviours, to input channels etc.

the `urb_spa` behaviour offers its attributes, upon demand,

1108. to a urban space analysis behaviours, tus_ana_i and one master urban server.

1109. The urb_spa otherwise behaves as follows:

- a. it repeatedly “assembles” a tuple, tus , of all attributes;
- b. then it external non-deterministically either offers the tus tuple
- c. to either any of the urban space analysis behaviours,
- d. or to the master urban planning behaviour;
- e. in these cases it resumes being the urb_spa behaviour;
- f. or internal-non-deterministically chooses to
- g. update the law, the cadastral, and the socio-economic attributes;
- h. whereupon it resumes being the urb_spa behaviour.

channel

1107 attr_Pts_ch:Pts, attr_GeoD_ch:GeoD, attr_GeoT_ch:GeoT, attr_Met_ch:Met

1108 tus_mps_ch:TUSm

1108 {tus_a_ch[ai]|ai ∈ a_{uis} }:TUSm

value

1104 urb_spa: TUS_UI × TUS_Mer →

1105 Pts →

1106 (Cada × Law × Soc_Eco × ...) →

1107 in attr_Pts_ch, attr_GeoD_ch, attr_GeoT_ch, attr_Met_ch →

1108 out tus_mps_ch, {tus_ana_ch[ai]|ai ∈ [a_1...a_a]} → **Unit**

1109 urb_spa(pts)(pro) ≡

1109a let geo = ["pts" ↦ attr_Pts_ch?, "ged" ↦ attr_GeoD_ch?, "cad" ↦ cada, "get" ↦ attr_geoT_ch?,

1109a "met" ↦ attr_Met_ch?, "law" ↦ law, "eco" ↦ eco, ...] in

1109c (([] {tus_a_ch[ai]|geo|ai ∈ a_{uis} }

1109b []

1109d tus_mps_ch!geo);

1109e urb_spa(pts)(pro) end

1109f []

1109g let pro':(Cada × Law × Soc_Eco × ...)·fit_pro(pro,pro') in

1109h urb_spa(pts)(pro') end

1109g fit_pro: (Cada × Law × Soc_Eco × ...) × (Cada × Law × Soc_Eco × ...) → **Bool**

We leave the *fitness predicate* fit_pro further undefined. It is intended to ensure that the biddable and programmable attributes evolve in a commensurate manner.

E.14.3 The ANALYSER $_{ann_j}$, $i:[1:n]$ TRANSLATOR

We refer to Sect. E.10.4 for the attributes that play a rôle in determining the analyser signature.

The TRANSLATE $_{A_{ann_j}}$ Function

1110. The $TRANSLATE_{A_{ann_j}}(a_{ann_j})$ results in three text elements:

- a. the **value** keyword,
- b. the *signature* of the analyser $_{ann_j}$ definition,
- c. and the *body* of that definition.

The analyser $_{ann_j}$ signature contains the *unique identifier* of the analyser, the *mereology* of the analyser, cf. Item E.9.3 [Page 365], and the *attributes*, here just the programmable attribute of the most recent analysis a_{ann_j} performed by the analyser $_{ann_j}$.

```

type
1110 Analysis = Analysisnm1 | Analysisnm2 | ... | Analysisnmn
value
1110 TRANSLATE_Anmi(anmi):
1110 " value
1110     analysernmi: (uid_A × mereo_A) →
1110     Analysisnmi →
1110     in tus_a_ch[uid_A(anmi)]
1110     out a_ad_ch[uid_A(anmi)]
1110     analyseruij(uid_A(anmi), mereo_A(anmi))(ananmi) ≡ ... "

```

The analyser_{ui_j} Behaviour

Analyses, or various kinds, of the urban space, is an important prerequisite for urban planning. We therefore introduce a number, n , of urban space analysis behaviours, analysis_{anm_i} (for anm_i in the set {anm₁, ..., anm_a}). The indexing designates that each analysis_{anm_i} caters for a distinct kind of urban space analysis, each analysis with respect to, i.e., across existing urban areas: ..., (a_i) traffic statistics, (a_j) income distribution, ..., (a_k) health statistics, (a_l) power consumption, ..., (a_a) We shall model, by an indexed set of behaviours, ana_i, the urban [space] analyses that are an indispensable prerequisite for urban planning.

- 1111. Urban [space] analyser, tus_ana_i, for a_i ∈ [a₁...a_a], performs analysis of an urban space whose attributes, except for its point set, it obtains from that urban space – via channel tus_ana_ch and
- 1112. offers analysis results to the mp_beh and the n derived behaviours.
- 1113. Urban analyser, ana_{a_i}, otherwise behaves as follows:
 - a. The analyser obtains, from the urban space, its most recent set of attributes.
 - b. The analyser then proceeds to perform the specific analysis as “determined” by its index a_i.
 - c. The result, tus_ana_{a_i}, is communicated whichever urban, the master or the derived, planning behaviour inquires.
 - d. Whereupon the analyser resumes being the analyser, improving and/or extending its analysis.

```

type
1110 Analysis = Analysisanm1 | Analysisanm2 | ... | Analysisanmn
value
1113 analysernmi(a_ui, a_mer)(analysisnmi) ≡
1113a let tsm = tus_a_ch[a_ui] ? in
1113b let analysis'nmi = perform_analysisnmi(tsm)(analysis) in
1113c [] a_ad_ch[a_ui] ! (clk_ck?, analysis'nmi) ;
1113d analyseri(a_ui, a_mer)(analysis'nmi) end end

1113b perform_analysisanmi: TUSm → Analysisanmi → Analysisanmi
1113b perform_analysisanmi(tsm)(analysisanmi) ≡ ...

```

E.14.4 The ANALYSIS DEPOSITORY TRANSLATOR

We refer to Sect. E.10.5 for the attributes that play a rôle in determining the analysis depository signature.

The TRANSLATE_AD Function

- 1114. The TRANSLATE_AD(ad) results in three text elements:
 - a. the **value** keyword

- b. the *signature* of the `ana_dep` definition,
- c. and the *body* of that definition.

The `ana_dep` signature essentially contains the *unique identifier* of the analyser, the *mereology* of the analyser, cf. Item E.9.4 [Page 365], and the *attributes*, in one form or another: the programmable attribute, `a_hist`, see Item 1051 [Page 373], the channels over which `ana_dep` either accepts time stamped *analyses*, $\text{Analysis}_{a_{ui}}$, from analyser_{ann_i} , or offers `a_hists` to either the *master planner server* or the *derived planner servers*.

```

value
1114 TRANSLATE_AD(ad)  $\equiv$ 
1114a   " value
1114b     ana_dep: (A_UI  $\times$  A_Mer)  $\rightarrow$  AHist  $\rightarrow$ 
1114b       in {a_ad_ch[i]|i:A_UI•i  $\in$   $a_{uis}$ }
1114b       out {ad_s_ch[i]|i:A_UI•i  $\in$   $s_{uis}$ } Unit
1114c     ana_dep(ui_A(ad),mereo_A(ad))(attr_AHist(ad))  $\equiv$  ... "
```

The `ana_dep` Behaviour

The definition of the *analysis depository* is as follows.

- 1115. The behaviour of `ana_dep` is as follows: non-deterministically, externally (\square), `ana_dep`
- 1116. either (\square , line 1118) offers to accept a time stamped analysis *from some* analyser ($\square\{ \dots | \dots \}$),
 - a. receiving such an analyses it “updates” its history,
 - b. and resumes being the `ana_dep` behaviour with that updated history;
- 1117. or offers the analysis history *to the* master planner server and resumes being the `ana_dep` behaviour;
- 1118. or offers the analysis history
 - a. *to whichever* ($\square\{ \dots | \dots \}$) planner server offers to accept a history
 - b. and resumes being the `ana_dep` behaviour with that updated history.

```

value
1115 ana_dep(a_ui,a_mer)(ahist)  $\equiv$ 
1116    $\square$  { (let ana = a_ad_ch[i] ? in
1116a     let ahist' = ahist†[i→(ana)^(ahist(i))] in
1116b     ana_dep(a_ui,a_mer)(ahist') end end
1116b   | i:A_UI•i  $\in$   $a_{uis}$  }
1117    $\square$  (ad_mps_ch!ahist ; ana_dep(a_ui,a_mer)(ahist))
1118    $\square$ 
1118a   ({ ad_s_ch[j]!ahist
1118a     | j:(MPS_UI|DPS_UI)•j  $\in$   $s_{uis}$ };
1118b     ana_dep(a_ui,a_mer)(ahist))
```

E.14.5 The DERIVED PLANNER INDEX GENERATOR TRANSLATOR

We refer to Sect. E.10.10 for the attributes that play a rôle in determining the derived planner index generator signature.

The TRANSLATE_DPXG(*dpxg*) Function

1119. The TRANSLATE_DPXG(*dpxg*) results in three text elements:
- the **value** keyword
 - the *signature* of the *dpxg* behaviour definition,
 - and the *body* of that definition.

The signature of the *dpxg* behaviour definition has many elements: the *unique identifier* of the *dpxg* behaviour, the *mereology* of the *dpxg* behaviour, cf. Item E.9.9 [Page 367], and the *attributes* in some form or another: the *unique identifier*, the *mereology*, and the *attributes*, in some form or another: the programmable attribute All_DPUIs, cf. Item 1065 [Page 375], the programmable attribute Used_DPUIs, cf. Item 1066 [Page 375], the *mp_dpxg_ch* input/output channel, and the *dp_dpxg_ch* input/output array channel.

value

```

1119 TRANSLATE_DPXG(dpxg) ≡
1119a " value
1119b   dpxg_beh: (DPXG_UI × DPXG_Mer) →
1119b   (All_DPUIs × Used_DPUIs) →
1119b   in, out {p_dpxg_ch[i]|i: (MP_UI|DP_UI)•i ∈ p_uis} Unit
1119c   dpxg_beh(uid_DPXG(dpxg), mergeo_DPXG(dpxg))(all_dpui, used_dpui) ≡ ... "
```

The *dpxg* Behaviour

1120. The index generator otherwise behaves as follows:
- It non-deterministically, externally, offers to accept requests from any planner, whether master or server. The request suggests the names, req, of some derived planners.
 - The index generator then selects a suitable subset, *sel_dpui*, of these suggested derived planners from those that are yet to be started.
 - It then offers these to the requesting planner.
 - Finally the index generator resumes being an index generator, now with an updated *used_dpui* programmable attribute.

value

```

1120 dpxg: (DPXG_UI × DPXG_Mer) → (All_DPUIs × Used_DPUIs) →
1120   in, out mp_dpxg_ch,
1120   {p_dpxg_ch[j]|j: (MP_UI|DP_UI)•j ∈ {p_uis}} Unit
1120   dpxg(dpxg_ui, dpxg_mer)(all_dpui, used_dpui) ≡
1120a   □ { let req = p_dpxg_c[j] ? in
1120b     let sel_dpui = all_dpui \ used_dpui • sel_dpui ⊆ req_dpui in
1120c     dp_dpxg_ch[j] ! sel_dpui ;
1120d     dpxg(dpxg_ui, dpxg_mer)(all_dpui, used_dpui ∪ sel_dpui) end end
1120   | j: (MP_UI|DP_UI)•j ∈ p_uis }
```

E.14.6 The PLAN REPOSITORY TRANSLATOR

We refer to Sect. E.10.11 for the attributes that play a rôle in determining the plan repository signature.

The TRANSLATE_PR Function

1121. The `TRANSLATE_PR(pr)` results in three text elements:
- the **value** keyword,
 - the *signature* of the plan repository definition,
 - and the *body* of that definition.

The plan repository signature contains the *unique identifier* of the plan repository, the *mereology* of the plan repository, cf. Item E.9.10 [Page 367], and the *attributes*: the *programmable* plans, cf. 1068 [Page 375], and the *input/out channel* `p_pr_ch`.

value

```

1121 TRANSLATE_PR(pr) ≡
1121a " value
1121b   plan_rep: PLANS →
1121b     in {p_pr_ch[i]|i:(MP_UI|DP_UI)•i∈puis}
1121b     out {s_pr_ch[i]|i:(MP_UI|DP_UI)•i∈suis} Unit
1121c   plan_rep(plans)(attr_AllDPUs(pr),attr_UsedDPUs(pr)) ≡ ... "
```

The plan_rep Behaviour

1122. The plan repository behaviour is otherwise as follows:
- The plan repository non-deterministically, externally chooses between
 - offering to accept time-stamped plans from a planner, *p_{ui}*, either the master planner or anyone of the derived planners,
 - from whichever planner so offers,
 - inserting these plans appropriately, i.e., at *p_{ui}*, as the new head of the list of “there”,
 - and then resuming being the plan repository behaviour appropriately updating its programmable attribute;
 - or
 - offering to provide a full copy of its plan repository map
 - to whichever server requests so,
 - and then resuming being the plan repository behaviour.

value

```

1122 plan_rep(pr_ui,ps_uis)(plans) ≡
1122(a)i   [] { let (t,plan) = p_pr_ch[i] ? in assert: i ∈ dom plans
1122(a)iii  let plans' = plans † [i→⟨(t,plan)⟩^plans(i)] in
1122(a)iv   plan_rep(pr_ui,ps_uis)(plans') end end
1122(a)ii  | i:(MP_UI|DP_UI)•i∈puis }
1122b     []
1122(b)i   [] { s_pr_ch[i] ! plans ; assert: i ∈ dom plans
1122(b)iii  plan_rep(pr_ui,ps_uis)(plans)
1122(b)ii  | i:(MP_UI|DP_UI)•i∈puis }
```

E.14.7 The MASTER SERVER TRANSLATOR

We refer to Sect. E.10.6 for the attributes that play a rôle in determining the master server signature.

The TRANSLATE_MPS Function

1123. The `TRANSLATE_MPS(mps)` results in three text elements:
- the **value** keyword,
 - the *signature* of the `master_server` definition,
 - and the *body* of that definition.

The `master_server` signature contains the *unique identifier* of the master server, the *mereology* of the master server, cf. Item E.9.5 [Page 365], and the *dynamic attributes* of the master server: the most recently, previously produced *auxiliary* information, the most recently, previously produced *plan requirements* information, the clock channel, the urban space channel, the analysis depository channel, and the master planner channel.

value

```

1123 TRANSLATE_MPS(mps) ≡
1123a " value
1123b     master_server: (mAUX × mREQ) →
1123b         in clk_ch, tus_m_ch, ad_s_ch[uid_MPS(mps)]
1123b         out mps_mp_ch Unit
1123c     master_server(uid_MPS(mps), mereo_MPS(mps))(attr_mAUX(mps), attr_mREQ(mps)) ≡ ... "
```

The master_server Behaviour

1124. The `master_server` obtains time from the clock, see Item 1125c, information from the urban space, and the most recent analysis history, assembles these together with “locally produced”
- auxiliary* planner information and
 - plan requirements*
- as input, `MP_ARG`, to the master planner.
1125. The master server otherwise behaves as follows:
- it obtains latest urban space information and latest analysis history, and
 - then produces auxiliary planning and plan requirements commensurate, i.e., fit, with the most recently, i.e., previously produced such information;
 - it then offers a time stamped compound of these kinds of information to the master planner,
 - whereupon the master server resumes being the master server, albeit with updated programmable attributes.

type

```

1124a mAUX
1124b mREQ
1124 mARG = (T × ((mAUX × mREQ) × (TUSm × AHist)))
```

value

```

1125 master_server(uid, mereo)(aux, req) ≡
1125a let tusm = tus_m_ch ? , ahist = ad_s_ch[mps_ui] ? ,
1125b     maux:mAUX, mreq:mREQ • fit_AuxReq((aux, req), (maux, mreq)) in
1125c s_p_ch[uid] ! (clk_ch?, ((maux, mreq), (tusm, ahist))) ;
1125d master_server(uid, mereo)(maux, mreq)
1125 end
```

```
1125b fitAuxReq: (mAUX × mREQ) × (mAUX × mREQ) → Bool
```

```
1125b fitAuxReq((aux, req), (maux, mreq)) ≡ ...
```

E.14.8 The MASTER PLANNER TRANSLATOR

We refer to Sect. E.10.7 for the attributes that play a rôle in determining the master planner signature.

The TRANSLATE_MP Function

1126. The `TRANSLATE_MP(mp)` results in three text elements:
- the **value** keyword,
 - the *signature* of the `master_planner` definition,
 - and the *body* of that definition.

The `master_planner` signature contains the *unique identifier* of the master planner, the *mereology* of the master planner, cf. Item E.9.6 [Page 365], and the *attributes* of the master planner: the script, cf. Sect. E.10.3 [Page 373] and Item 1049 [Page 373], a set of script pointers, cf. Item 1056 [Page 374], a set of analyser names, cf. Item 1057 [Page 374], a set of planner identifiers, cf. Item 1058 [Page 374], and the channels as implied by the master planner mereology.

value

```

1126 TRANSLATE_MP(mp) ≡
1126a " value
1126b     master_planner: Mmpui:P_UI×MP_Mer×(Script×ANms×DPUIs) →
1126b     Script_Pts →
1126b     in     clk_ch, mps_mp_ch, ad_ps_ch[mpui]
1126b     out    p_pr_ch[mpui]
1126b     in,out p_dpxg_ch[mpui] Unit
1126c     master_planner(uid_MP(mp),mereo_MP(mp),
1126c     (attr_Script(mp),attr_ANms(mp),attr_DPUIs(mp)))(attr_Script_Ptrs(mp)) ≡ ... "
```

The Master urban_planning Function

1127. The core of the `master_planner` behaviour is the `master_urban_planning` function.
1128. It takes as arguments: the script, a set of analyser names, a set of derived planner identifiers, a set of script pointers, and the time-stamped master planner argument, cf. Item 1124 [Page 390];
1129. and delivers, i.e., yields, a set of “remaining” derived planner identifiers, an updated set of script pointers, and a master result: `M_RES`, i.e., a master plan, `mp:M_PLAN` together with the time stamped master argument from which the plan was constructed.
1130. The `master_urban_planning` function is not defined by other than a predicate:
- the “remaining” derived planner identifiers is a subset of the arguments derived planner identifiers;
 - the “resulting” master argument is the same as the input master argument, i.e., it is “carried forward”;
 - the arguments: the script, the analyser names, the derived planner identifiers, the set of script pointers, the time-stamped master planner argument, and the result plan otherwise satisfies a predicate $\mathcal{P}(\text{script}, \text{anms}, \text{dp_{uis}}, \text{ptrs}, \text{marg})(\text{mplan})$ expressing that the result `mplan` is an appropriate plan in view of the other arguments.

type

```

1129 M_PLAN
1129 M_RES = M_PLAN × DPUI-set × M_ARG
```

value

```

1128 master_urban_planning:
1128     Script × ANm-set × DP_UI-set × Script_Ptr-set × M_ARG
```

```

1129      → (DP_UI-set × Script_Ptr-set) × M_RES
1127 master_urban_planning(script,anms,dpuis,ptrs,marg)
1130a   as ((dpuis',ptrs'),(mplan,marg'))
1130a   dpuis' ⊆ dpuis
1130b   ∧ marg' = marg
1130c   ∧  $\mathcal{P}$ (script,anms,dpuis,ptrs,marg)(mplan)
1127  $\mathcal{P}$ : ((Script×ANM-set×DP_UI-set×Script_Ptr-set×M_ARG×MPLAN×Script_Ptr-set)
1127      × (DP_UI-set×Script_Ptr-set×M_ARG×MPLAN)) → Bool
1127  $\mathcal{P}$ ((script,anms,dpuis,ptrs,marg,mplan,ptrs),(dpuis',ptrs',marg,mplan)) ≡ ...

```

The master_planner Behaviour

1131. The master_planner behaviours is otherwise as follows:
- a. The master_planner obtains, from the master server, its time stamped master argument, cf. Item 1124 [Page 390];
 - b. it then invokes the master urban planning function;
 - c. the time-stamped result is offered to the plan repository;
 - d. if the result is OK as a final result,
 - e. then the behaviour is stopped;
 - f. otherwise
 - i. the master planner inquires the derived planner index generator as for such derived planner identifiers which are not used;
 - ii. the master planner behaviour is resumed with the appropriately updated programmable script pointer attribute, in parallel with
 - iii. the distributed parallel composition of the parallel behaviours of the derived servers
 - iv. and the derived planners
 - v. designated by the derived planner identifiers transcribed into (*nm_dp_ui*) derived server, respectively into (*nm_dp_ui*) derived planner names. For these transcription maps we refer to Sect. E.8.12 [Page 363], Item 1006 [Page 364].

value

```

1131 master_planner(uid,meroo,(script,anms,puis))(ptrs) ≡
1131a   let (t,((maux,mreq),(tasm,ahist))) = mps_mp_ch ? in
1131b   let ((dpuis',ptrs'),mres) = master_urban_planning(script,anms,dpuis,ptrs) in
1131c   p_pr_ch[uid] ! mres ;
1131d   if completed(mres) assert: ptrs' = {}
1131e   then init_der_serv_plans(uid,dpuis')
1131f   else
1131(f)i   init_der_serv_plans(ui,dpuis)
1131(f)ii  || master_planner(uid,meroo,(script,anms,puis))(ptrs')
1131   end end end

```

The initiate derived servers and derived planners Behaviour

The *init_der_serv_plans* behaviour plays a central rôle. The outcome of the urban planning functions, whether for master or derived planners, result in a possibly empty set of derived planner identifiers, *dpuis*. If empty then that shall mean that the planner, in the iteration, of the planner behaviour is suggesting that no derived server/derived planner pairs are initiated. If *dpuis* is not empty, say consists of the set $\{dp_{ui_i}, dp_{ui_j}, \dots, dp_{ui_k}\}$ then the planner behaviour is suggesting that derived server/derived planner pairs whose planner element has one of these unique identifiers, be appropriately initiated.

1132. The `init_der_serv_planrs` behaviour takes the unique identifier, `uid`, of the “initiate issuing” planner and a suggested set of derived planner identifiers, `dpuis`.
1133. It then obtains, from the *derived planner index generator*, `dp_xg`, a subset, `dpuis'`, that may be equal to `dpuis`.

It then proceeds with the parallel initiation of

1134. derived servers (whose names are extracted, `extr_Nm`, from their identifiers, cf. Item 1000 [Page 363]),
1135. and planners (whose names are extracted, `extr_Nm`, from their identifiers, cf. Item 1001 [Page 363])
1136. for every `dp_ui` in the set `dpuis'`.

However, we must first express the selection of appropriate arguments for these server and planner behaviours.

1137. The selection of the server and planner parts, making use of the identifier to part mapping `nms_dp_ui` and `nm_dp_ui`, cf. Items 1006–1007 [Page 364];
1138. the selection of respective identifiers,
1139. mereologies, and
1140. auxiliary and
1141. requirements attributes.

value

```

1132  init_der_serv_planrs: uid:(DP_UI|MP_UI) × DP_UI-set → in,out pr_dp_xg[uid]  Unit
1132  init_der_serv_planrs(uid,dpuis) ≡
1133    let dpuis' = (pr_dp_xg_ch[uid] ! dpuis ; pr_dp_xg_ch[uid] ?) in
1137    || { let p = c_p(dp_ui), s = c_s(nms_dp_ui(dp_ui)) in
1138        let ui_p = uid_DP(p), ui_s = uid_DPS(s),
1139            me_p = mereo_DP(p), me_s = mereo_DPS(s),
1140            aux_p = attr_sAUX(p), aux_s = attr_sAUX(s),
1141            req_p = attr_sREQ(p), req_s = attr_sREQ(s) in
1134            derived_server_extr_Nm(dp_ui)(ui_s,me_s,(aux_s,req_s)) ||
1135            derived_planner_extr_Nm(dp_ui)(ui_p,me_p,(aux_p,req_p))
1136    | dp_ui:DP_UI•dpui ∈ dpuis' end end }
1132  end

```

E.14.9 The DERIVED SERVER_{nm_i}, *i*: [1 : *p*] TRANSLATOR

We refer to Sect. E.10.8 for the attributes that play a rôle in determining the derived server signature.

The TRANSLATE_DPS_{nm_j} Function

1142. The `TRANSLATE_DPS(dpsnmj)` results in three text elements:
- the **value** keyword,
 - the *signature* of the `derived_server` definition,
 - and the *body* of that definition.

The `derived_servernmj` signature of the derived server contains the *unique identifier*; the *mereology*, cf. Item E.9.7 [Page 366] – used in determining channels: the dynamic clock identifier, the analysis depository identifier, the derived planner identifier; and the *attributes* which are: the auxiliary, `dAUXnmj` and the plan requirements, `dREQnmj`.

value

```

1142 TRANSLATE_DPS( $dps_{nm_j}$ )  $\equiv$ 
1142a " value
1142b   derived_server $_{nm_j}$ :
1142b     DPS_UI $_{nm_j}$   $\times$  DPS_Mer $_{nm_j}$   $\rightarrow$  (DAUX $_{nm_j}$   $\times$  dREQ $_{nm_j}$ )  $\rightarrow$ 
1142b       in clk_ch, ad_s_ch[uid_DPS( $dps_{nm_j}$ )]
1142b       out s_p_ch[uid_DPS( $dps_{nm_j}$ )] Unit
1142c   derived_server $_{nm_j}$ 
1142c     (uid_DPS( $dps_{nm_j}$ ), mereo_DPS( $dps_{nm_j}$ )), (attr_dAUX( $dps_{nm_j}$ ), attr_dREQ( $dps_{nm_j}$ ))  $\equiv$  ... "
```

The derived_server Behaviour

The derived_server is almost identical to the master server, cf. Sect. E.14.7, except that *plans* replace *urban space* information.

1143. The derived_server obtains time from the clock, see Item 1144c, , and the most recent analysis history, assembles these together with “locally produced”
- auxiliary* planner information and
 - plan *requirements*
- as input, MP_ARG, to the master planner.
1144. The master server otherwise behaves as follows:
- it obtains latest plans and latest analysis history, and
 - then produces auxiliary planning and plan requirements commensurate, i.e., fit, with the most recently, i.e., previously produced such information;
 - it then offers a time stamped compound of these kinds of information to the derived planner,
 - whereupon the derived server resumes being the derived server, albeit with updated programmable attributes.

type

```

1143a dAUX $_{nm_j}$ 
1143b dREQ $_{nm_j}$ 
1143 dARG $_{nm_j}$  = (T  $\times$  ((dAUX $_{nm_j}$   $\times$  dREQ $_{nm_j}$ )  $\times$  (PLANS  $\times$  AHist)))
```

value

```

1144 derived_server $_{nm_j}$ (uid, mereo)(aux, req)  $\equiv$ 
1144a   let plans = ps_pr_ch[uid] ?, ahist = ad_s_ch[uid] ?,
1144b     daux:dAUX, dreq:dREQ  $\cdot$  fit_AuxReq $_{nm_j}$ ((aux, req), (daux, dreq)) in
1144c   s_p_ch[uid] ! (clk_ch?, ((maux, mreq), (plans, ahist))) ;
1144d   derived_server $_{nm_j}$ (uid, mereo)(daux, dreq)
1144   end
```

```

1144b fitAuxReq $_{nm_j}$ : (dAUX $_{nm_j}$   $\times$  dREQ $_{nm_j}$ )  $\times$  (dAUX $_{nm_j}$   $\times$  dREQ $_{nm_j}$ )  $\rightarrow$  Bool
```

```

1144b fitAuxReq $_{nm_j}$ ((aux, req), (daux, dreq))  $\equiv$  ...
```

You may wish to compare formula Items 1143–1144d above with those of formula Items 1124–1125d of Sect. E.14.7 [Page 390].

E.14.10 The DERIVED PLANNER $_{nm_i}$, $i:[1:p]$ TRANSLATOR

We refer to Sect. E.10.9 for the attributes that play a rôle in determining the derived planner signature.

The `TRANSLATE_DP` dp_{nm_j} Function

This function is an “almost carbon copy” of the `TRANSLATE_MP` dp_{nm_j} function. Thus Items 1145–1145c [Page 395] are “almost the same” as Items 1126–1126c [Page 391].

1145. The `TRANSLATE_DP` (nm_j) results in three text elements:
- the **value** keyword,
 - the *signature* of the `derived_planner` nm_j definition,
 - and the *body* of that definition.

The `derived_planner` nm_j signature of the derived planner contains the *unique identifier*, the *mereology*, cf. Item E.9.8 [Page 366] and the *attributes*: the *script*, cf. Sect. E.10.3 [Page 373] and Item 1049 [Page 373], a set of script pointers, cf. Item 1062 [Page 375], a set of analyser names, cf. Item 1063 [Page 375], a set of planner identifiers, cf. Item 1064 [Page 375], and the channels as implied by the master planner mereology.

value

```

1145 TRANSLATE_DP( $dp$ )  $\equiv$ 
1145a " value
1145b   derived_planner:  $dp_{ui}$ :DP_UI $\times$ DP_Mer $\times$ (Script $\times$ ANms $\times$ DPUIs)  $\rightarrow$  Script_Pts  $\rightarrow$ 
1145b   in   s_p_ch[ $dp_{ui}$ ], clk_ch, ad_ps_ch[ $dp_{ui}$ ]
1145b   out  p_pr_ch[ $dp_{ui}$ ]
1145b   in,out p_dp_xg_ch[ $dp_{ui}$ ] Unit
1145c   derived_planner(uid_DP( $dp$ ),mereo_DP( $dp$ ),
1145c   (attr_Script( $dp$ ),attr_ANms( $dp$ ),attr_DPUIs( $dp$ )))(attr_Script_Ptrs( $dp$ ))  $\equiv$  ... "
```

The `derived_urban_planning` Function

This function is an “almost carbon copy” of the `master_urban_planning` function. Thus Items 1146–1149c [Page 395] are “almost the same” as Items 1127–1130c [Page 391].

1146. The core of the `derived_planner` behaviour is the `derived_urban_planning` function.
1147. It takes as arguments: the *script*, a set of analyser names, a set of derived planner identifiers, a set of script pointers, and the time-stamped derived planner argument, cf. Item 1124 [Page 390];
1148. and delivers, i.e., yields, a set of “remaining” derived planner identifiers, an updated set of script pointers, and a master result, `M_RES`, i.e., a master plan, `mp`:`M_PLAN` together with the time stamped master argument from which the plan was constructed.
1149. The master urban planning function is not defined by other than a predicate:
- the “remaining” derived planner identifiers is a subset of the arguments derived planner identifiers;
 - the “resulting” master argument is the same as the input master argument, i.e., it is “carried forward”;
 - the arguments: the *script*, the analyser names, the derived planner identifiers, the set of script pointers, the time-stamped master planner argument, and the result plan otherwise satisfies a predicate $\mathcal{P}_{dnm_i}(\text{script}_{dnm_i}, \text{anms}, \text{dpuis}, \text{ptrs}, \text{marg}_{dnm_i})(\text{dplan}_{dnm_i})$ expressing that the result `mplan` is an appropriate plan in view of the other arguments.

type

```

1148 D_PLAN $_{dnm_i}$ 
1148 D_RES $_{dnm_i} = \text{D\_PLAN}_{dnm_i} \times \text{DP\_UI-set} \times \text{D\_ARG}_{dnm_i}$ 
```

value

```

1147 derived_urban_planning $_{dnm_i}$ :
1147   Script $_{dnm_i} \times \text{ANm-set} \times \text{DP\_UI-set} \times \text{Script\_Ptr-set} \times \text{D\_ARG}_{dnm_i}$ 
```

```

1148     → (DP_UI-set × Script_Ptr-set) × D_RESdnmi
1146 derived_urban_planningdnmi(script,anms,dpuis,ptrs,darg)
1149a   as ((dpuis',ptrs'),(dplan,ptrs'darg'))
1149a   dpuis' ⊆ dpuis
1149b   ∧ darg' = darg
1149c   ∧  $\mathcal{P}_{dnm_i}$ (script,anms,dpuis,ptrs,darg),((dpuis',ptrs'),(dplan,ptrs'darg'))

1146  $\mathcal{P}_{dnm_i}$ : ((Scriptdnmi × ANM-set × DP_UI-set × Script_Ptr-set × D_ARGdnmi)
1146           × (DP_UI-set × Scriptdnmi_Ptr-set × D_RESdnmi)) → Bool
1146  $\mathcal{P}_{dnm_i}$ ((scriptdnmi,anms,dpuis,ptrs,dargdnmi),(dp_uis',ptrs',dres)) ≡ ...

```

The derived_planner_{nm_j} Behaviour

This behaviour is an “almost carbon copy” of the `derived_plannernmj` behaviour. Thus Items 1150–1150k [Page 396] are “almost the same” as Items 1131–1131(f)v [Page 392].

1150. The `derived_planner` behaviour is otherwise as follows:
- The `derived_planner` obtains, from the derived server, its time stamped master argument, cf. Item 1124 [Page 390];
 - it then invokes the derived urban planning function;
 - the time-stamped result is offered to the plan repository;
 - if the result is OK as a final result,
 - then the behaviour is stopped;
 - otherwise
 - the derived planner inquires the derived planner index generator as for such derived planner identifiers which are not used;
 - the derived planner behaviour is the resumed with the appropriately updated programmable script pointer attribute, in parallel with
 - the distributed parallel composition of the parallel behaviours of the derived servers
 - and the derived planners
 - designated by the derived planner identifiers transcribed into (*nm_dp_{ui}*) derived server, respectively into (*nm_dp_{ui}*) derived planner names. For these transcription maps we refer to Sect. E.8.12 [Page 363], Item 1006 [Page 364].

value

```

1131 derived_plannerdnmi(uid,merao,(scriptdnmi,anms,puis))(ptrs) ≡
1131a   let (t,((daudnmi,dreqdnmi),(plans,ahist))) = s_p_ch[uid] ? in
1131b   let ((dpuis',ptrs'),dresdnmi) = derived_urban_planningdnmi(scriptdnmi,anms,dpuis,ptrs) in
1131c   p_pr_ch[uid] ! dresdnmi ;
1131d   if completed(dresdnmi)
1131e     then init_der_serv_planrs(uid,dpuis') assert: ptrs' = {}
1131f   else
1131(f)i     init_der_serv_plans(uid,dpuis')
1131(f)ii    || derived_planner(uid,merao,(scriptdnmi,anms,puis))(ptrs')
1131   end end end

```

E.15 Initialisation of The Urban Space Analysis & Planning System

Section E.12 presents a *compiler* from *structures* and *parts* to *behaviours*. This section presents an initialisation of some of the behaviours. First we postulate a global *universe of discourse*, *uod*. Then we

summarise the global values of *parts* and *part names*. This is followed by a summaries of *part qualities* – in four subsections: a summary of the global values of unique identifiers; a summary of channel declarations; the system as it is initialised; and the system of derived servers and planners as they evolve.

E.15.1 Summary of Parts and Part Names

value

957 [Page 359] $uod : UoD$
 958 [Page 359] $clk : CLK = \text{obs_CLK}(uod)$
 959 [Page 359] $tus : TUS = \text{obs_TUS}(uod)$
 960 [Page 359] $ans : A_{ann_i}\text{-set}, i:[1..n] = \{ \text{obs_A}_{ann_i}(aa) \mid aa \in (\text{obs_AA}(uod)), i:[1..n] \}$
 961 [Page 359] $ad : AD = \text{obs_AD}(\text{obs_AA}(uod))$
 962 [Page 359] $mps : MPS = \text{obs_MPS}(\text{obs_MPA}(uod))$
 963 [Page 359] $mp : MP = \text{obs_MP}(\text{obs_MPA}(uod))$
 964 [Page 359] $dps : DPS_{nm_i}\text{-set}, i:[1..p] =$
 964 [Page 359] $\{ \text{obs_DPS}_{nm_i}(dpc_{nm_i}) \mid$
 964 [Page 359] $dpc_{nm_i} : DPC_{nm_i} \cdot dpc_{nm_i} \in \text{obs_DPCS}_{nm_i}(\text{obs_DPA}(uod)), i:[1..p] \}$
 965 [Page 359] $dps : DP_{nm_i}\text{-set}, i:[1..p] =$
 965 [Page 359] $\{ \text{obs_DP}_{nm_i}(dpc_{nm_i}) \mid$
 965 [Page 359] $dpc_{nm_i} : DPC_{nm_i} \cdot dpc_{nm_i} \in \text{obs_DPCS}_{nm_i}(\text{obs_DPA}(uod)), i:[1..p] \}$
 966 [Page 359] $dpxg : DPXG = \text{obs_DPXG}(uod)$
 967 [Page 359] $pr : PR = \text{obs_PR}(uod)$
 968 [Page 359] $spsps : (DPS_{nm_i} \times DP_{nm_i})\text{-set}, i:[1..p] =$
 968 [Page 359] $\{ (\text{obs_DPS}_{nm_i}(dpc_{nm_i}), \text{obs_DP}_{nm_i}(dpc_{nm_i})) \mid$
 968 [Page 359] $dpc_{nm_i} : DPC_{nm_i} \cdot dpc_{nm_i} \in \text{obs_DPCS}_{nm_i}(\text{obs_DPA}(uod)), i:[1..p] \}$

E.15.2 Summary of of Unique Identifiers

value

983 [Page 362] $clk_{ui} : CLK_UI = \text{uid_CLK}(uod)$
 984 [Page 362] $tus_{ui} : TUS_UI = \text{uid_TUS}(uod)$
 985 [Page 362] $a_{uis} : A_UI\text{-set} = \{ \text{uid_A}(a) \mid a : A \cdot a \in ans \}$
 986 [Page 362] $ad_{ui} : AD_UI = \text{uid_AD}(ad)$
 987 [Page 362] $mps_{ui} : MPS_UI = \text{uid_MPS}(mps)$
 988 [Page 362] $mp_{ui} : MP_UI = \text{uid_MP}(mp)$
 989 [Page 362] $dps_{uis} : DPS_UI\text{-set} = \{ \text{uid_DPS}(dps) \mid dps : DPS \cdot dps \in dps \}$
 990 [Page 362] $dp_{uis} : DP_UI\text{-set} = \{ \text{uid_DP}(dp) \mid dp : DP \cdot dp \in dps \}$
 991 [Page 362] $dpxg_{ui} : DPXG_UI = \text{uid_DPXG}(dpxg)$
 992 [Page 362] $pr_{ui} : PR_UI = \text{uid_PR}(pr)$
 992 [Page 362] $s_{uis} : (MPS_UI \mid DPS_UI)\text{-set} = \{ mps_{ui} \} \cup dps_{uis}$
 994 [Page 362] $p_{uis} : (MP_UI \mid DP_UI)\text{-set} = \{ mp_{ui} \} \cup dp_{uis}$
 995 [Page 362] $sips : (DPS_UI \times DP_UI)\text{-set} = \{ (\text{uid_DPS}(dps), \text{uid_DP}(dp)) \mid (dps, dp) : (DPS \times DP) \cdot (dps, dp) \in sps \}$
 996 [Page 362] $si_pi_m : DPS_UI \xrightarrow{\text{m}} DP_UI = \{ \text{uid_DPS}(dps) \mapsto \text{uid_DP}(dp) \mid (dps, dp) : (DPS \times DP) \cdot (dps, dp) \in sps \}$
 997 [Page 362] $pi_si_m : DP_UI \xrightarrow{\text{m}} DPS_UI = \{ \text{uid_DP}(dp) \mapsto \text{uid_DPS}(dps) \mid (dps, dp) : (DPS \times DP) \cdot (dps, dp) \in sps \}$

E.15.3 Summary of Channels

channel

1078 [Page 379] clk_ch:CLK_MSG
 1079 [Page 380] $\{\text{tus_a_ch}[a_ui]:\text{TUS_MSG}|a_ui:A_UI \cdot a_ui \in a_{uiS}\}$
 1081 [Page 380] $\text{tus_mps_ch:TUS_MSG}$
 1082 [Page 380] $\{\text{a_ad_ch}[a_ui]:A_MSG|a_ui:A_UI \cdot a_ui \in a_{uiS}\}$
 1084 [Page 381] $\{\text{ad_s_ch}[s_ui]|s_ui:(\text{MPS_UI}|\text{DPS_UI}) \cdot s_ui \in \{mps_{ui}\} \cup dps_{uiS}\}:\text{AD_MSG}$
 1086 [Page 381] mps_mp_ch:MPS_MSG
 1088 [Page 381] $\{\text{p_pr_ch}[p_ui]:\text{PLAN_MSG}|p_ui:(\text{MP_UI}|\text{DP_UI}) \cdot p_ui \in p_{uiS}\}$
 1090 [Page 382] $\{\text{p_dpxg_ch}[ui]:\text{DPXG_MSG}|ui:(\text{MP_UI}|\text{DP_UI}) \cdot ui \in p_{uiS}\}$
 1092 [Page 382] $\{\text{pr_s_ch}[ui]:\text{PR_MSGd}|ui:(\text{MPS_UI}|\text{DPS_UI}) \cdot ui \in s_{uiS}\}$
 1094 [Page 382] $\{\text{dps_dp_ch}[ui]:\text{DPS_MSG}_{nm_j}|ui:\text{DPS_UI} \cdot ui \in dps_{uiS}\}$

E.15.4 The Initial System

1103c [Page 384] $\text{urb_spa}(\text{uid_TUS}(tus), \text{mereo_TUS}(tus))(\text{attr_Pts}(tus))$
 \parallel
 1096c [Page 383] $\text{clock}(\text{uid_CLK}(clk), \text{mereo_CLK}(clk))(\text{attr_T}(clk))$
 \parallel
 1110 [Page 385] $\parallel \{\text{analyser}_{ui_i}(\text{uid_A}(a_{ui_i}), \text{mereo_A}(a_{ui_i}))(\text{ana}_{anm_i}) \mid ui_i:A_UID \cdot ui_i \in a_{uiS}\}$
 \parallel
 1096c [Page 383] $\text{ana_dep}(\text{ui_A}(ad), \text{mereo_A}(ad))(\text{attr_AHist}(ad))$
 \parallel
 1121c [Page 389] $\text{plan_rep}(\text{plans})(\text{attr_AllDPUIs}(pr), \text{attr_UsedDPUIs}(pr))$
 \parallel
 1119c [Page 388] $\text{dpxg_beh}(\text{uid_DPXG}(dpxg), \text{mereo_DPXG}(dpxg))(\text{all_dpuis}, \text{used_dpuis})$
 \parallel
 1123c [Page 390] $\text{master_server}(\text{uid_MPS}(mps), \text{mereo_MPS}(mps))(\text{attr_mAUX}(mps), \text{attr_mREQ}(mps))$
 \parallel
 1126c [Page 391] $\text{master_planner}(\text{uid_MP}(mp), \text{mereo_MP}(mp),$
 1126c [Page 391] $\quad (\text{attr_Script}(mp), \text{attr_ANms}(mp), \text{attr_DPUIs}(mp)))(\text{attr_Script_Ptrs}(mp))$

E.15.5 The Derived Planner System

1142c [Page 393] $\{ \text{derived_server}_{dps_{nm_j}}$
 1142c [Page 393] $\quad (\text{uid_DPS}(dps_{nm_j}), \text{mereo_DPS}(dps_{nm_j}))(\text{attr_dAUX}(dps_{nm_j}), \text{attr_dREQ}(dps_{nm_j}))$
 \parallel
 1145c [Page 395] $\text{derived_planner}(\text{uid_DP}(dp_{nm_j}), \text{mereo_DP}(dp_{nm_j}),$
 1145c [Page 395] $\quad (\text{attr_Script}(dp_{nm_j}), \text{attr_ANms}(dp_{nm_j}), \text{attr_DPUIs}(dp_{nm_j})))$
 1145c [Page 395] $\mid j:[1..p] \}$

E.16 Further Work

E.16.1 Reasoning About Deadlock, Starvation, Live-lock and Liveness

The current author is quite unhappy about the way in which he has defined the urban planning, oracle and repository behaviours. Such issues as which invariants are maintained across behaviours are not addressed. In fact, it seems to be good practice, following Dijkstra, Lamport and others, to formulate appropriate such invariants and only then “derive” behaviour definitions accordingly. In a rewrite of this research note, if ever, into a proper paper, the current author hopes to follow proper practices. He hopes to find younger talent to co-author this effort.

E.16.2 Document Handling

I may appear odd to the reader that I now turn to document handling. One central aspect of urban planning, strange, perhaps, to the reader, is that of handling the “zillions upon zillions” of documents that enter into and accrue from urban planning. If handling of these documents is not done properly a true nightmare will occur. So we shall briefly examine the urban planning document situation ! From that we conclude that we must first try understand:

- **What do we mean by a document?**

Urban Planning Documents

The urban planning functions and the urban planning behaviours, including both the base and the n derived variants, rely on documents. These documents are **created, edited, read, copied**, and, eventually, **shredded** by urban-planners. Editing documents result in new versions of “the same” document. While a document is being **edited** or **read** we think of it as not being **accessible** to other urban-planners. If urban-planners need to read a latest version of a document while that version is subject to editing by another urban planner, copies must first be made, before editing, one for each “needy” reader. Once, editing has and readings have finished, the “reader” copies need, or can, be shredded.

A Document Handling System

In Appendix D, [72], we sketch a document handling system domain. That is, not a document handling software system, not even requirements for a document handling software system, but just a description which, in essence, models documents and urban planners’ actions on documents. (The urban planners are referred to as document handlers.) The description further models two ‘aggregate’ notions: one of ‘handler management’, and one of ‘document archive’. Both seem necessary in order to “sort out” the granting of document access rights (that is, permissions to perform operations on documents), and the creation and shredding of documents, and in order to avoid dead-locks in access to and handling of documents.

E.16.3 Validation and Verification (V&V)

By **validation** of a document we shall mean: the primarily informal and social process of checking that the document description meets customer expectations.

Validation serves to get the right product.

By **verification** of a document we shall mean: the primarily formal, i.e., mathematical process of checking, testing and formal proof that the model, which the document description entails, satisfies a number of properties.

Verification serves to get the product right.

By **validation of the urban planning model** of this document we shall understand the social process of explaining the model to urban planning stakeholders, to obtain their reaction, and to possibly change the model according to stakeholder objections.

By **verification of the urban planning model** of this document we shall understand the formal process, based on formalisations of the argument and result types of the description, of testing, model checking and formally proving properties of the model.

MORE TO COME

E.16.4 Urban Planning Project Management

In this research note we have focused on the urban planning project behaviours, their interactions, and their information “passing”. Usually publications about urban planning: research papers, technical papers, survey papers, etcetera, focus on specific “functions”. In this research note we do not. We focus instead on what we can say about the domain of urban planning: the fact, or the possibility, that an initial, a core, here referred to as a base, urban planning effort (i.e., project, hence behaviour) can “spew off”, generate, a number of (derived, i.e., in some sense subsidiary), more specialised, urban planning projects.

Urban Planning Projects

We think of a comprehensive urban planning project as carried out by urban planners. As is evident from the model the project consists of one base urban planning project and up to n derived urban planning projects. The urban planners involved in these projects are professionals in the areas of planning:

- master urban planning issues:
 - ⊗ geodesy,
 - ⊗ geotechniques,
 - ⊗ meteorology,
- master urban plans:
 - ⊗ cartography,
 - ⊗ cadastral matters,
 - ⊗ zoning;
- derived urban planning issues:
 - ⊗ industries,
- ⊗ residential and shopping,
- ⊗ apartment buildings,
- ⊗ villas,
- ⊗ recreational,
- ⊗ etcetera;
- technological infrastructures:
 - ⊗ transport,
 - ⊗ electricity,
 - ⊗ telecommunications,
 - ⊗ gas,
- ⊗ water,
- ⊗ waste,
- ⊗ etcetera;
- societal infrastructures:
 - ⊗ health care,
 - ⊗ schools,
 - ⊗ police,
 - ⊗ fire brigades,
 - ⊗ etcetera;
 - etcetera, etcetera, etcetera !

To anyone with any experience in getting such diverse groups and individuals of highly skilled professionals to work together it is obvious that some form of management is required. The term ‘comprehensive’ was mentioned above. It is meant to express that the comprehensive urban planning project is the only one “dealing” with a given geographic area, and that no other urban planning projects “infringe” upon it, that is, “deal” with sub-areas of that given geographic area.

Strategic, Tactical and Operational Management

We can distinguish between

- strategic,
- tactical and
- operational

management.

Project Resources

But first we need take a look at the **resources** that management is charged with:

- the urban planners, i.e., humans,
- time,
- finances,
- office space,
- support technologies: computing etc.,
- etcetera.

Strategic Management

By **strategic management** we shall understand the analysis and decisions of, and concerning, scarce resources: people (skills), time, monies: their deployment and trade-offs.

Tactical Management

By **tactical management** we shall understand the analysis and decisions with respect to budget and time plans, and the monitoring and control of serially reusable resources: office space, computing.

Operational Management

By **operational management** we shall understand the monitoring and control of the enactment, progress and completion of individual deliverables, i.e., documents, the quality (adherence to “standards”, fulfillment of expectations, etc.) of these documents, and the day-to-day human relations.

Urban Planning Management

The above (*strategic, tactical & operational management*) translates, in the context of *urban planning*, into:

TO BE WRITTEN

E.17 Conclusion

TO BE WRITTEN

E.17.1 What Were Our Expectations?

E.17.2 What Have We Achieved?

TO BE WRITTEN

E.17.3 What Next?

TO BE WRITTEN

E.17.4 Acknowledgement

TO BE WRITTEN

F

Swarms of Drones

We speculate on a domain of *swarms* and *drones monitored and controlled by a command center* in some *geography*. Awareness of swarms is registered only in an enterprise command center. We think of these swarms of drones as an enterprise of either package deliverers, crop-dusters, insect sprayers, search & rescuers, traffic monitors, or wildfire fighters – or several of these, united in a notion of *an enterprise* possibly consisting of “disjoint” *businesses*. We analyse & describe the properties of these phenomena as *endurants* and as *perdurants*: parts one can observe and behaviours that one can study. We do not yet examine the problem of drone air traffic management¹. The analysis & description of this postulated domain follows the principles, techniques and tools laid down in [70].

F.1 An Informal Introduction

F.1.1 Describable Entities

The Endurants: Parts

In the universe of discourse we observe *endurants*, here in the form of parts, and *perdurants*, here in the form of behaviours.

The parts are *discrete endurants*, that is, can be seen or touched by humans, or that can be conceived as an abstraction of a discrete part.

We refer to Fig. F.1 [Page 404].

There is a *universe of discourse*, uod:UoD. The universe of discourse embodies: an *enterprise*, e:E. The enterprise consists of an *aggregate of enterprise drones*, aed:AED (which consists of a set, eds:EDs, of enterprise drones). and a *command center*, cc:CC; The universe of discourse also embodies a *geography*, g:G. The universe of discourse finally embodies an *aggregate of 'other' drones*, aod:AOD (which consists of a set, ods:ODs, of these 'other' drones). A *drone* is an *unmanned aerial vehicle*.² We distinguish between *enterprise drones*, ed:ED, and *'other' drones*, od:OD. The pragmatics of the enterprise swarms is that of providing enterprise drones for one or more of the following kinds of *businesses*:³ delivering parcels

¹ <https://www.nasa.gov/feature/ames/first-steps-toward-drone-traffic-management>, <http://www.sciencedirect.com/science/article/pii/S20460>

² Drones are also referred to as UAVs.

³ <http://www.latimes.com/business/la-fi-drone-traffic-20170501-htlstory.html>

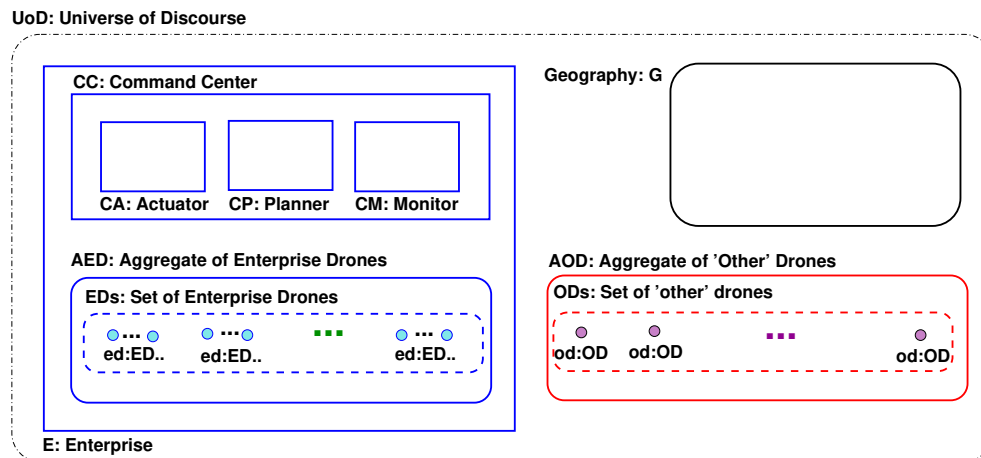


Fig. F.1. Universe of Discourse

(mail, packages, etc.)⁴, crop dusting⁵, aerial spraying⁶, wildfire fighting⁷, traffic control⁸, search and rescue⁹, etcetera. A notion of *swarm* is introduced. A swarm is a concept. As a concept a swarm is a set of drones. We associate swarms with businesses. A business has access to one or more swarms. The enterprise *command center*, cc:CC, can be seen as embodying three kinds of functions: a *monitoring* service, cm:CM, whose function it is to know the locations and dynamics of all drones, whether enterprise drones or ‘other’ drones; a *planning* service, cp:CP, whose function it is to plan the next moves of all that enterprise’s drones; and an *actuator* service, ca:CA, whose functions it is to guide that enterprise’s drones as to their next moves. The swarm concept “resides” in the command planner.

The Perdurants

The perdurants are entities for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, were we to freeze time we would only see or touch a fragment of the perdurant.

The major ***

MORE TO COME

F.1.2 The Contribution of [70]

The major contributions of [70] are these: a methodology¹⁰ for analysing & describing manifest domains¹¹, where the methodology builds on an *ontological principle* of viewing the domains as consisting of *endurants*

⁴ <https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011> and <https://www.digitaltrends.com/cool-tech/amazon-prime-air-delivery-drones-history-progress/>

⁵ <http://www.uavcropdusterspray.com/>, <http://sprayingdrone.com/>

⁶ <https://abjdrones.com/commercial-drone-services/industry-specific-solutions/agriculture/>

⁷ <https://www.smithsonianmag.com/videos/category/innovation/drones-are-now-being-used-to-battle-wildfires/>

⁸ https://business.esa.int/sites/default/files/Presentation%20on%20UAV%20Road%20Surface%20Monitoring%20and%20Traffic%20Information_0.pdf

⁹ <http://sardrones.org/>

¹⁰ By a *methodology* we shall understand a set of *principles* for selecting and applying a number of *techniques*, using *tools*, to – in this case – analyse & describe a domain.

¹¹ A manifest domain is a human- and artifact-assisted arrangement of *endurant*, that is spatially “stable”, and *perdurant*, that is temporally “fleeting” entities. *Endurant* entities are either parts or components or materials. *Perdurant* entities are either actions or events or behaviours.

and *perdurants*. Endurants possess properties such as *unique identifiers*, *mereologies*, and *attributes*. Perdurants are then analysed & described as either *actions*, *events*, or *behaviours*. The *techniques* to go with the ***

The *tools* are ***

MORE TO COME

MORE TO COME

F.1.3 The Contribution of This Report

TO BE WRITTEN

We relate our work to that of [181].

•••

The main part of this report is contained in the next three sections: endurants; states, constants, and operations on states; and perdurants.

F.2 Entities, Endurants

By an *entity* we shall understand a *phenomenon*, i.e., something that can be observed, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity. We further demand that an entity can be objectively described.

By an *endurant* we shall understand an entity that can be observed or conceived and described as a “complete thing” at no matter which given snapshot of time. Were we to “freeze” time we would still be able to observe the entire endurant.

F.2.1 Parts, Atomic and Composite, Sorts, Abstract and Concrete Types

By a *discrete endurant* we shall understand an endurant which is separate, individual or distinct in form or concept.

By a *part* we shall understand a discrete endurant which the domain engineer chooses to endow with internal qualities such as unique identification, mereology, and one or more attributes. We shall define the concepts of unique identifier, mereology and attribute later in this case study.

Atomic parts are those which, in a given context, are deemed to not consist of meaningful, separately observable proper sub-parts.

Sub-parts are parts.

Composite parts are those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts.

By a *sort* we shall understand an abstract type.

By a *type* we shall here understand a set of values “of the same kind” – where we do not further define what we mean by *the same kind*”.

By an *abstract type* we shall understand a type about whose values we make no assumption [as to their atomicity or composition].

By a *concrete type* we shall understand a type about whose values we are making certain assumptions as to their atomicity or composition, and, if composed then how and from which other types they are composed.

Universe of Discourse

By a *universe of discourse* we shall understand *that which we can talk about, refer to and whose entities we can name*. Included in that universe is the *geography*. By *geography* we shall understand a section of the globe, an area of land, its geodecy, its meteorology, etc.

1151. In the **Universe of Discourse** we can observe the following parts:
- a. an atomic **Geography**,
 - b. a composite **Enterprise**,
 - c. and an aggregate of '**Other**'¹² **Drones**.

type

1151 UoD, G, E, AOD

value

1151a obs_G: UoD → G

1151b obs_E: UoD → E

1151c obs_AOD: UoD → AOD

The Enterprise

1152. From an enterprise one can observe:
- a. a(n enterprise) command center. and
 - b. an aggregate of enterprise drones.

type

1152a CC

1152a AED

value

1152a obs_CC: E → CC

1152b obs_AED: E → AED

From Abstract Sorts to Concrete Types

1153. From an *aggregate of enterprise drones*, AED, we can observe a possibly empty set of drones, EDs
 1154. From an *aggregate of 'other' drones*, AOD, we can observe a possibly empty set, ODs, of '*other*' drones.

type

1153 ED

1153 EDs = ED-**set**

1154 OD

1154 ODs = OD-**set**

value

1153 obs_EDs: AED → EDs

1154 obs_ODs: AOD → ODs

Drones, whether 'other' or 'enterprise', are considered atomic.

¹² We apologize for our using the term 'other' drones. These 'other' drones are not necessarily adversary or enemy drones. They are just there – coexisting with the enterprise drones.

The Auxiliary Function xtr_Ds :

We define an auxiliary function, xtr_Ds .

1155. From the universe of discourse we can extract all its drones;
 1156. similarly from its enterprise;
 1157. similarly from the aggregate of enterprise drones; and
 1158. from an aggregate of ‘other’ drones.

1155 $xtr_Ds: UoD \rightarrow (ED|OD)\text{-set}$
 1155 $xtr_Ds(uod) \equiv$
 1155 $\quad \cup \{xtr_Ds(obs_AED(obs_E(uod)))\} \cup xtr_Ds(obs_AOD(uod))$
 1156 $xtr_Ds: E \rightarrow ED\text{-set}$
 1156 $xtr_Ds(e) \equiv xtr_Ds(obs_AED(e))$
 1157 $xtr_Ds: AED \rightarrow ED\text{-set}$
 1157 $xtr_Ds(aed) \equiv obs_EDs(obs_EDs(aed))$
 1158 $xtr_Ds: AOD \rightarrow OD\text{-set}$
 1158 $xtr_Ds(aod) \equiv obs_ODs(aod)$

1159. In the universe of discourse a drone cannot be both among the enterprise drones and among the ‘other’ drones.

axiom

1159 $\forall uod:UoD, e:E, aed:ES, aod:AOD \cdot$
 1159 $\quad e=obs_E(uod) \wedge aed=obs_AED(e) \wedge aod:obs_AOD(uod)$
 1159 $\quad \Rightarrow xtr_Ds(aed) \cap xtr_Ds(aod) = \{\}$

The functions are partial as the supplied swarm identifier may not be one of the universe of discourse, etc.

Command Center

A Simple Narrative Figure F.1 [Page 404] shows a graphic rendition of a space of interest. The command center, CC, a composite part, is shown to include three atomic parts: An atomic part, the monitor, CM. It monitors the location and dynamics of all drones. An atomic part, the planner, CP. It plans the next, “friendly”, drone movements. The command center also has yet an atomic part, the actuator, CA. It informs “friendly” drones of their next movements. The planner is where “resides” the notion of a enterprise consisting of one or more businesses, where each business has access to zero, one or more swarms, where a swarm is a set of enterprise drone identifiers.

The purpose of the control center is to monitor the whereabouts and dynamics of all drones (done by CM); to plan possible next actions by enterprise drones (done by CP); and to instruct enterprise drones of possible next actions (done by CA).

Command Center Decomposition From the composite command center we can observe

1160. the center monitor, CM;
 1161. the center planner, CP; and
 1162. the center actuator, CA .

type	value
1160 CM	1160 $obs_CM: CC \rightarrow CM$
1161 CP	1161 $obs_CP: CC \rightarrow CP$
1162 CA	1162 $obs_CA: CC \rightarrow CA$

F.2.2 Unique Identifiers

Parts are distinguishable through their unique identifiers. A *unique identifier* is a further undefined quantity which we associate with parts such that no two parts of a universe of discourse are identical.

The Enterprise, the Aggregates of Drones and the Geography

1163. Although we may not need it for subsequent descriptions we do, for completeness of description, introduce unique identifiers for parts and sub-parts of the universe of discourse:
- Geographies, $g:G$, have unique identification.
 - Enterprises, $e:E$, have unique identification.
 - Aggregates of enterprise drones, $aed:AED$, have unique identification.
 - Aggregates of ‘other’ drones, $aod:AOD$, have unique identification.
 - Command centers, $cc:CC$, have unique identification.

type

1163 $GI, EI, AEDI, AODI, CCI$

value

1163a $uid_G: G \rightarrow GI$

1163b $uid_E: E \rightarrow EI$

1163c $uid_{AED}: AED \rightarrow AEDI$

1163d $uid_{OD}: AOD \rightarrow AODI$

1163e $uid_{CC}: CC \rightarrow CCI$

Unique Command Center Identifiers

1164. The monitor has a unique identifier.
 1165. The planner has a unique identifier.
 1166. The actuator has a unique identifier.

type

1164 CMI

1165 CPI

1166 CAI

value

1164 $uid_{CM}: CM \rightarrow CMI$

1165 $uid_{CP}: CP \rightarrow CPI$

1166 $uid_{CA}: CA \rightarrow CAI$

Unique Drone Identifiers

1167. Drones have unique identifiers.
- whether enterprise or
 - ‘other’ drones

type

1167 $DI = EDI \mid ODI$

value

1167a $uid_{ED}: ED \rightarrow EDI$

1167b $uid_{OD}: OD \rightarrow ODI$

Auxiliary Function: xtr_dis:

1168. From the aggregate of enterprise drones;
 1169. From the aggregate of ‘other’ drones;
 1170. and from the two parts of a universe of discourse: the enterprise and the ‘other’ drones.

value

- 1168 xtr_dis: AED \rightarrow DI-set
 1168 xtr_dis(aed) $\equiv \{uid_ED(ed)|ed:ED \cdot ed \in obs_EDs(aed)\}$
 1169 xtr_dis: AOD \rightarrow DI-set
 1169 xtr_dis(aod) $\equiv \{uid_D(od)|od:OD \cdot od \in obs_ODs(aod)\}$
 1170 xtr_dis: UoD \rightarrow DI-set
 1170 xtr_dis(uod) $\equiv xtr_dis(obs_AED(uod)) \cup xtr_dis(obs_AOD(uod))$

Auxiliary Function: xtr_D:

1171. From the universe of discourse, given a drone identifier of that space, we can extract the identified drone;
 1172. similarly from the enterprise;
 1173. its aggregate of enterprise drones; and
 1174. and from its aggregate of ‘other’ drones;

- 1171 xtr_D: UoD \rightarrow DI $\xrightarrow{\sim}$ D
 1171 xtr_D(uod)(di) $\equiv \mathbf{let} d:D \cdot d \in xtr_Ds(uod) \wedge uid_D(d)=di \mathbf{in} d \mathbf{end}$
 1171 **pre:** di $\in xtr_dis(soi)$
 1172 xtr_D: E \rightarrow DI $\xrightarrow{\sim}$ D
 1172 xtr_D(e)(di) $\equiv \mathbf{let} d:D \cdot d \in xtr_Ds(obs_ES(e)) \wedge uid_D(d)=di \mathbf{in} d \mathbf{end}$
 1172 **pre:** di $\in xtr_dis(e)$
 1173 xtr_D: AED \rightarrow DI $\xrightarrow{\sim}$ D
 1173 xtr_D(aed)(di) $\equiv \mathbf{let} d:D \cdot d \in xtr_Ds(aed) \wedge uid_D(d)=di \mathbf{in} d \mathbf{end}$
 1173 **pre:** di $\in xtr_dis(es)$
 1174 xtr_D: AOD \rightarrow DI $\xrightarrow{\sim}$ D
 1174 xtr_D(aod)(di) $\equiv \mathbf{let} d:D \cdot d \in xtr_Ds(aod) \wedge uid_D(d)=di \mathbf{in} d \mathbf{end}$
 1174 **pre:** di $\in xtr_dis(ds)$

F.2.3 Mereologies**Definition**

Mereology is the study and knowledge of parts and their relations (to other parts and to the “whole”) [104].

Origin of the Concept of Mereology as Treated Here

We shall [thus] deploy the concept of mereology as advanced by the Polish mathematician, logician and philosopher Stanislaw Lécsniewski. Douglas T. (“Doug”) Ross¹³ also contributed along the lines of our approach [227] – hence [78] is dedicated to Doug.

¹³ Doug Ross is the originator of the term CAD for *computer aided design*, of APT for *Automatically Programmed Tools*, a language to drive numerically controlled manufacturing, and also SADT for *Structure Analysis and Design Techniques*

Basic Mereology Principle

The basic principle in modelling the mereology of a any universe of discourse is as follows: Let p' be a part with unique identifier p'_{id} . Let p be a sub-part of p' with unique identifier p_{id} . Let the immediate sub-parts of p be p_1, p_2, \dots, p_n with unique identifiers $p_{1id}, p_{2id}, \dots, p_{nid}$. That p has mereology $(p'_{id}, \{p_{1id}, p_{2id}, \dots, p_{nid}\})$. The parts p_j , for $1 \leq j \leq n$ for $n \geq 2$, if atomic, have mereologies $(p_{id}, \{p_{1id}, p_{2id}, \dots, p_{j-1id}, p_{j+1id}, \dots, p_{nid}\})$ – where we refer to the second term in that pair by m ; and if composite, have mereologies $(p_{id}, (m, m'))$, where the m' term is the set of unique identifiers of the sub-parts of p_j .

Engineering versus Methodical Mereology

We shall restrict ourselves to an engineering treatment of the mereology of our universe of discourse. That is in contrast to a strict, methodical treatment. In a methodical description of the mereologies of the various parts of the universe of discourse one assigns a mereology to every part: to the enterprise, the aggregate of 'other' drones and the geography; to the command center of the enterprise and its aggregate of drones; to the monitor, the planner and the actuator of the command center; to the drones of the aggregate of enterprise drones, and to the drones of the aggregate of 'other' drones. We shall "shortcut" most of these mereologies. The reason is this: The *pragmatics* of our attempt to model *drones*, is rooted in our interest in the interactions between the command center's monitor and actuator and the enterprise and 'other' drones. For "completeness" we also include interactions between the geography's meteorology and the above command center and drones. The mereologies of the enterprise, E, the enterprise aggregate of drones AED, and the set of (enterprise) drones, EDs, do not involve drone identifiers. The only "thing" that the monitor and actuator are interested in are the drone identifiers. So we shall thus model the mereologies of our universe of discourse by omitting mereologies for the enterprise, the aggregates of drones, the sets of these aggregates, and the geography, and only describe the mereologies of the monitor, planner and actuator, the enterprise drones and the 'other' drones.

Planner Mereology

1175. The planner mereology reflects the center planners awareness¹⁴ of the monitor, the actuator,, and the geography of the universe of discourse.
1176. The planner mereology further reflects that a *eureka*¹⁵ is provided by, or from, an outside source reflected in the autonomous attribute Cmdl. The value of this attribute changes at its own volition and ranges over commands that directs the planner to perform either of a number of operations.

Eureka examples are: calculate and effect a new flight plan for one or more designated swarms of a designated business; effect the transfer of an enterprise drone from a designated swarm of a business to another, distinctly designated swarm of the same business; etcetera.

type

- 1175 CPM = (CAI × CMI × GI) × Eureka
 1176 Eureka == mkNewFP(BI × SI-set × Plan)
 1176 | mkChgDB(fsi:SI × tsi:SI × di × DI)
 1176 | ...

value

¹⁴ That "awareness" includes, amongst others, the planner obtaining information from the monitor of the whereabouts of all drones and providing the actuator with directives for the enterprise drones — all in the context of the *land* and "its" *meteorology*.

¹⁵ "Eureka" comes from the Ancient Greek word *εὕρηκα* *heúrēka*, meaning "I have found (it)", which is the first person singular perfect indicative active of the verb *εὕρηκα* *heuriskō* "I find".[1] It is closely related to heuristic, which refers to experience-based techniques for problem solving, learning, and discovery.

1175 mereo_CP: CP \rightarrow CPM
 1176 Plan = ...

We omit expressing a suitable axiom concerning center planner mereologies. Our behavioural analysis & description of monitoring & control of operations on the space of drones will show that command center mereologies may change.

Monitor Mereology

The monitor's mereology reflects its awareness of the drones whose position and dynamics it is expected to monitor.

1177. The mereology of the center monitor is a pair: the set of unique identifiers of the drones of the universe of discourse, and the unique identifier of the center planner.

type

1177 CMM = DI-set \times CPI

value

1177 mereo_CM: CM \rightarrow CMM

1178. For the universe of discourse it is the case that
- the drone identifiers of the mereology of a monitor must be exactly those of the drones of the universe of discourse, and
 - the planner identifier of the mereology of a monitor must be exactly that of the planner of the universe of discourse.

axiom

1178 $\forall uod:UoD, e:E, cc:CC, cp:CP, cm:CM, g:G \cdot$
 1178 $e=obs_E(uod) \wedge cc=obs_CC(e) \wedge cp=obs_CP(cc) \wedge cm=obs_CM(cc) \Rightarrow$
 1178 **let** (dis, cpi) = mereo_CM(cm) **in**
 1178a dis = xtr_dis(uod)
 1178b $\wedge cpi = uid_CP(cp)$ **end**

Actuator Mereology

The center actuator's mereology reflects its awareness of the enterprise drones whose position and dynamics it is expected to control.

1179. The mereology of the center actuator is a pair: the set of unique identifiers of the business drones of the universe of discourse, and the unique identifier of the center planner.

type

1179 CAM = EDI-set \times CPI

value

1179 mereo_CA: CA \rightarrow CAM

1180. For all universes of discourse
- the drone identifiers of the mereology of a center actuator must be exactly those of the enterprise drones of the space of interest (of the monitor), and
 - the center planner identifier of the mereology of a center actuator must be exactly that of the center planner of the command center of the space of interest (of the monitor)

axiom

```

1180  $\forall$  uod:UoD,e:E,cc:CC,cp:CP,ca:CA •
1180   e=obs_E(uod) $\wedge$ cc=obs_CC(e) $\wedge$ cp=obs_CP(cc) $\wedge$ ca=obs_CA(cc)  $\Rightarrow$ 
1180     let (dis,cpi) = mereo_CA(ca) in
1180a    dis = tr_dis(e)
1180b     $\wedge$  cpi = uid_CP(cp) end

```

Enterprise Drone Mereology

1181. The mereology of an enterprise drone is the triple of the command center monitor, the command center actuator¹⁶, and the geography.

type

1181 EDM = CMI \times CAI \times GI

value

1181 mereo_ED: ED \rightarrow EDM

1182. For all universes of discourse the enterprise drone mereology satisfies:

- a. the unique identifier of the first element of the drone mereology is that of the enterprise's command monitor,
- b. the unique identifier of the second element of the drone mereology is that of the enterprise's command actuator, and
- c. the unique identifier of the third element of the drone mereology is that of the universe of discourse's geography.

axiom

```

1182  $\forall$  uod:UoD,e:E,cm:CM,ca:CA,ed:ED,g:G •
1182   e=obs_E(uod) $\wedge$ cm=obs_CM(obs_CC(e)) $\wedge$ ca=obs_CA(obs_CC(e))
1182    $\wedge$  ed  $\in$  xtr_Ds(e) $\wedge$ g=obs_G(uod)  $\Rightarrow$ 
1182     let (cmi,cai,gi) = mereo_D(ed) in
1182a    cmi = uid_CMM(ccm)
1182b     $\wedge$  cai = uid_CAI(cai)
1182c     $\wedge$  gi = uid_G(g) end

```

'Other' Drone Mereology

1183. The mereology of an 'other' drone is a pair: the unique identifier of the monitor and the unique identifier of the geography.

type

1183 ODM = CMI \times GI

value

1183 mereo_OD: OD \rightarrow ODM

We leave it to the reader to formulate a suitable axiom, cf. axiom 1182 [Page 412].

¹⁶ The command center monitor and the command center actuator and their unique identifiers will be defined in Items 1160, 1162 [Page 407], 1164 and 1166 [Page 408].

Geography Mereology

1184. The geography mereology is a pair¹⁷ of the unique of the unique identifiers of the planner and the set of all drones.

type

1184 $GM = CPI \times CMI \times DI\text{-set}$

value

1184 mereo_G: $G \rightarrow GM$

We leave it to the reader to formulate a suitable axiom, cf. axiom 1182 [Page 412].

F.2.4 Attributes

We analyse & describe attributes for the following parts: *enterprise drones* and *'other' drones, monitor, planner* and *actuator*, and the *geography*. The attributes, that we shall arrive at, are usually concrete in the sense that they comprise values of, as we shall call them, *constituent* types. We shall therefore first analyse & describe these constituent types. Then we introduce the part attributes as expressed in terms of the constituent types. But first we introduce three notions core notions: time, Sect. F.2.4, positions, Sect. F.2.4, and flight plans, Sect. F.2.4.

The Time Sort

1185. Let the special sort identifier \mathbb{T} denote times

1186. and the special sort identifier \mathbb{TI} denote time intervals.

1187. Let identifier time designate a “magic” function whose invocations yield times.

type

1185 \mathbb{T}

1185 \mathbb{TI}

value

1185 time: **Unit** $\rightarrow \mathbb{T}$

1188. Two times can not be added, multiplied or divided, but subtracting one time from another yields a time interval.

1189. Two times can be compared: smaller than, smaller than or equal, equal, not equal, etc.

1190. Two time intervals can be compared: smaller than, smaller than or equal, equal, not equal, etc.

1191. A time interval can be multiplied by a real number.

Etcetera.

value

1188 $\ominus: \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{TI}$

1189 $\langle, \leq, =, \neq, \geq, \rangle: \mathbb{T} \times \mathbb{T} \rightarrow \mathbf{Bool}$

1190 $\langle, \leq, =, \neq, \geq, \rangle: \mathbb{TI} \times \mathbb{TI} \rightarrow \mathbf{Bool}$

1191 $\otimes: \mathbb{TI} \times \mathbf{Real} \rightarrow \mathbb{TI}$

¹⁷ 30.11.2017: I think !

Positions

Positions (of drones) play a pivotal rôle.

1192. Each *position* being designated by

1193. *longitude*, *latitude* and *altitude*.

type

1193 LO, LA, AL

1192 $P = LO \times LA \times AL$

A Neighbourhood Concept

1194. Two positions are said to be *neighbours* if the *distance* between them is small enough for a drone to fly from one to the other in one to three minutes' time – for drones flying at a speed below Mach 1.

value

1194 neighbours: $P \times P \rightarrow \mathbf{Bool}$

We leave the neighbourhood proposition further undefined.

Flight Plans

A crucial notion of our universe of discourse is that of flight plans.

1195. A *flight plan element* is a pair of a time and a position.

1196. A *flight plan* is a sequence of flight plan elements.

type

1195 $FPE = \mathbb{T} \times P$

1196 $FP = FLE^*$

1197. such that adjacent entries in flight plans

a. record increasing times and

b. neighbouring positions.

axiom

1197 $\forall fp:FP, i:\mathbf{Nat} \cdot \{i, i+1\} \subseteq \mathbf{inds}fp \Rightarrow$

1197 **let** $(t, p) = fp[i], (t', p') = fp[i+1]$ **in**

1197a $t \leq t'$

1197b $\wedge \mathbf{neighbours}(p, p')$

1197 **end**

Enterprise Drone Attributes

Constituent Types

1198. Enterprise drones have *positions* expressed, for example, in terms of *longitude*, *latitude* and *altitude*.¹⁸
1199. Enterprise drones have *velocity* which is a vector of *speed* and three-dimensional, i.e., spatial, *direction*.
1200. Enterprise drones have *acceleration* which is a vector of *increase/decrease of speed per time unit* and *direction*.
1201. Enterprise drones have orientation which is expressed in terms of three quantities: *yaw*, *pitch* and *roll*.¹⁹

We leave *speed*, *direction* and *increase/decrease per time unit* unspecified.

type

- 1198 POS = P
- 1199 VEL = SPEED × DIRECTION
- 1200 ACC = IncrDecrSPEEDperTimeUnit × DIRECTION
- 1201 ORI = YAW × PITCH × ROLL
- 1199 SPEED = ...
- 1199 DIRECTION = ...
- 1200 IncrDecrSPEEDperTimeUnit = ...

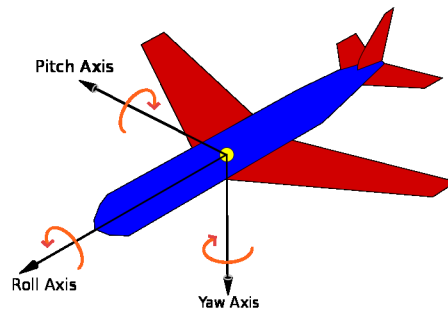


Fig. F.2. Aircraft Orientation

¹⁸ *Longitude* is a geographic coordinate that specifies the east-west position of a point on the Earth's surface. It is an angular measurement, usually expressed in degrees and denoted by the Greek letter lambda. Meridians (lines running from the North Pole to the South Pole) connect points with the same longitude. *Latitude* is a geographic coordinate that specifies the north-south position of a point on the Earth's surface. Latitude is an angle (defined below) which ranges from 0° at the Equator to 90° (North or South) at the poles. Lines of constant latitude, or parallels, run east-west as circles parallel to the equator. *Altitude* or height (sometimes known as depth) is defined based on the context in which it is used (aviation, geometry, geographical survey, sport, and many more). As a general definition, altitude is a distance measurement, usually in the vertical or "up" direction, between a reference datum and a point or object. The reference datum also often varies according to the context.

¹⁹ *Yaw*, *pitch* and *roll* are seen as symmetry axes of a drone: normal axis, lateral (or transverse) axis and longitudinal (or roll) axis. See Fig. F.2 [Page 415].

Attributes

1202. One of the enterprise properties is that of its *dynamics* which is seen as a quadruple of *velocity*, *acceleration*, *orientation* and *position*. It is recorded as a reactive attribute.
1203. Enterprise drones follow a flight course, as prescribed in and recorded as a programmable attribute, referred to a the *future flight plan*, FFP.
1204. Enterprise drones have followed a course recorded, also a programmable attribute, as a *past flight plan list*, PFPL.
1205. Finally enterprise drones “remember”, in the form of a programmable attribute, the *geography* (i.e., the *area*, the *land* and the *weather*) it is flying over and in !

type

- 1205 $\text{ImG} = A \times L \times W$
- 1202 $\text{DYN} = s_{\text{vel}}:\text{VEL} \times s_{\text{acc}}:\text{ACC} \times s_{\text{ori}}:\text{ORI} \times s_{\text{pos}}:\text{POS}$
- 1203 $\text{FPL} = \text{FP}$
- 1204 $\text{PFPL} = \text{FP}^*$

value

- 1202 $\text{attr_DYN}: \text{ED} \rightarrow \text{DYN}$
- 1203 $\text{attr_FPL}: \text{ED} \rightarrow \text{FPL}$
- 1204 $\text{attr_PFPL}: \text{ED} \rightarrow \text{PFPL}$
- 1205 $\text{attr_ImG}: \text{ED} \rightarrow \text{ImG}$

Enterprise, as well as ‘other’ drone, positions must fall within the *Euclidian Point Space* of the geography of the universe of discourse. We leave that as an axiom to be defined – or we could decide that if a drone leaves that space then it is lost, and if drones suddenly “appear, out of the blue”, then they are either “brand new”, or “reappear”.

Enterprise Drone Attribute Categories:

The position, velocity, acceleration, position and past position list attributes belong to the **reactive** category. The future position list attribute belong to the **programmable** category. Drones have a “zillion” more attributes – which may be introduced in due course.

‘Other’ Drones Attributes**Constituent Types**

The constituent types of ‘other’ drones are similar to those of some of the enterprise drones.

Attributes

1206. ‘Other’ drones have *dynamics*, $\text{dyn}:\text{DYN}$.
1207. ‘Other’ drones “remember”, in the form of a programmable attribute, the *immediate geography*, ImG (i.e., the *area*, the *land* and the *weather*) it is flying over and in !

type

- 1207 A, L, W
- 1207 $\text{ImG} = A \times L \times W$

value

- 1206 $\text{attr_DYN}: \text{OD} \rightarrow \text{DYN}$
- 1207 $\text{attr_ImG}: \text{OD} \rightarrow \text{ImG}$

Drone Dynamics

1208. By a timed drone dynamics, TiDYN, we understand a quadruplet of *time*, *position*, *dynamics* and *immediate geography*.
1209. By a *current drone dynamics* we shall understand a drone identifier-indexed set of timed drone dynamics.
1210. By a *record of [traces of] timed drone dynamics* we shall understand a drone identifier-indexed set of sequences of timed drone dynamics.

type

- 1208 $TiDYN = \mathbb{T} \times POS \times DYN \times ImG$
 1209 $CuDD = (EDI \xrightarrow{m} TiDYN) \cup (ODI \xrightarrow{m} TiDYN)$
 1210 $RoDD = (EDI \xrightarrow{m} TiDYN^*) \cup (ODI \xrightarrow{m} TiDYN^*)$

We shall use the notion of *current drone dynamics* as the means whereby the *monitor* ascertains (obtains, by interacting with drones) the dynamics of drones, and the notion of a *record of [traces of] drone dynamics* in the *monitor*.

Drone Positions

1211. For all drones whether enterprise or ‘other’, their positions must lie within the geography of their universe of discourse.

axiom

- 1211 $\forall uod:UoD,e:E,g:G,d:(ED|OD) \cdot$
 1211 $e = obs_E(uod) \wedge g = obs_G(uod) \wedge d \in xtr_Ds(uod) \Rightarrow$
 1211 $\quad \text{let } eps = attr_EPS(g), (_,_ ,p) = attr_DYN(d) \text{ in } p \in eps \text{ end}$

Monitor Attributes

The *monitor* “sits between” the *drones* whose dynamics it monitors and the *planner* which it provides with records of drone dynamics. Therefore we introduce the following.

1212. The monitor has just one, a programmable attribute: a trace of the most recent and all past time-stamped recordings of the dynamics of all drones, that is, an element $rodd:RoDD$, cf. Item 1210 [Page 416].

type

- 1212 $MRODD = RoDD$

value

- 1212 $attr_MRODD: CM \rightarrow MRODD$

The monitor “obtains” current drone dynamics, $cudd:CuDD$ (cf. Item 1209 [Page 416]) from the *drones* and offers records of [traces of] drone dynamics,(cf. Item 1210 [Page 416]) $rodd:RoDD$, to the *planner*.

Planner Attributes

Swarms and Businesses:

The *planner* is where all decisions are made with respect to where enterprise drones should be flying; which enterprise drones fly together, which no longer – (with this notion of “flying together” leading us to the concept of *swarms*); which swarms of enterprise drones do which kinds of work – (with this notion of work specialisation leading us to the concept of businesses.)

1213. The is a notion of a *business identifier*, BI.

type
1213 BI

Planner Directories:

Planners have three directories. These are attributes, BDIR (businesses), SDIR (swarms) and DDIR (drones).

- 1214. BDIR records which swarms are resources of which businesses;
- 1215. SDIR records which drones “belong” to which swarms.
- 1216. DDIR “keeps track” of past and present enterprise drone positions, as per enterprise drone identifier.
- 1217. We shall refer to this triplet of directories by TDIR

type
 1214 $BDIR = BI \xrightarrow{m} SI\text{-set}$
 1215 $SDIR = SI \xrightarrow{m} DI\text{-set}$
 1216 $DDIR = DI \xrightarrow{m} RoDD$
 1217 $TDIR = BDIR \times SDIR \times DDIR$
value
 1214 $attr_BDIR: CP \rightarrow BDIR$
 1215 $attr_SDIR: CP \rightarrow SDIR$
 1216 $attr_DDIR: CP \rightarrow DPL$

All three directories are *programmable attributes*.

The business swarm concept can be visualized by grouping together drones of the same swarm in the visualiarion of the aggregate set of enterprise drones. Figure F.3 [Page 418] attempts this visualization.

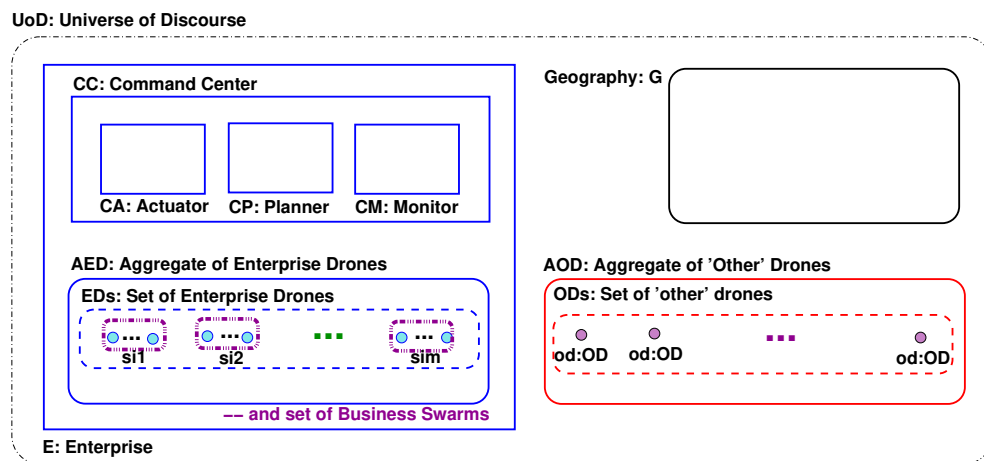


Fig. F.3. Conceptual Swarms of the Universe of Discourse

- 1218. For the planners of all universes of discourse the following must be the case.
 - a. The swarm directory must
 - i. have entries for exactly the swarms of the business directory,

- ii. define disjoint sets of enterprise drone identifiers, and
 - iii. these sets must together cover all enterprise drones.
- b. The drone directory must record the present position, the past positions, a list, dpl:DPL, and, besides satisfying axioms 1211, satisfy some further constraints:
- i. they must list exactly the drone identifiers of the aggregate of enterprise drones, and the sum total of its enterprise drone identifiers must be exactly those of the enterprise drones aggregate of enterprise swarms, and
 - ii. the head of a drone's present and past position list must similarly be within reasonable distance of that drone's current position.

axiom

```

1218   $\forall$  uod:UpD,e:E,cp:CP,g:G •
1218  e=obs_E(uod) $\wedge$ cp=obs_CP(obs_CC(e))  $\Rightarrow$ 
1218a  let (bdir,sdir,ddir) = (attr_BDIR,attr_SDIR,attr_DDIR)(cp) in
1218(a)i   $\cup$  rng bdir = dom sdir
1218(a)ii  $\wedge \forall si,si' \{si,si'\} \subseteq \mathbf{dom} \text{ sdir} \wedge si \neq si' \Rightarrow$ 
1218(a)ii   sdir(si)  $\cap$  sdir(si') = {}
1218(a)iii  $\wedge \cup$  rng sdir = xtr_dis(e)
1218(b)i   $\wedge \mathbf{dom} \text{ ddir} = \text{xtr\_dis}(e)$ 
1218(b)ii  $\wedge \forall di:DI \cdot di \in \mathbf{dom} \text{ ddir}$ 
1218(b)ii   let (d,dpl) = (attr_DDIR(cp))(di) in
1218(b)ii   dpl  $\neq \langle \rangle$ 
1218(b)ii    $\Rightarrow$  neighbours(f,hd(dpl))
1218(b)ii    $\wedge$  neighbours(hd(dpl),
1218(b)ii   attr_EDPOS(xtr_D(obs_Ss(e))(di)))
1218  end end

```

Actuator Attributes

The actuator receives, from the planner, flight directives as to which enterprise drones should be redirected. The actuator maintains a record of most recent and all past such flight directives. Finally, the actuator, effects the directives by informing designated enterprise drones as to their next flight plans.

1219. Actuators have one programmable attribute: a flight directive directory. It lists, for each enterprise drone, by identifier, a pair: its current flight plan and a list of past flight plans.

type

1219 FDDIR = EDI \rightarrow_m (FP \times FP*)

value

1219 attr_FDDIR: CA \rightarrow FDDIR

Geography Attributes**Constituent Types:**

The constituent types of *longitude*, *latitude* and *altitude* and *positions*, of a *geography*, were introduced in Items 1153.

1220. A further concept of geography is that of *area*.

1221. An area, a:A, is a subset of positions within the geography.

type1220 $A = P\text{-infset}$ **axiom**1221 $\forall uod:UoD, g:G, a:A \cdot g = \text{obs}_G(uod) \Rightarrow a \subseteq \text{attr_EPS}(g)$ **Attributes**

1222. Geographies have, as one of their attributes, a *Euclidian Point Space*, in this case, a *compact*²⁰ infinite set of three-dimensional positions.

type1222 $\text{EPS} = P\text{-infset}$ **value**1222 $\text{attr_EPS}: G \rightarrow \text{EPS}$

Further geography attributes reflect the “lay of the land and the weather right now!”.

1223. The “lay of the land”, L is a “conglomerate” further undefined geodetics and cadestra²¹

1224. The “weather” W is another “conglomerate” of temperature, humidity, precipitation, air pressure, etc.

type1223 L 1224 W **value**1223 $\text{attr}_L: G \rightarrow L$ 1224 $\text{attr}_W: G \rightarrow W$ **F.3 Operations on Universe of Discourse States**

Before we analyse & describe perdurants let us take a careful look at the actions that drone and swarm behaviours may take. We refer to this preparatory analysis & description as one of analysing & describing the state operations. From this analysis & description we move on to the analysis & description of behaviours, events and actions. The idea is to be able to prove some relations between the two analyses & descriptions: the state operation and the behaviour analyses & descriptions. We refer to [75, Sects. 2.3 and 2.5].

F.3.1 The Notion of a State

A state is any subset of parts each of which contains one or more dynamic attributes. Following are examples of states of the present case study: a space of interest, an aggregate of ‘business’ swarms, an aggregate of ‘other’ swarms, a pair of the aggregates just mentioned, a swarm, or a drone.

F.3.2 Constants

Some quantities of a given universe of discourse are constants. Examples are the unique identifiers of the:

²⁰ In mathematics, and more specifically in general topology, compactness is a property that generalizes the notion of a subset of Euclidean space being closed (that is, containing all its limit points) and bounded (that is, having all its points lie within some fixed distance of each other). Examples include a closed interval, a rectangle, or a finite set of points.

²¹ land surface altitude, streets, buildings (tall or not so tall), power lines, etc.

1225. enterprise, e_i ;
 1226. aggregate of 'other' drones, oi ;
 1227. geography, g_i ;
 1228. command center, cc_i ;
 1229. monitor, cm_i ;
1230. planner, cp_i ;
 1231. actuator, ca_i ;
 1232. set of 'other' drones, od_{is} ;
 1233. set of enterprise drones, ed_{is} ;
 1234. and the set of all drones, ad_{is} .

value

- 1225 $aed_i:EI = uid_AED(obs_AED(uod))$
 1226 $aod_i:OI = uid_AOD(obs_AOD(uod))$
 1227 $g_i:GI = uid_G(obs_G(uod))$
 1228 $cc_i:CCI = uid_CC(obs_CC(obs_AED(uod)))$
 1229 $cm_i:CMI = uid_CM(obs_CM(obs_CC(obs_AED(uod))))$
 1230 $cp_i:CPI = uid_CP(obs_CP(obs_CC(obs_AED(uod))))$
 1231 $ca_i:CAI = uid_CA(obs_CA(obs_CC(obs_AED(uod))))$
 1232 $od_{is}:ODIs = xtr_dis(obs_AOD(uod))$
 1233 $ed_{is}:EDIs = xtr_dis(obs_AED(uod))$
 1234 $ad_{is}:DI\text{-}set = od_{is} \cup ed_{is}$

F.3.3 Operations

An operation is a function from states to states. Following are examples of operations of the present case study: a drone *transfer*: leaving a swarm to join another swarm, a drone *changing course*: an enterprise drone changing course, a swarm *split*: a swarm splitting into two swarms, and swarm *join*: two swarms joining to form one swarm.

A Drone Transfer

1235. The *transfer* operator specifies two distinct and unique identifiers, si , si' , of two enterprise swarms, and the unique identifier, di , of an enterprise drone – all of the same universe of discourse. The *transfer* operation further takes a universe of discourse and yields a universe of discourse as follows:
 1236. The input argument 'from' and 'to' swarm identifiers are different.
 1237. The initial and the final state aggregates of enterprise drones, 'other' drones and geographies are unchanged.
 1238. The initial and final state monitors and actuators are unchanged.
 1239. The business and the drone directors of the initial and final planner are unchanged.
 1240. The 'from' and 'to' input argument swarm identifiers are in the swarm directory and the input argument drone identifiers is in the initial swarm directory entry for the 'from' swarm identifier.
 1241. The input argument drone identifier is in final the swarm directory entry for the 'to' swarm identifier.
 1242. And the final swarm directory is updated ...

value

- 1235 $transfer: DI \times SI \times SI \rightarrow UoD \xrightarrow{\sim} UoD$
 1235 $transfer(di, fsi, tsi)(uod) \text{ as } uod'$
 1236 $fsi \neq tsi \wedge$
 1235 **let** $aed = obs_AED(uod)$, $aed' = obs_AED(uod')$, $g = obs_G(uod)$, $g' = obs_G(uod')$ **in**
 1235 **let** $cc = obs_CC(aed)$, $cc' = obs_CC(aed')$, $aod = obs_AOD(uod)$, $aod' = obs_AOD(uod')$ **in**
 1235 **let** $cm = obs_CM(cc)$, $cm' = obs_CM(cc')$, $cp = obs_CP(cc)$, $cp' = obs_CP(cc')$ **in**
 1235 **let** $ca = obs_CA(cc)$, $ca' = obs_CA(cc')$ **in**
 1235 **let** $bdir = attr_BDIR(cc)$, $bdir' = attr_BDIR(cc')$,
 1235 $sdir = attr_SDIR(cc)$, $sdir' = attr_SDIR(cc')$,

```

1235     ddir = attr_DDIR(cc), ddir' = attr_DDIR(cc') in
1237     post: aed = aed' ∧ aod = aod' ∧ g = g' ∧
1238           cm = cm' ∧ ca = ca' ∧
1239           bdir = bdir' ∧ ddir = ddir'
1240     pre {fsi,tsi} ⊆ dom sdir ∧ di ∈ sdir(fsi)
1241     post di ∉ sdir(fsi') ∧ di ∈ sdir(tsi') ∧
1242           sdir' = sdir † [fsi→sdir(fsi) ∪ di] † [tsi→sdir(tsi)\di]
1235     end end end end end

```

An Enterprise Drone Changing Course

TO BE WRITTEN

A Swarm Splitting into Two Swarms

TO BE WRITTEN

Two Swarms Joining to form One Swarm

TO BE WRITTEN

Etcetera

TO BE WRITTEN

F.4 Perdurants

We observe that the term *train* can have the following “meanings”: the *train*, as an *endurant*, parked at the railway station platform, i.e., as a *composite part*; the *train*, as a *perdurant*, as it “speeds” down the railway track, i.e., as a *behaviour*; the *train*, as an *attribute*. This observation motivates that we “magically”, as it were, introduce a **COMPILEr** function, cf. [70, Sect. 4]

F.4.1 System Compilation

The **COMPILEr** function “worms” its way, so-to-speak, “down” the “hierarchy” of parts, from the universe of discourse, via its immediate sup-parts, and from these to their sub-parts, and so on, until the **COMPILEr** reaches atomic parts. We shall henceforth do likewise.

The Compile Functions

1243. Compilation of a *universe of discourse* results in
- a. the RSL-**Text** of the *core* of the universe of discourse behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
 - b. followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise,
 - c. followed by the RSL-**Text** of the parallel composition of the compilation of the geography,
 - d. followed by the RSL-**Text** of the parallel composition of the compilation of the aggregate of ‘other’ drones.

- 1243 $\text{COMPILE}_{UoD}(uod) \equiv$
 1243a $\mathcal{M}_{uid_UoD}(uod)(\text{mereo_UoD}(uod), \text{sta}(uod))(\text{pro}(uod))$
 1243b $\parallel \text{COMPILE}_{AED}(\text{obs_AED}(uod))$
 1243c $\parallel \text{COMPILE}_G(\text{obs_G}(uod))$
 1243d $\parallel \text{COMPILE}_{AOD}(\text{obs_AOD}(uod))$

1244. Compilation of an *enterprise* results in
- the RSL-**Text** of the *core* of the enterprise behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
 - followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise aggregate of enterprise drones,
 - followed by the RSL-**Text** of the parallel composition of the compilation of the enterprise command center.

- 1244 $\text{COMPILE}_{AED}(e) \equiv$
 1244a $\mathcal{M}_{uid_AED}(e)(\text{mereo_E}(e), \text{sta}(e))(\text{pro}(e))$
 1244b $\parallel \text{COMPILE}_{EDs}(\text{obs_EDs}(e))$
 1244c $\parallel \text{COMPILE}_{CC}(\text{obs_CC}(e))$

1245. Compilation of an *enterprise aggregate of enterprise drones* results in
- the RSL-**Text** of the *core* of the aggregate behaviour (which we set to **skip** – allowing us to ignore *core* arguments),
 - followed by the RSL-**Text** of the parallel composition of the distributed compilation of the enterprise aggregate's set of enterprise drones.

- 1245 $\text{COMPILE}_{EDs}(es) \equiv$
 1245a $\mathcal{M}_{uid_EDs}(es)(\text{mereo_EDS}(es), \text{sta}(es))(\text{pro}(es))$
 1245b $\parallel \{ \text{COMPILE}_{ED}(\text{ed}) \mid \text{ed} : \text{ED} \cdot \text{ed} \in \text{obs_EDs}(s) \}$

1246. Compilation of an *enterprise drone* results in
- the RSL-**Text** of the *core* of the enterprise drone behaviour – which is what we really wish to express – and since enterprise drones are here considered atomic, that is where the compilation of enterprise ends.

- 1246 $\text{COMPILE}_{ED}(\text{ed}) \equiv$
 1246a $\mathcal{M}_{uid_ED}(\text{ed})(\text{mereo_ED}(\text{ed}), \text{sta}(\text{ed}))(\text{pro}(\text{ed}))$

1247. Compilation of an *aggregate of 'other' drones* results in
- the RSL-**Text** of the *core* of the aggregate 'other' drones behaviour (which we set to **skip** – allowing us to ignore *core* arguments) –
 - followed by the RSL-**Text** of the parallel composition of the distributed compilation of the 'other' drones in the 'other' drones' aggregate set of 'other' drones.

- 1247 $\text{COMPILE}_{AOD}(\text{aod}) \equiv$
 1247a $\mathcal{M}_{uid_OD}(\text{od})(\text{mereo_S}(\text{ods}), \text{sta}(\text{ods}))(\text{pro}(\text{ods}))$
 1247b $\parallel \{ \text{COMPILE}_{OD}(\text{od}) \mid \text{od} : \text{OD} \cdot \text{od} \in \text{obs_ODs}(\text{ods}) \}$

1248. Compilation of a(n) *'other' drone* results in

- a. the RSL-**Text** of the *core* of the ‘other’ drone behaviour – which is what we really wish to express – and since ‘other’ drones are here considered atomic, that is where the compilation of the ‘other’ drones aggregate

1248a $\text{COMPILE}_{\{OD\}}(\text{ed}) \equiv$

1248a $\mathcal{M}_{\text{uid_OD}}(\text{od})(\text{mereo_OD}(\text{od}), \text{sta}(\text{od}))(\text{pro}(\text{od}))$

1249. Compilation of an atomic *geography* results in

- a. the RSL-**Text** of the *core* of the geography behaviour.

1249 $\text{COMPILE}_G(\text{g}) \equiv$

1249a $\mathcal{M}_{\text{uid_G}}(\text{g})(\text{mereo_G}(\text{g}), \text{sta}(\text{g}))(\text{pro}(\text{g}))$

1250. Compilation of a composite *command center* results in

- a. the RSL-**Text** of the *core* of the command center behaviour (which we set to **skip** – allowing us to ignore *core* arguments)
 b. followed by the RSL-**Text** of the parallel composition of the compilation of the command monitor,
 c. followed by the RSL-**Text** of the parallel composition of the compilation of the command planner,
 d. followed by the RSL-**Text** of the parallel composition of the compilation of the command actuator.

1250 $\text{COMPILE}_M(\text{cc}) \equiv$

1250a $\mathcal{M}_{\text{uid_CC}}(\text{cc})(\text{mereo_CC}(\text{cc}), \text{sta}(\text{cc}))(\text{pro}(\text{cc}))$

1250b $\parallel \text{COMPILE}_{CC}(\text{obs_CM}(\text{cc}))$

1250c $\parallel \text{COMPILE}_{CP}(\text{obs_CP}(\text{cc}))$

1250d $\parallel \text{COMPILE}_{CA}(\text{obs_CA}(\text{cc}))$

1251. Compilation of an atomic *command monitor* results in

- a. the RSL-**Text** of the *core* of the monitor behaviour.

1251 $\text{COMPILE}_{CM}(\text{cm}) \equiv$

1251a $\mathcal{M}_{\text{uid_CM}}(\text{cm})(\text{mereo_CM}(\text{cm}), \text{sta}(\text{cm}))(\text{pro}(\text{cm}))$

1252. Compilation of an atomic *command planner* results in

- a. the RSL-**Text** of the *core* of the planner behaviour.

1252 $\text{COMPILE}_{CP}(\text{cp}) \equiv$

1252a $\mathcal{M}_{\text{uid_CP}}(\text{cp})(\text{mereo_CP}(\text{cp}), \text{sta}(\text{cp}))(\text{pro}(\text{cp}))$

1253. Compilation of an atomic *command actuator* results in

- a. the RSL-**Text** of the *core* of the actuator behaviour.

1253 $\text{COMPILE}_{CA}(\text{ca}) \equiv$

1253a $\mathcal{M}_{\text{uid_CA}}(\text{ca})(\text{mereo_CA}(\text{ca}), \text{sta}(\text{ca}))(\text{pro}(\text{ca}))$

Some CSP Expression Simplifications

We can justify the following CSP simplifications [147, 150, 225, 233]:

1254. **skip** in parallel with any CSP expression csp is csp .
 1255. The distributed parallel composition of the distributed parallel composition of CSP expressions, $csp(i,j)$, i indexed over I , i.e., $i:I$, and $j:J$ respectively, is the distributed parallel composition over CSP expressions, $csp(i,j)$, i.e., indexed over $(i,j):I \times J$ – where the index sets $iset$ and $jset$ are assumed.

axiom

- 1255 **skip** $\parallel csp \equiv csp$
 1255 $\parallel \{ \parallel \{ csp(i,j) \mid i:I \in iset \} \mid j:J \in jset \} \equiv \parallel \{ csp(i,j) \mid i:I, j:J \cdot i \in I\text{-set} \wedge j \in J\text{-set} \}$

The Simplified Compilation

1256. The simplified compilation results in:

- 1256 **COMPILE**(uod) \equiv
 1246a $\{ \mathcal{M}_{uid_ED}(ed)(mereo_ED(ed),sta(ed))(pro(ed))$
 1246a $\mid ed:ED \cdot ed \in xtr_Ds(obs_AED(uod)) \}$
 1248a $\parallel \{ \mathcal{M}_{uid_OD}(od)(mereo_OD(od),sta(od))(pro(od))$
 1248a $\mid od:OD \cdot od \in xtr_ODs(obs_AOD(uod)) \}$
 1249a $\parallel \mathcal{M}_{uid_G}(g)(mereo_G(g),sta(g))(pro(g))$
 1249a **where** $g \equiv obs_G(uod)$
 1251a $\parallel \mathcal{M}_{uid_CM}(cm)(mereo_CM(cm),sta(cm))(pro(cm))$
 1251a **where** $cm \equiv obs_CM(obs_CC(obs_E(uod)))$
 1252a $\parallel \mathcal{M}_{uid_CP}(cp)(mereo_CP(cp),sta(cp))(pro(cp))$
 1252a **where** $cp \equiv obs_CP(obs_CC(obs_E(uod)))$
 1253a $\parallel \mathcal{M}_{uid_CA}(ca)(mereo_CA(ca),sta(ca))(pro(ca))$
 1253a **where** $ca \equiv obs_CA(obs_CC(obs_E(uod)))$

1257. In Item 1256's Items 1246a, 1248a, 1249a, 1251a, 1252a, and 1253a we replace the “anonymous” behaviour names \mathcal{M} by more meaningful names.

- 1257 **COMPILE**(uod) \equiv
 1246a $\{ enterprise_drone_{uid_ED}(ed)(mereo_ED(ed),sta(ed))(pro(ed))$
 1246a $\mid ed:ED \cdot ed \in xtr_Ds(obs_AED(uod)) \}$
 1248a $\parallel \{ other_drone_{uid_OD}(od)(mereo_OD(od),sta(od))(pro(od))$
 1248a $\mid od:OD \cdot od \in xtr_ODs(obs_AOD(uod)) \}$
 1249a $\parallel geography_{uid_G}(g)(mereo_G(g),sta(g))(pro(g))$
 1249a **where** $g \equiv obs_G(uod)$
 1251a $\parallel monitor_{uid_CM}(cm)(mereo_CM(cm),sta(cm))(pro(cm))$
 1251a **where** $cm \equiv obs_CM(obs_CC(obs_E(uod)))$
 1252a $\parallel planner_{uid_CP}(cp)(mereo_CP(cp),sta(cp))(pro(cp))$
 1252a **where** $cp \equiv obs_CP(obs_CC(obs_E(uod)))$
 1253a $\parallel actuator_{uid_CA}(ca)(mereo_CA(ca),sta(ca))(pro(ca))$
 1253a **where** $ca \equiv obs_CA(obs_CC(obs_E(uod)))$

F.4.2 An Early Narrative on Behaviours

Either Endurants or Perdurants, Not Both!

First the reader should observe that the manifest parts, in some sense, do no longer “exist”! They have all been replaced by their corresponding behaviours. These behaviours embody all the qualities of their “origin”: the unique identifiers, the mereology, and all the attributes – the latter in one form or another: the static attributes as constants (referred to in the bodies of the behaviour definitions); the programmable attributes as arguments (“‘carried over’” from one invocation to the next); and the remaining dynamic attributes as “inputs” (whose varying values are “‘accessed’” through [dynamic attribute] channels).

Focus on Some Behaviours, Not All!

Secondly we focus, in this case study, only on the behaviour of the *planner*. The other behaviours, the ‘other’ *drones*, *enterprise drones*, *monitor*, *actuator*, and the *geography*, are, in this case study of less interest to us. That is, other case studies could focus on the behaviours of *drones*, or *geographies*, or *monitor*, or *actuator*.

The Behaviours – a First Narrative

Drones “continuously” offer their identified dynamics (location, velocity, and possibly more) **to** the *monitor*. *Enterprise drones* “continuously”, and in addition, offers to accept flight guidance **from** the *actuator*. The *monitor* “continuously sweeps” the air space and collects the identities of all recognizable drones and their dynamics, and offers this **to** the *planner*. The *planner* does all the interesting work! It effects the *allocation/reallocation* of drones to/from business swarms; it *calculates enterprise drone flights* and instructs the *actuator* to offer such flight plans to relevant drones; etcetera! Finally the *actuator*, as instructed by the *planner*, offers flight guidance, as per instructions **from** the *planner*, **to** all or some *enterprise drones*.

F.4.3 Channels

Channels is a concept of CSP [147, 149, 150].

CSP channels are a means for synchronising behaviours and for communicating values between synchronised behaviours, as well as, as a technicality, conveying values of most kinds of dynamic attributes of parts (i.e., endurants) to “their” behavioural counterparts.

There are thus two starting point for the analysis & description of channels: the mereologies and the dynamic attributes of parts. Here we shall single out the following parts and behaviours: the command *monitor*, *planner* and *actuator*, the *enterprise drones* and the ‘other’ *drones*, and the *geography*. We refer to Fig. F.4 [Page 427], a slight “refinement” of Fig. F.1 [Page 404].

The Part Channels

General Remarks:

Let there be given a *universe of discourse*. Let us analyse the *unique identifiers* and the *mereologies* of the *planner* cp : (cpi, cpm) , *monitor* cm : (cmi, cmm) and *geography* g : (mi, mm) , where $cpm = (cai, cmi, gi)$, $cmm = (\{di_1, di_2, \dots, di_n\}, cpi)$ and $gm = (cpi, \{di_1, di_2, \dots, di_n\})$.

We now interpret these facts. When the *planner mereology* specifies the unique identifiers of the *actuator*, the *monitor*, and the *geography*, then that shall mean there there is a way of communicating messages between the actuator, and the geography, and one side, and the planner on the other side.

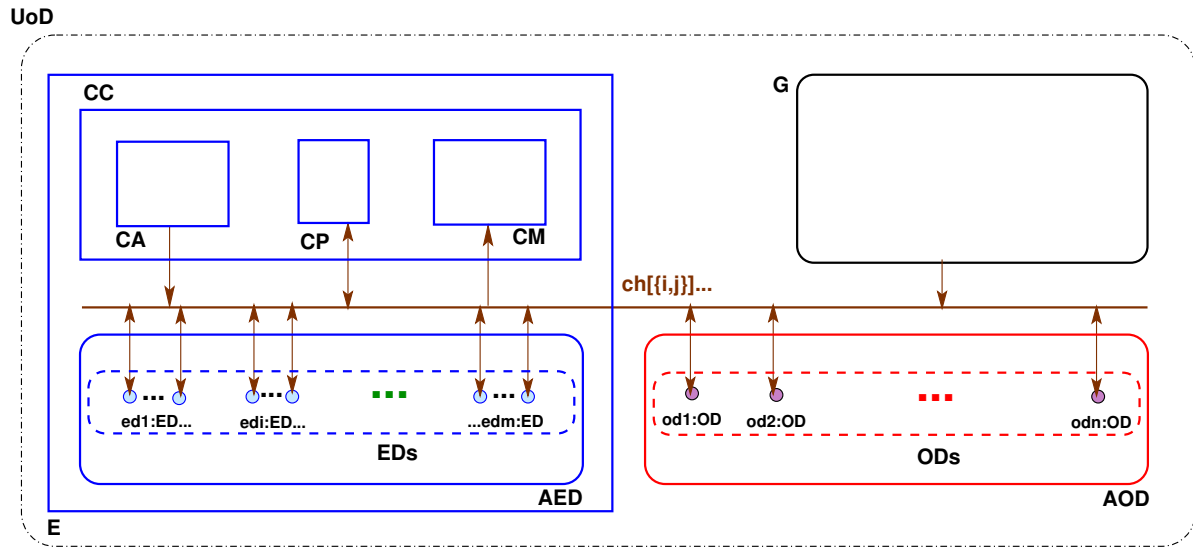


Fig. F.4. Universe of Discourse with General Channel: $ch[\{i,j\}] \dots$

1258. We shall therefore, in a first step of specification development, think of a “grand” array channel over which all communication between behaviours take place. See Fig. F.4 [Page 427].

1259. Example indexes into this array channel are shown in the formulas just below.

type
 1258 MSG
channel
 1258 $\{ch[fui,tui]|fui,tui:PI \cdot \dots\}:MSG$
value
 1259 $ch[cpi,cai]!msg$ *output from planner to actuator.*
 1259 $ch[cpi,cai]?$ *input from planner to actuator.*
 1259 etc.

We presently leave the type of messages, MSG, that can be communicated over this “grand” channel further unspecified. We also leave unspecified the pair of distinct unique identifiers that index the channel array. We emphasize that the uniqueness of all part identifiers allow us to use pairs of such as indices. Expression $ch[fui,tui]!$,sg thus expresses *output from* behaviour indexed by *fuit* to behaviour indexed by *tui*, whereas expression $ch[tui,fui]?$ thus expresses *input from* behaviour indexed by *tui* to behaviour indexed by *fui*. Not all combinations of unique identifiers are needed. The channel array is “sparse”! That property allows us to refine the “grand” channel into the channels illustrated on Fig. F.5 [Page 428]. Some channels are array channels: The channels to the drones whether all drones, or just the enterprise drones. Other channels are “single” channels: these are the channels which are anchored in parts with a priori known, i.e., constant unique identifiers.

Part Channel Specifics

1260. There is an array channel, $d_cm_ch[di,cm_i]:D_CM_MSG$, from any *drone* ($[di]$) behaviour to the *monitor* behaviour (whose unique identifier is cm_i). The channel, as an array, forwards the current drone dynamics $D_CM_MSG = CuDD$.

type

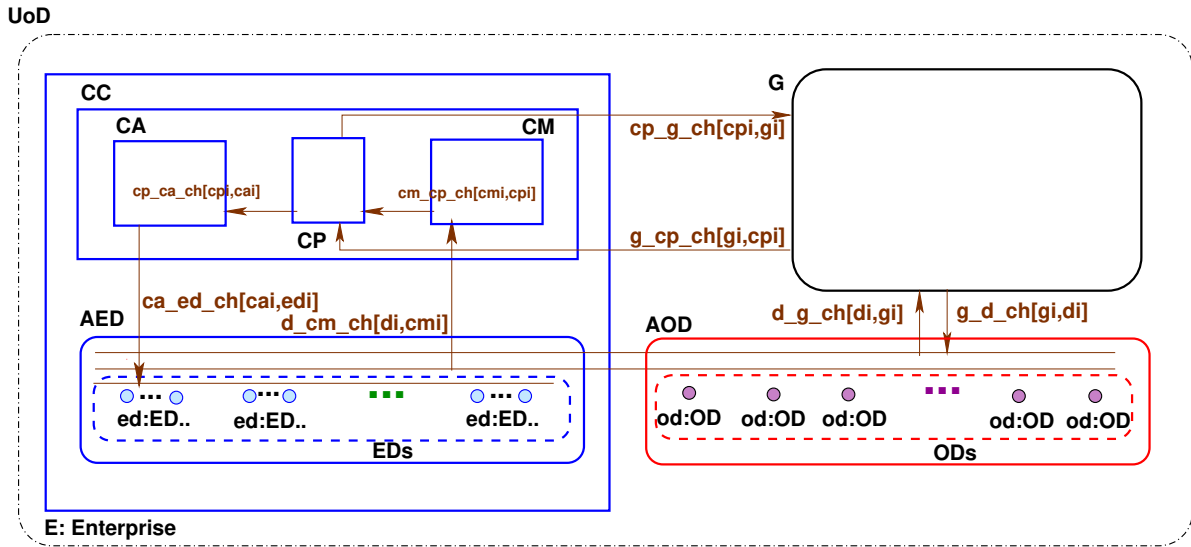


Fig. F.5. Universe of Discourse with Specific Channels

1260 $D_CM_MSG = CuDD$

channel

1260 $\{d_cm_ch[di,cm_i] | di:(EDI|ODI) \cdot di \in dis\}:D_CM_MSG$

1261. There is a channel, $cm_cp_ch[cm_i,cp_i]$, from the *monitor* behaviour (cm_i) to the *planner* behaviour (cp_i). It forwards the monitor's records of drone dynamics $CM_CP_MSG = MRoDD$.

type

1261 $CM_CP_MSG = MRoDD$

1261 **channel** $m_cp_ch[cm_i,cp_i]:CM_CP_MSG$

1262. There is a channel, $cp_ca_ch[cp_i,ca_i]:CP_CA_MSG$, from the *planner* behaviour (cp_i) to the *actuator* behaviour (ca_i). It forwards flight plans $CP_CA_MSG = FP$.

type

1262 $CP_CA_MSG = EID \xrightarrow{m} FP$

channel

1262 $cp_ca_ch[cp_i,ca_i]:CM_CP_MSG$

1263. There is an array channel, $ca_ed_ch[ca_i,edi]$, from the *actuator* behaviour (ca_i) to the *enterprise drone* behaviours (edi for suitable edis). It forwards flight plans, $CA_ED_MSG = FP$, to enterprise drones in a designated set.

type

1263 $CA_ED_MSG = EID \times FP$

channel

1263 $\{ca_ed_ch[ca_i,edi] | edi:EDI \cdot edi \in edis\}:CA_ED_MSG$

1264. There is an array channel, $g_d_ch[di,gi]:D_G_MSG$, from all the *drone* behaviours (di) to the *geography* behaviour. The channels convey, requests for an *immediate geography* for and around a *point*: $D_G_MSG = P$.

type

1264 D_G_MSG = P

channel

1264 {d_g_ch[di,gi]|di:(EDI|ODI)•di ∈ dis}:D_H_MSG

1265. There is an array channel, g_d_ch[gi,di]:G_D_MSG, from the *geography* behaviour to all the *drone* behaviours. The channels convey, for a requested *point*, the immediate geography for that area: G_D_MSG = ImG.

type

1265 G_D_MSG = ImG

channel

1265 {g_d_ch[gi,di]|di:(EDI|ODI)•di ∈ dis}:G_D_MSG

Attribute Channels, General Principles

Some of the drone attributes are *reactive*. Being reactive means that their values change surreptitiously. In the physical world of parts that means that these values must be measured, or somehow ascertained, whenever needed, i.e., “on the fly”. Now “our world” is that of a domain description. When dealing with endurants, the value of an attribute, a:A, of part p:P, is expressed as attr_A(p). When dealing with perdurants, that same value is to be expressed as attr_A_ch[uid_P(p)]?

1266. This means that we must declare a channel for each part with one or more *dynamic*, however not including *programmable*, attributes A1, A2, ..., An.

channel

1266 attr_A1_ch[p_i]:A1, attr_A2_ch[p_i]:A2, ..., attr_An_ch[p_i]:An

1267. If there are several parts, p1,p2,...,pm:P then an array channel over indices p1_i,p2_i,...,pm_i is declared for each applicable attribute.

channel

1267 {attr_A1_ch[p_j_i]|p_j_i:PI•p_j_i∈{p1_i,p2_i,...,pm_i}}:A1,

1267 {attr_A2_ch[p_j_i]|p_j_i:PI•p_j_i∈{p1_i,p2_i,...,pm_i}}:A2,

1267 ...

1267 {attr_An_ch[p_j_i]|p_j_i:PI•p_j_i∈{p1_i,p2_i,...,pm_i}}:An

The Case Study Attribute Channels**‘Other’ Drones:**

‘Other’ drones have the following not biddable or programmable dynamic channels:

1268. dynamics, including velocity, acceleration, orientation and position,
{attr_DYN_ch[odi]:DYN|odi:ODI•odi ∈ odis}.

channel

1268 {attr_DYN_ch[odi]:DYN|odi:ODI•odi ∈ odis}

Enterprise Drones:

Enterprise drones have the following not biddable or programmable dynamic channels:

1269. dynamics, including velocity, acceleration, orientation and position,
 $\{\text{attr_DYN_ch}[\text{edi}]:\text{DYN}|\text{edi}:\text{EDI}\cdot\text{edi}\in\text{odis}\}$.

channel

1269 $\{\text{attr_DYN_ch}[\text{odi}]:\text{DYN}|\text{odi}:\text{ODI}\cdot\text{odi}\in\text{odis}\}$

Geography:

The geography has the following not biddable or programmable dynamic channels:

1270. land, $\text{attr_L_ch}[g_i]:L$, and
 1271. weather, $\text{attr_W_ch}[g_i]:W$.

channel

1270 $\text{attr_L_ch}[g_i]:L$
 1271 $\text{attr_W_ch}[g_i]:W$

We do not show any graphics for the attribute channels.

F.4.4 The Atomic Behaviours

TO BE WRITTEN

Monitor Behaviour

1272. The signature of the monitor behaviour
- a. lists the monitor's unique identifier, carries the monitor's mereology, has no static arguments (... maybe ...), has the programmable time-stamped recordings, dtp , of all drone positions (present and past) and
 - b. further designates the **input** channel $\text{d_cm_ch}[*.*]$ from all drones and the channel **output** $\text{cm_cp_ch}[\text{cmi},\text{cpi}]$ to the planner.
1273. The monitor [otherwise] behaves as follows:
- a. All drones provide as input, $\text{d_cm_ch}[\text{di},\text{cmi}]?$, their time-stamped positions, rec .
 - b. The programmable mrodd attribute is updated, mrodd' , to reflect the latest time stamped dynamics per drone identifier.
 - c. The updated attribute is provided to the planner.
 - d. Then the monitor resumes being the monitor, forwarding, as the programmable attribute, the time-stamped drone position recording.

value

1272a $\text{monitor}:\text{cmi}:\text{CMI}\times\text{cmm}:(\text{dis}:\text{DI}\text{-}\text{set}\times\text{cpi}:\text{CPI})\rightarrow\text{MRoDD}\rightarrow$
 1272b $\text{in}\{\text{d_cm_ch}[\text{di},\text{cmi}]|\text{di}:\text{DI}\cdot\text{di}\in\text{dis}\}\text{out}\text{cm_cp_ch}\text{Unit}$
 1273 $\text{monitor}(\text{mi},(\text{dis},\text{cpi}))(\text{mrodd})\equiv$
 1273a $\text{let}\text{rec}=\{\{\text{di}\mapsto\text{d_cm_ch}[\text{di},\text{cmi}]?\mid\text{di}:\text{DI}\cdot\text{di}\in\text{dis}\}\}\text{in}$
 1273b $\text{let}\text{mrodd}'=\text{mrodd}\dagger[\text{di}\mapsto\langle\text{rec}(\text{di})\rangle^{\wedge}\text{mrodd}(\text{di})\mid\text{di}:\text{DI}\cdot\text{di}\in\text{dis}]\text{in}$
 1273c $\text{cm_cp_ch}[\text{cmi},\text{cpi}]\text{!mrodd}';$
 1273d $\text{monitor}(\text{cmi},(\text{dis},\text{cpi}))(\text{mrodd}')$
 1273 **end end**
 1273 **axiom** $\text{cmi}=\text{cm}_i\wedge\text{cpi}=\text{cp}_i$

We have decided to let the monitor maintain the present and past time-stamped drone positions. It is the monitor which records these positions. Not the planner. But the monitor provides these traces, again-and-again, to the planner.

Planner Behaviour

1274. The signature of the planner behaviour
- lists the planner's unique identifier, carries the planner's mereology, has, perhaps ..., some static arguments, has the programmable planner directories, and
 - further designates the single **input** channel `cm_cp_ch` and the single **output** channel `cp_ca_ch`.
1275. The planner [otherwise] behaves as follows:
- the planner [internal] non-deterministically ("coin-flipping") decides whether to transfer a drone between business swarms, or to calculate flight plans, or ... other.
 - Depending on the [outcome of the "coin-flipping"] the planner
 - either effects a transfer,
 - by delegating to an auxiliary function, `transfer`, the necessary modifications of the swarm directory –
 - whereupon the planner behaviour resumes;
 - or effects a [re-]calculation on drone flights,
 - by, again, delegating to an auxiliary function, `flight_planning`, the necessary calculations –
 - which are communicated to the *actuator*,
 - whereupon the planner behaviour resumes;
 - or ... other !

value

```

1275 planner: cpi:CPI × (cai>CAI×cmi:CMI×gi:GI) × TDIR →
1275   in cm_cp_ch[cmi,cpi], g_cp_ch[gi,cpi] out cp_ca_ch[cpi,cai] Unit
1274 planner(cpi,(cai,cmi,gi),...)(bdir,sdir,ddir) ≡
1275a   let cmd = "transfer" [] "flight_plan" [] ... in
1275b   cases cmd of
1275c     "transfer" →
1275(c)i       let sdir' = transfer(tdir) in
1275(c)ii      planner(cpi,(cai,cmi,gi),...)(bdir,sdir',ddir) end
1275d     "flight_plan" →
1275(d)i       let ddir' = flight_planning(tdir) in
1275(d)ii      planner(cpi,(cai,cmi,gi),...)(bdir,sdir,ddir') end
1275e     ...
1274   end
1274 axiom cpi=cp_i∧cai=ca_i∧cmi=cm_i∧gi=gi

```

The Auxiliary transfer Function

1276. The *transfer* function has a simpler signature than the planner behaviour in that it need not communicate with other behaviours.
- The transfer function *internal non-deterministically chooses* a business designator, `bi`;
 - from among that business' swarm designators it *internal non-deterministically chooses* two distinct swarm designators, `fsi`, `tsi`;
 - and from the `fsi` entry in `sdir` (which is set of enterprise drone identifiers), it *internal non-deterministically chooses* an enterprise drone identifier, `di`.

- d. Given the swarm and drone identifiers *the resulting swarm directory* can now be made to reflect the transfer: reference to di is *removed* from the fsi entry in sdir and that reference instead *inserted* into the tsi entry.

value

```

1276 transfer: TDIR → SDIR
1276 transfer(bdir,sdir,ddir) ≡
1276a   let bi:BI•bi ∈ dom bdir in
1276b   let fsi,tsi:SI•{fsi,tsi} ⊆ bdir(bi) ∧ fsi ≠ tsi in
1276c   let di:DI•di ∈ sdir(fsi) in
1276d   sdir † [fsi→sdir(fsi)\{di}] † [tsi→sdir(tsi)∪{di}]
1276   end end end

```

The Auxiliary flight_planning Function

1277. The signature of the flight_planning behaviour needs two elements: the triplet of business, swarm and drone directories, and the planner-to-actuator channel.
- The flight_planning behaviour offers to accept the time-stamped recordings of the most recent drone positions and dynamics as well as all the past such recordings.
 - The flight_planning behaviour selects, *internal, non-deterministically* a business, designated by bi,
 - one of whose swarms, designated by si, it has thus decided to perform a flight [re-]calculation for.
 - An objective for the new flight plan is chosen.
 - The flight_plan is calculated.
 - That flight plan is communicated to the *actuator*.
 - And the flight plan, appended to the drone directory's (past) flight plans.

value

```

1277 flight_planning: TDIR → in cm_cp_ch[cmi,cpi], out cp_ca_ch[cpi,cai] DTP
1277 flight_planning(bdir,sdir,ddir) ≡
1277a   let dtp = cm_cp_ch[cpi,cai] ? ,
1277b   bi:BI • bi ∈ dom bdir
1277c   let si:SI • si ∈ bdir(bi) in
1277d   let fp_obj:fp_objective(bi,si) in
1277e   let flight_plan = calculate_flight_plan(dtp,sdir(si),fp_obj,tdir) in
1277f   cp_ca_ch[cpi,cai] ! flight_plan ;
1277g   ⟨flight_pla⟩ddir
1277   end end end end

```

type

1277d FP_OBJ

value

```

1277d fp_objective: BI × SI → FP_OBJ
1277d fp_objective(bi,si) ≡ ...

```

1278. The calculate_flight_plan function is the absolute focal point of the *planner*.

```

1278 calculate_flight_plan: DTP × DI-set × FP-Obj × TDIR → FP
1278 calculate_flight_plan(dtp,sdir(si),fp_obj,tdir) ≡ ...

```

There are many ways of calculating flight plans.

[181, Mehmood et al., Stony Brook, 2018: *Declarative vs Rule-based Control for Flocking Dynamics*] is one such:

TO BE WRITTEN

In [220, 221, 222, Craig Reynolds: OpenSteer, *Steering Behaviours for Autonomous Characters*]

TO BE WRITTEN

In [194, Reza Olfati-Saber: Flocking for Multi-agent Dynamic Systems: Algorithms and Theory, 2006]

TO BE WRITTEN

The `calculate_flight_plan` function, Item 1278 [Page 432], is deliberately provided with all such information that can be gathered and hence can be the only ‘external’²² data that can be provided to such calculation functions,²³ and is therefore left further unspecified; future work²⁴ will show whether this assumption holds. If it does, then, OK, and we can proceed. If it does not, we shall revise the present model.

Actuator Behaviour

1279. The actuator accepts a current flight plan, `cfp:CFP`, i.e., a number of enterprise drone identifier-indexed flight plans, from the planner.
1280. The signature of the actuator behaviour lists the actuator’s unique identifier, carries the actuator’s mereology, has, perhaps ..., some static arguments, has the programmable flight directory, and further designates the **input** channel `cp_ca_ch[cp_i,cai]` and the **output** channel `ca_ed_ch[cai,*]`.
1281. The actuator further behaves as follows:
- a. It offers to accept a current flight plan from the planner.
 - b. It then proceeds to offer those enterprise drones which are designated in the flight plan their flight plan.
 - c. Whereupon the actuator resumes being the actuator, now with its programmable flight plan directory updated with the latest such !

type

1279 `CFP = EDI \rightarrow FP`

value

1280 `actuator: cai:CAI \times (cp_i:CPI \times edis:EDI-set) \rightarrow FDDIR \rightarrow`

1280 `in cp_ca_ch[cp_i,cai] out {ca_ed_ch[cai,edi]|edi:EDI•edi \in edis} Unit`

1281 `actuator(cai,(cp_i,edis),...)(pfp,pfpl) \equiv`

1281a `let cfp = ca_cp_ch[cai,cp_i] ? in comment: fp:EDI \rightarrow FP`

1281b `|| {ca_ed_ch[cai,edi]!cfp(edi)|edi:EDI•edi \in dom cfp} ;`

1281c `actuator(cai,(cp_i,edis),...)(cfp,⟨pfp⟩ \wedge pfpl)`

1279 `end`

1280 `axiom cai=ca_i \wedge cp_i=cp_i`

‘Other’ Drone Behaviour

1282. The signature of the ‘other’ drone behaviour
- a. lists the ‘other’ drone’s unique identifier, the ‘other’ drone’s mereology, has, perhaps ..., some static arguments; then the programmable attribute of the geography (i.e., the area, the land and the weather) it is moving over and in;

²² Flight plan *objectives* are here referred to as ‘internal’.

²³ Well – better check this!

²⁴ – for you ShaoFa !

- b. then, as **input** channels, the *inert*, *active*, *autonomous* and *biddable* attributes: velocity, acceleration, orientation and position, and, finally
 - c. further designates the array **input** channel `g_d_ch[*]` from the *geography* and the array **output** channel `d_cm_ch[*]` to the *monitor*.
1283. The ‘other’ drone otherwise behaves as follows:
1284. internal, non-deterministically the ‘other’ drone chooses to either ..., or "pro"viding to the monitors request for drone "dyn"amics, or
1285. If the choice is ... ,
1286. If the choice is "provide dynamics" the behaviour `drone_monitor` is invoked, with arguments similar to that of `other_drone`, but “marked” with an additional, “frontal” argument: "other", and with “tail”, programmable arguments ($\langle \rangle, \langle \rangle$).
1287. If the choice is

value

```

1282 other_drone: odi:ODI × (cmi:CMI×gi:GI) × ... → (DYN×ImG) →
1282b   in attr_DYN_ch[odi],g_d_ch[gi,odi] out d_cm_ch[odi,cmi] Unit
1283 other_drone(odi,(cmi,gi),...)(dyn:(v,a,o,p),img) ≡
1284   let mode = "... " ∪ "pro_dyn" ∪ "... " in
1284   case mode of
1285     "... " → ... ,
1286     "pro_dyn" → drone_moni(odi,(cmi,gi),...)(dyn:(v,a,o,p),img)
1287     "... " → ...
1284   end
1282   end

```

1288. If the choice is "provide dynamics"
- a. then the drone-monitor behaviour ascertains its dynamics (velocity, acceleration, orientation and position),
 - b. informs the monitor ‘thereof’, and
 - c. resumes being the ‘other’ drone with that updated, programmable dynamics.

value

```

1288 drone_moni: odi:ODI × (cmi:CMI×gi:GI) × ... → (DYN×ImG) →
1288   in attr_DYN_ch[odi],g_d_ch[gi,odi] out d_cm_ch[odi,cmi] Unit
1287 drone_moni(odi,(cmi,gi),...)(dyn:(v,a,o,p),img) ≡
1288a   let (ti,dyn',img') =
1288a     (time(),
1288a       (let (v',a',o',p') = attr_DYN[odi]? in
1288a         (v',a',o',p'),
1288a         d_g_ch[odi,gi]!p' ; g_d_ch[gi,odi]? end)) in
1288b   d_cm_ch[odi,cmi] ! (ti,dyn') ;
1288c   other_drone(cai,(cpi,edis),...)(dyn',img')
1288a   end

```

Enterprise Drone Behaviour

1289. The enterprise donor lists its enterprise drone’s unique identifier, carries it’s mereology, has, perhaps ..., some static arguments, the programmable enterprise drone attributes: a pair of the present flight plan, and the past flight plans, and a pair of the most recently observed dynamics and immediate geography, and further designates the single **input** channel and the **output** channel array .

Enterprise drones otherwise behave as follows:

1290. internal, non-deterministically an enterprise drone chooses to either "rec"ording the "geo"graphy, i.e., the area, land and weather it is situated in, or "pro"viding to the monitors request for drone "dyn"amics, or "acc"epting the actuators offer of a new "f"light "p"lan, or "move" "on" (i.e., continue to fly), either "follow"ing the "flight plan" most recently received from the actuator, or, "ignor"ing this directive, "just plondering on"!
1291. If the choice is "rec_geo" then the enterprise_geo behaviour is invoked,
1292. If the choice is "pro_dyn" (provide dynamics to the *monitor*) then the enterprise_moni behaviour is invoked,
1293. If the choice is "acc_fp" then the enterprise_accept_flight_plan behaviour is invoked,
1294. If the choice is "move_on" then the enterprise drone decides either to "ignore" the flight plan, or to "follow" it.
- If it "ignore"s the flight plan then the enterprise_ignore behaviour is invoked,
 - If the choice is "follow" then the enterprise_follow behaviour is invoked.

```

1289   enterprise_drone: edi:EDI × (cmi:CMI × cai:CAI × gi:GI) →
1289   ((FPL × PFPL) × (DDYN × ImG)) →
1289   in attr_DYN_ch[edi], g_d_ch[gi,edi], ca_ed_ch[cai,edi]
1289   out d_cm_ch[edi,cmi], d_g_ch[edi,gi]   Unit
1289   enterprise_drone(edi, (cmi, cai, gi), ...) (fpl, pfpl, (ddyn, img)) ≡
1290   let mode = "rec_geo" [] "pro_dyn" [] "acc_fp" [] "move_on" in
1290   case mode of
1291     "rec_geo" → enterprise_geo(edi, (cmi, cai, gi), ...) (fpl, pfpl, (ddyn, img))
1292     "pro_dyn" → enterprise_moni(edi, (cmi, cai, gi), ...) (fpl, pfpl, (ddyn, img))
1293     "acc_fp" → enterprise_acc_fl_pl(edi, (cmi, cai, gi), ...) (fpl, pfpl, (ddyn, img))
1294     "move_on" →
1294       let m_o_mode = "ignore" [] "follow" in
1294       case m_o_mode of
1294a         "ignore" → enterprise_ignore(edi, (cmi, cai, gi), ...) (fpl, pfpl, (ddyn, img))
1294b         "follow" → enterprise_follow(edi, (cmi, cai, gi), ...) (fpl, pfpl, (ddyn, img))
1300       end
1300     end
1290   end
1290   end
1289   axiom cmi=cmi ∧ cai=cai ∧ gi=gi

```

1295. If the choice is "rec_geo"
- then dynamics is ascertained so as to obtain a positions;
 - that position is used in order to obtain a "fresh" immediate geography;
 - with which to resume the enterprise drone behaviour.

```

1289   enterprise_geography: edi:EDI × (cmi:CMI × cai:CAI × gi:GI) →
1289   ((FPL × PFPL) × (DDYN × ImG)) →
1289   in attr_DYN_ch[edi], g_d_ch[gi,edi], ca_ed_ch[cai,edi]
1289   out d_cm_ch[edi,cmi], d_g_ch[edi,gi]   Unit
1289   enterprise_geography(edi, (cmi, cai, gi), ...) ((fpl, pfpl), (ddyn, img)) ≡
1295a   let (v, a, o, p) = attr_DYN_ch[edi]? in
1295b   let img' = d_g_ch[edi,gi]!p; g_d_ch[gi,edi]? in
1295c   enterprise_drone(edi, (cmi, cai, gi), ...) ((fpl, pfpl), ((v, a, o, p), img'))
1295a   end end

```

1296. If the choice is "pro_dyn" (provide dynamics to the *monitor*)
- then a triplet is obtained as follows:
 - the current time,
 - the dynamics (v,a,o,p) , and
 - the immediate geography of position p ,
 - such that the *monitor* can be given the current dynamics,
 - and the enterprise drone behaviour is resumed with updated dynamics and immediate geography.

```

1289 enterprise_monitor: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1289 ((FPL×PFPL)×(DDYN×ImG)) →
1289 in attr_DYN_ch[edi],g_d_ch[gi,edi],
1289 out d_cm_ch[edi,cmi],d_g_ch[edi,gi] Unit
1289 enterprise_monitor(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1296a let (ti,ddyn',img') =
1296b (time(),
1296c (let (v,a,o,p) = attr_DYN[edi]? in
1296c (v,a,o,p),
1296d d_g_ch[edi,gi]!p;g_d_ch[gi,edi]? end)) in
1296e d_cm_ch[edi,cmi] ! (ti,ddyn') ;
1296f enterprise_drone(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn',img'))
1296a end

```

1297. If the choice is "acc_fp"
- the enterprise drone offers to accept a new flight plan from the *actuator*
 - and the enterprise drone behaviour is resumed with that flight plan now becoming the next current flight plan and whatever is left of the hitherto current flight plan appended to the past flight plan list.

```

1289 enterprise_acc_fl_pl: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1289 ((FPL×PFPL)×(DDYN×ImG)) → in ca_ed_ch[cai,edi] Unit
1289 enterprise_acc_fl_pl(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1297a let fpl' = ca_ed_ch[cmi,edi] ? in
1297b enterprise_drone(edi,(cmi,cai,gi),...)(fp',⟨fpl⟩^pfpl,(ddyn,img))
1297a end

```

1298. If the choice is "move_on" and the enterprise drone decides to "ignore" the flight plan,
- then it ascertains where it might be moving with the current dynamics
 - and then it just keeps moving on till it reaches that dynamics
 - from about where it resumes the enterprise drone behaviour.

```

1289 enterprise_ignore: edi:EDI×(cmi:CMI×cai:CAI×gi:GI) →
1289 ((FPL×PFPL)×(DDYN×ImG)) →
1289 in attr_DYN_ch[edi] out d_cm_ch[edi,cmi],d_g_ch[edi,gi] Unit
1289 enterprise_ignore(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) ≡
1298a let (v',a',o',p') = increment(dyn,img) in
1298b while let (v'',a'',o'',p'') = attr_DYN_ch[odi]? in
1298b ~close(p',p'') end do manoeuvre(dyn,img) ; wait δ t end ;
1298c enterprise_drone(cai,(cpi,edis),...)(fpl,pfpl,(attr_DYN_ch[odi]? ,img))
1298a end

```

1299. The manoeuvre behaviour is further unspecified. For a fixed wing aircraft it controls the *yaw*, the *roll* and the *pitch* of the aircraft, hence its flight path, by operating the *elevator*, *aileron*, *rudder* and the *thrust* of the aircraft based on its current dynamics, weight (including aircraft fuel), meteorological conditions (winds etc.).

value

1299 manoeuvre: $DYN \times ImG \rightarrow Unit$

1299 manoeuvre(dyn,img) $\equiv \dots$

The **wait** δt is some drone constant.

1300. If the choice is "move_on" and the enterprise drone decides to "follow" the flight plan,

- then, if the current flight plan has been exhausted, i.e., "used-up" it aborts (**chaos**²⁵)
- otherwise it ascertains where it might be moving, i.e., a next dynamics from with the current dynamics.
- So it then "moves along" until it has reached that dynamics –
- from about where it resumes the enterprise drone behaviour.

value

1289 enterprise_follow: $edi:EDI \times (cmi:CMI \times cai:CAI \times gi:GI) \rightarrow$

1289 $((FPL \times PFPL) \times (DDYN \times ImG)) \rightarrow$

1289 **in** attr_DYN_ch[edi] **out** d_cm_ch[edi,cmi],d_g_ch[edi,gi] **Unit**

1289 enterprise_follow(edi,(cmi,cai,gi),...)((fpl,pfpl),(ddyn,img)) \equiv

1300a **if** fpl = $\langle \rangle$ **then chaos else**

1300b **let** (v',a',o',p') = increment(dyn,img,hd fpl) **in**

1300c **while let** (v'',a'',o'',p'') = attr_DYN_ch[odi]? **in**

1300c \sim close(p',p'') **end do** manoeuvre(hd fpl,dyn,img) ; **wait** δt **end** ;

1300d enterprise_drone(edi,(cmi,cai,gi),...)((**tl**fpl,pfpl),(attr_DYN_ch[odi]?,img))

1300a **end end**

1301. The (overloaded) manoeuvre behaviour is further unspecified. For a fixed wing aircraft it controls the *yaw*, the *roll* and the *pitch* of the aircraft, hence its flight path, by operating the *elevator*, *aileron*, *rudder* and the *thrust* of the aircraft based on its current dynamics, weight (including aircraft fuel), meteorological conditions (winds etc.).

value

1301 manoeuvre: $FPE \times DYN \times ImG \rightarrow Unit$

1301 manoeuvre(fpe,dyn,img) $\equiv \dots$

The **wait** δt is some drone constant.

Geography Behaviour

1302. The *geography* behaviour definition

- lists the geography behaviour's unique identifier, carries the its mereology, has the static argument of its Euclidean point space, and
- further designates the single **input** channels cp_g_ch[cp,gi] from the *planner* and d_g_ch[*,gi] from the drones and the **output** channels g_cp_ch[gi,cp] to the *planner* and g_d_ch[gi,*] to the *drones*.

1303. The *geography* otherwise behaves as follows:

- Internal, non-deterministically the geography chooses to either "resp"nd to a request from the "plan"ner.

²⁵ **chaos** means that we simply decide not to describe what then happens !

- b. If the choice is
 - c. "resp_plan"
 - i. then the *geography* offers to accept a request from the *planner* for the *immediate geography* of an *area* "around" a *point* and
 - ii. then the *geography* offers that information to the *planner*,
 - iii. whereupon the *geography* resumes being that;
 - else if the choice is
 - d. "resp_dron"
 - i. then then the *geography* offers to accept a request from the *planner* for the *immediate geography* of an *area* "around" a *point* and
 - ii. then the *geography* offers that information to the *planner*,
 - iii. whereupon the *geography* resumes being that.
1304. The *area* function takes a pair of a point and a pair of *land* and *weather* and yields an *immediate geography*.

value

```

1302 geography: gi:GI × gm:(cpi:CPI×cmi:CMI×dis:DI-set) × EPS →
1302a   in cp_g_ch[cpi,gi], d_g_ch[*,gi]
1302b   out g_cp_ch[gi,cpi], g_d_ch[gi,*] Unit
1302   geography(gi,(cpi,cmi,dis),eps) ≡
1303a   let mode = "resp_plan" [] "resp_dron" [] ... in
1303b   case mode of
1303c     "resp_plan" →
1303(c)i     let p = cp_g_ch[cpi,gi] ? in
1303(c)ii     g_cp_ch[gi,cpi] ! area(p,(attr_L_ch[gi]?,attr_W_ch[gi]?)) end
1303(c)iii    geography(gi,(cpi,cmi,dis),eps)
1303d     "resp_dron" →
1303(d)i     let (p,di) = []{(d_g_ch[di,gi]?,di)|di:DI·di ∈ dis} in
1303(d)ii     g_cp_ch[di,cpi] ! area(p,(attr_L_ch[gi]?,attr_W_ch[gi]?)) end
1303(d)iii    geography(gi,(cpi,cmi,dis),eps)
1302   end end

```

axiom

```
1302 gi=gi^cpi=cpi^smi=cmi^dis=dis
```

value

```
1304 area: P × (L × W) → ImG
```

```
1304 area(p,(l,w)) ≡ ...
```

F.5 Conclusion

TO BE WRITTEN

G

Container Terminals

Abstract

- We present a recording of stages and steps of a development of a domain analysis & description.
- It exemplifies a response to the question:
 - ⊗ *what may be a mathematical answer to the question*
 - ⊗ *what is a container terminal?*

G.1 Introduction

Domain descriptions precede requirements prescriptions, and these precede software designs.

There is an analogy: The study and knowledge about physics, that is, physics as a science, precedes the various classes of engineering: building engineering, mechanical engineering, chemical engineering, electrical & electronics engineering. Today You would not embark on any of these without a firm grasp of their underlying sciences.

So we, as computer/computing scientists and software engineers shall not undertake designs of software without first having a firm grasp of its requirements; and we shall not attempt to document requirements without first having a firm grasp of the domain of that software.

A variants of the domain for the software monitoring and control of container terminal ports is described in this experimental report.

G.2 Basic Concepts of Container Terminal Ports

G.2.1 Pictorial Renditions: Diagrams

Figure G.1 [Page 440] shows one “rendition” of a container terminal port. Figures G.2 [Page 441] and G.3 [Page 457] shows two other such “renditions:”.

G.2.2 Terminology - a Caveat

The terms introduced in this section are mine. They are most likely not the correct technical terms of the container shipping and stowage trade. I expect to revise this section.

Bay Stack: contains indexed set of *rows* (of stacks of containers) on land and in terminal ports.

Container: smallest unit of central (i.e., huge) concern !

Container Stowage Area: An area of a vessel or a terminal port where containers are stored, during voyage, respectively awaiting to be either brought out to shippers or onto vessels (from shippers).

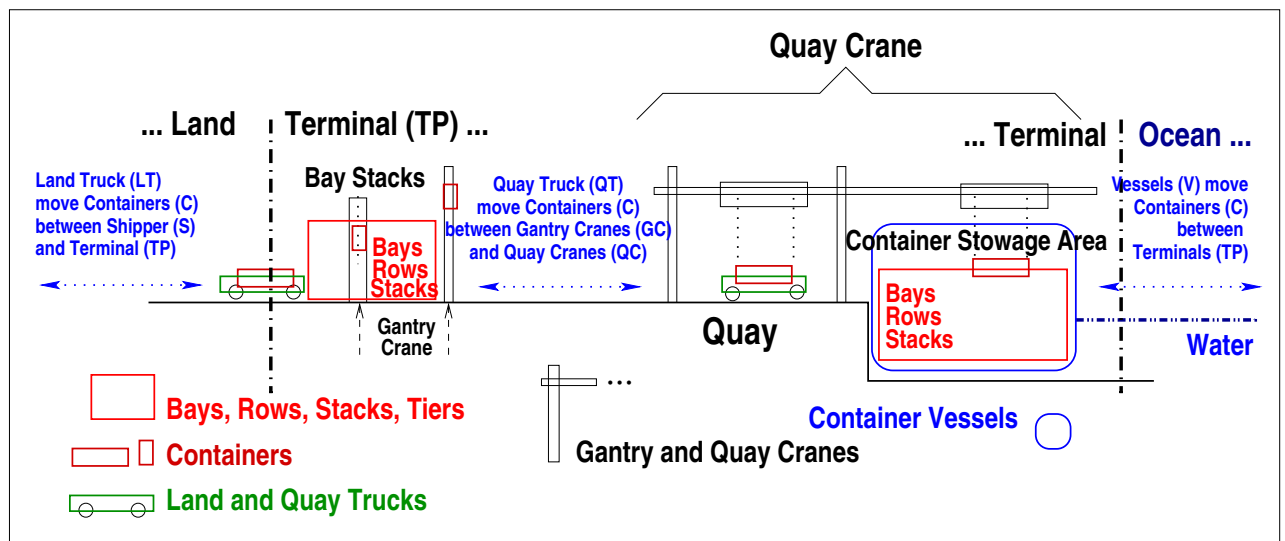


Fig. G.1. A “from the side” snapshot of terminal port activities

Crane:

Gantry¹ Crane: transfers *containers* between *land trucks* and *bay stacks*.

Quay Crane: transfer *containers* between *quay trucks* and *vessels*.

Land ... as you know it ...

Ocean ... as you know it ...

Shipper: arranges shipment of containers with container lines².

Quay: area of terminal next to vessels (hence water).

Row: contains indexed set of *stacks* (of containers).

Stack: contains indexed set of *containers*.

We shall also, perhaps confusingly, use the term *stack* referring to the land-based bays of a terminal.

Terminal: area of land and water between land and ocean equipped with container stowage area, stack and quay cranes, etc.

Truck:

Land Truck: usually separately operated truck which transports *containers* between *shippers* and *gantry cranes*.

Quay Truck: terminal operated special truck which transports *containers* between *gantry cranes* and *quay cranes*.

Tier: index of *container* in *stack*.

Vessel: contains a *container stowage area*.

G.2.3 Assumptions

Without loss of generality we can assume that there is exactly one gantry crane per land-based terminal bay stack; quay cranes each serve exactly one bay on a vessel; there are enough quay cranes to serve all bays of any berthed vessel; quay trucks may serve any (quay and gantry) crane; land trucks may serve more than one terminal; et cetera.

² The present model neither analyses nor describes the logistics of shipping. A shipping logistics model, however, can be developed [nicely] in the context of the present model, i.e., with clear interfaces to this model.

G.3 Endurants

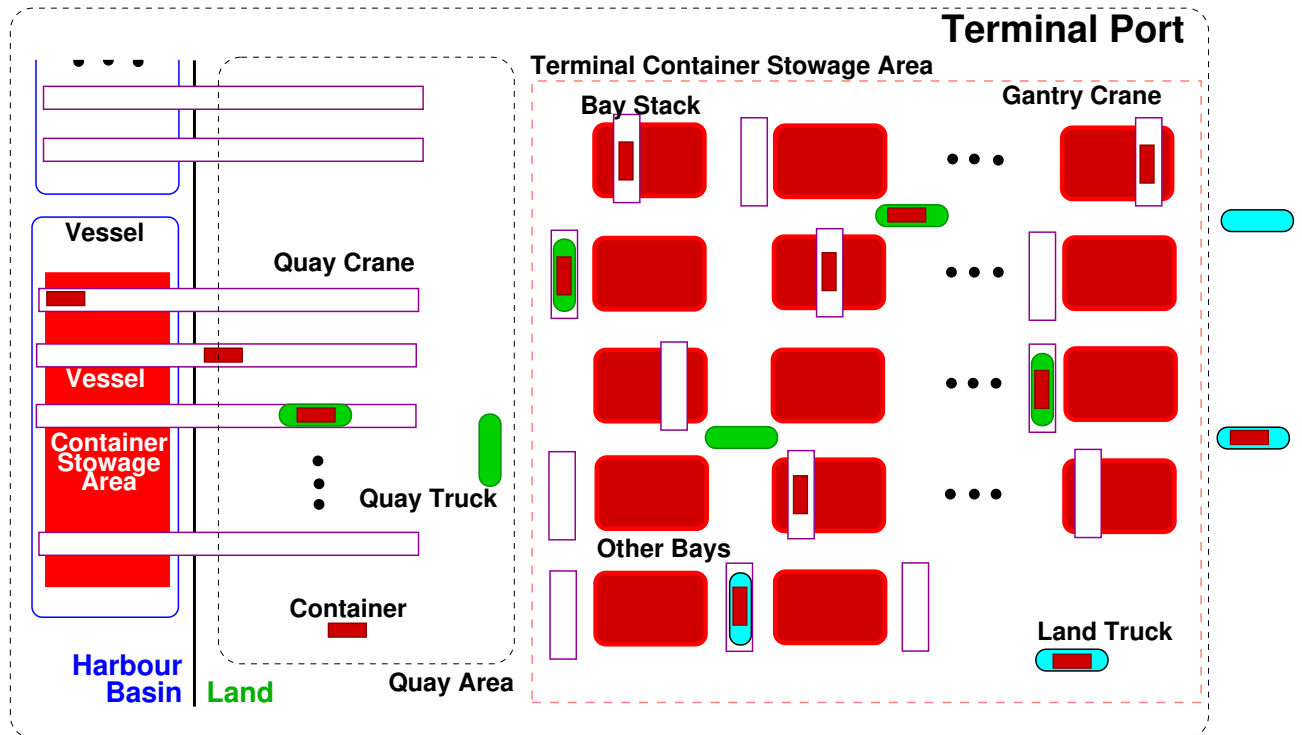


Fig. G.2. A “from above” snapshot of terminal port activities

G.3.1 The Container Line Industry

0. The domain to be analysed & described is the *container line industry*.

type

0 CLI

G.3.2 Parts

Our model has, perhaps arbitrarily, focused on just some of the manifest, i.e., observable parts of a domain of container terminal ports. We shall, invariably, refer to container terminal ports as either container terminals, or terminal ports, tp:TP, or just terminals.

In the container line industry, CLI, we can observe

1. a structure, STP, of all terminal ports, and from each such structure, an indexed set, TPs, of two or more container *terminal* ports, TP;
2. a structure, SV, of all container vessels, and from each such structure, an indexed set, Vs, of one or more container *vessels*, V; and
3. a structure, SLT, of all land trucks, and from each such structure, a non-empty, indexed set, LTs of land trucks, LT.

type

- 1 STP, $TPs = TP\text{-set}$, TP
- 2 SV, $Vs = V\text{-set}$, V
- 3 SLT, $LTs = LT\text{-set}$, LT

value

- 1 $obs_STPs: CLI \rightarrow STPs$, $obs_TPs: STPs \rightarrow TPs$
- 2 $obs_SV: CLI \rightarrow SV$, $obs_Vs: SVs \rightarrow Vs$
- 3 $obs_SLT: CLI \rightarrow SLT$, $obs_LTs: SLT \rightarrow LTs$

axiom

- 1 $\forall cli:CLI \cdot \mathbf{card} \text{ obs_TPs}(obs_STPs(cli)) \geq 2$
- 2 $\wedge \mathbf{card} \text{ obs_Vs}(obs_SV(cli)) \geq 1$
- 3 $\wedge \mathbf{card} \text{ obs_LTs}(obs_SLT(cli)) \geq 1$

G.3.3 Terminal Ports

In a terminal port, $tp:TP$, one can observe

4. a [composite] container stowage area, $csa:CSA$, cf. Item 38 [Page 445];
5. a structure, $sqc:SQC$, of quay cranes, and from that, a non-empty, indexed set, $qcs:QCs$, of one or more quay cranes, $qc:QC$;
6. structure, $sqt:SQT$, of quay trucks, and from that a non-empty, indexed set, $qts:QTs$, of quay trucks, $qt:QT$;
7. a structure, $sgc:SGC$, of gantry cranes, and from that a non-empty, indexed set, $gcs:GCs$, of one or more stack cranes, $gc:GC$; and
8. a[n atomic] terminal port command center, $cc:CC$.
9. As we shall see later, all of the above parts potentially embody containers.

type

- 4 CSA
- 5 SQC, $QCs = QC\text{-set}$, QC
- 6 SQT, $QTs = QT\text{-set}$, QT
- 7 SGC, $GCs = GC\text{-set}$, GC
- 8 CC
- 9 C

value

- 4 $obs_CSA: TP \rightarrow CSA$
- 5 $obs_SQC: TP \rightarrow SQC$, $obs_QCs: SQC \rightarrow QCs$
- 6 $obs_SQT: TP \rightarrow SQT$, $obs_QTs: SQT \rightarrow QTs$
- 7 $obs_SGC: TP \rightarrow SGC$, $obs_GCs: SGC \rightarrow GCs$
- 8 $obs_CC: TP \rightarrow CC$

axiom

- 5 $\forall sqc:SQC \cdot \mathbf{card} \text{ obs_QCs}(sqc) \geq 1$
- 6 $\forall sqt:SQT \cdot \mathbf{card} \text{ obs_QTs}(sqt) \geq 1$
- 7 $\forall sgc:SGC \cdot \mathbf{card} \text{ obs_GCs}(sgc) \geq 1$

G.3.4 Unique Identifications**Unique Identifiers**

10. Land trucks have unique identifiers.
11. Gantry cranes have unique identifiers.
12. Bays stacks of terminal container stowage areas have unique identifiers, cf. Item 18.
13. Quay trucks have unique identifiers.
14. Quay cranes have unique identifiers.
15. Vessels have unique identifiers.
16. Terminal port command centers have unique identifiers.
17. Containers have unique identifiers.
18. Bays of container stowage areas have unique identifiers.
19. Rows of a bay have unique identifiers.
20. Stacks of a row have unique identifiers.
21. The part unique identifier types are mutually disjoint.

type

- 10 LTI
- 11 GCI
- 12 BI
- 13 QTI
- 14 QCI
- 15 VI
- 16 CCI
- 17 CI
- 18 BI
- 19 RI
- 20 SI

axiom

- 21 LTI,GCI,QTI,QCI,VI,CCI,CI,BIU,RI,SI

- 21 mutually disjoint

value

- 10 uid_LT: LT → LTI
- 11 uid_GC: GC → GCI
- 12 uid_BS: BAY → BI
- 13 uid_QT: QT → QTI
- 13 uid_QC: QC → QCI
- 14 uid_V: V → VI
- 16 uid_CC: CC → CCI
- 17 uid_C: C → CI
- 13 uid_BAY: BAY → BI
- 14 uid_ROW: ROW → RI
- 15 uid_STK: STK → SI

Distinctness of Parts

22. If two containers are different then their unique identifiers must be different.

axiom

- $$22 \quad \forall c, c': C \cdot c \neq c' \Rightarrow \text{uid}_C(c) \neq \text{uid}_C(c')$$

The same distinctness criterion applies to stacks, rows, bays, container storage areas, terminal ports, cranes, vessels, etc.

G.3.5 Containers**General**

Containers, their transport, stowage during transport, and stowage before and after transport is “*what it is all about!*”

Containers are first **1** transported by trucks from shippers to gantry cranes of terminal ports; then **2** transferred by gantry cranes onto bay stacks of terminal ports, also referred to as the terminal stowage area; they may rest there for some time **3-4** before being transferred onto vessels; these transfers take place in a short sequence of transfers and moves: **5** by gantry cranes from bay stacks onto quay trucks; **6** by quay trucks from gantry cranes to quay cranes; **7** by quay cranes onto the vessel stowage area of vessels; by vessels from terminal ports to terminal ports. And vice versa: in the other direction **8-14**. Container transport may involve two or more vessel transports. Containers undergoing two or more vessel journeys are temporarily stowed in terminal port bay stacks while awaiting onward voyage.

Figure G.3 [Page 457] explains the, **1-14**, numbering above.

We shall refer to the stowage area on-board vessels as vessel stowage areas, and the container in the terminal stowage area as in “the bay stacks” !

Container Properties: Unique Identifiers and Attributes

9 We [have] postulate[d] a type of containers, $c:C$.

17 We have identified containers by their unique identifiers, $ci:CI$.

And we ascribe a number of attributes to containers.

23. There is a *bill-of-lading*³.

24. There is a *way bill*⁴

25. Containers are of either 20'' (feet) or 40'' (feet) *size*.

26. Containers may contain *hazardous* (i.e., explosive or inflammable) goods.

27. Containers may need *refrigeration*.

28. Containers have a *weight*.

29. Finally containers may or may not contain *goods*.⁵

type

9 C

17 CI

23 BoL

24 WB

25 SZ == mkTF("twenty") | mkFF("fourty")

26 HG == mkEX("explosive") | mkIF("inflammable")

27 RF == mkOR("ordinary") | mkRE("refrigerated")

28 W

29 GOODS

value

17 uid_C: $C \rightarrow CI$

23 attr_BoL: $C \rightarrow BoL$

24 attr_WB: $C \rightarrow WB$

25 attr_SZ: $C \rightarrow SZ$

26 attr_HG: $C \rightarrow HG$

27 attr_RF: $C \rightarrow RF$

28 attr_W: $C \rightarrow W$

29 attr_GOODS: $C \rightarrow \text{GOODS-set}$

Container History

Since we can speak about it, we shall also model it: namely the history of a container.

30. A container history is a sequence of time-stamped place markers⁶.

A place marker is either

31. a vessel bay, row, stack and stack position,

³ Bill of lading: a document issued by a carrier (or their agent) to acknowledge receipt of cargo for shipment.

⁴ Way bill: a document issued by a carrier giving details and instructions relating to the shipment of a consignment of goods. Typically it will show the names of the consignor and consignee, the point of origin of the consignment, its destination, route and carriers (incl. vessels, terminal ports of origin, destination, as well as possible intermediary terminal ports).

⁵ We shall treat contained goods as an attribute rather than as components.

⁶ **Editorial:** I need to check the definition of the below markers versus their use. It appears that both my definitions and uses are rather hap-hazard !

- 32. a terminal port quay crane identifier,
- 33. a terminal port quay truck identifier,
- 34. a terminal port gantry crane identifier,
- 35. a terminal bay, row, stack and stack position, or a
- 36. a land truck identifier.

Markers are basically simple or composite unique identifiers.

type

- 30 $C_Hi = (TIME \times Marker)^*$
- 31 $Marker = VI \times BRSIP$
- 32 | $CCI \times QCI$
- 33 | $CCI \times QTI$
- 34 | $CCI \times GCI$
- 35 | $CCI \times BRSIP$
- 36 | LTI

Tagged Containers

- 37. By a tagged container we shall understand a pair of a container history and a container

type

- 37 $TagC = C_Hi \times C$

Discussion:

We thus make a distinction between a container and a tagged container. There is the “bare” container, with its unique identity and its (many) attributes. And then there is that container tagged with its history. Throughout a transport, from a sender (shipper) to a receiver (shipper), the container, it is assumed in this model, remains unchanged: no change to any attribute. And then there is the “additional tag” which records the transport history of that container. It is augmented whenever the container is moved: from a land truck to a gantry crane [2] Item 211b [Page 470]], from a gantry crane to a terminal bay stack [3] Item 216a [Page 472]], onto a terminal bay stack top [4] Item 216a [Page 472]], off a terminal bay stack [5] Item 216a [Page 472]], from a terminal bay stack to a gantry crane [6] Item 219f [Page 473]], from a gantry crane to a quay truck [7] Item 220d [Page 473]], from a quay truck to a quay crane [8] Item 215d [Page 471]], from a quay crane to a vessel [9] Item 222d [Page 474]], and vice versa.

Modelling Container Stowage

We model terminal and vessel container stowage areas, $csa:CSA$, as maps over bays, rows and stacks – that is, maps from bay, row and stack identifiers to bays, rows and stacks – where stacks are possibly empty sequences of unique containers.

- 38. So there is an abstract notion of container stowage areas with a container stowage area being modelled a map from bay identifiers to bays.
- 39. A bay is a map from row identifiers to rows.
- 40. A row is a map from stack identifiers to stacks.
- 41. A stack is a sequence of zero, one or more unique, tagged containers.

type

```

38 CSA = BI  $\rightarrow$  BAY
38 BI
39 BAY = RI  $\rightarrow$  ROW, BS  $\equiv$  BAY
39 RI
40 ROW = SI  $\rightarrow$  STK
40 SI
41 STK = TagC*

```

Container Stowage Actions

Two actions can be performed on container stowage: *loading* a container onto the top of a stack of a row of a bay, and *unloading* a container onto the top of a stack of a row of a bay. Two versions of these actions can be defined: one that apply to vessels, and one that apply to bay stacks. The former applies to the entire set of bays of a[ny] vessel. The latter to a single bay of a[ny] terminal port. We formalise the *load* and *unload* operations in Sect. G.4.4 [Page 460].

Unique Identifiers: Two Useful Abbreviations

Container positions within a container stowage area can be represented:

42. by a triple of a bay identifier, a row identifier and a stack identifier, and
43. by the same triple extended with a stack position.

```

42 BRSI = BI  $\times$  RI  $\times$  SI
43 BRSIP = BI  $\times$  RI  $\times$  SI  $\times$  Nat

```

Unique Identifiers: Some Useful Index Set Selection Functions

44. From a container stowage area once can observe all bay identifiers.
45. From a bay once can observe all row identifiers.
46. From a row once can observe all stack identifiers.

value

```

44 xtr_BIs: CSA  $\rightarrow$  BI-set
44 xtr_BIs(csa)  $\equiv$  {uid_BAY(bay)|bay:BAY•bay  $\in$  xtr_BAYs(csa)}

44 xtr_RIs: BAY  $\rightarrow$  RI-set
45 xtr_RIs(bay)  $\equiv$  {uid_ROW(bay)|row:ROW•row  $\in$  obs_ROWs(bay)}

44 xtr_SIs: ROW  $\rightarrow$  SI-set
46 xtr_SIs(row)  $\equiv$  {uid_STK(row)|stk:STK•stk  $\in$  obs_STKs(row)}

```

Vessel Stowage

47. We consider the container stowage of a vessel to be an attribute of vessels.

```

47 attr_CSA: V  $\rightarrow$  CSA

```

We refer to Sect. G.3.11 [Page 456].

Terminal Stowage

48. We consider the container stowage of a terminal port to be a composite part of a terminal.

48 obs_CSA: TP \rightarrow CSA

We refer to Item 4 Sect. G.3.3 [Page 442].

Container Stowage Descriptors

Stowage can be described in terms of bay, row and stack descriptors. Descriptors define bay, row and stack identifiers and their orderings, and the maximum height of stacks.

49. A bays descriptor, bd:BD, is a pair
- of a list of the bay identifiers of a stowage, and,
 - for each bay identifier a rows descriptor.
50. A rows descriptor, rd:RD, is a pair
- of a list of the row identifiers of a bay, and,
 - for each row identifier a stacks descriptor.
51. A stacks descriptor, sd:SD, is a pair
- of a list of the stack identifiers of a row, and,
 - for each stack identifier a maximum height (natural number).

type

49 $BD = BI^* \times (BI \xrightarrow{m} RD)$

50 $RD = RI^* \times (RI \xrightarrow{m} SD)$

51 $SD = SI^* \times (SI \xrightarrow{m} \text{Nat})$

axiom

49 $\forall (bil, bim):BD \cdot \mathbf{elems} \ bil = \mathbf{dom} \ bim \wedge \mathbf{card} \ bil = \mathbf{card} \ \mathbf{elems} \ bil$

50 $\forall (ril, rim):RD \cdot \mathbf{elems} \ ril = \mathbf{dom} \ rim \wedge \mathbf{card} \ ril = \mathbf{card} \ \mathbf{elems} \ ril$

51 $\forall (sil, sim):SD \cdot \mathbf{elems} \ sil = \mathbf{dom} \ sim \wedge \mathbf{card} \ sil = \mathbf{card} \ \mathbf{elems} \ sil$

We omit treatment of special stowage for hazardous and refrigerated containers, and we shall, for brevity, assume all containers to be forty feet containers.

G.3.6 Trucks, Cranes, Bay Stacks and Vessels

We consider land and quay trucks, gantry and quay cranes, terminal bay stacks and vessels to all be atomic. All of these embody containers: land trucks, bay stacks and vessels over indefinite time intervals, cranes and quay trucks over short intervals.

G.3.7 Stacks

An aside: We shall use the term ‘stack’ in two senses: (i) as a component of container storage area bays; and (ii) to refer to the collection of stacks in a bay of a terminal container storage area.

52. Stacks are created empty, and hence stacks can be *empty*.
53. One can *push* a container onto a stack and obtain a *non-empty stack*.
54. One can *pop* a container from a *non-empty stack* and obtain a pair of a container and a possibly empty stack.

value52 empty: () → STK, is_empty: STK → **Bool**

53 push: C × STK → STK

54 pop: STK $\xrightarrow{\sim}$ (C × STK)**axiom**52 is_empty(empty()), \sim is_empty(push(c,stk))

53 pop(push(c,stk)) = (c,stk)

54 **pre** pop(stk),pop(push(c,stk)): \sim is_empty(stk)54 pop(empty()) = **chaos****G.3.8 Terminal Port Command Centers****Discussion**

We consider terminal port monitoring & control command centers to be atomic parts. The purpose of a terminal port command center is to monitor and control the servicing of

- arriving land trucks [A] Item 252 [Page 480]⁷;
- land truck delivery to gantry cranes [1] Item 211b [Page 470];
- gantry crane delivery to bay stacks [2] Item 216a [Page 472];
- bay stack delivery from gantry cranes [3] Item 216a [Page 472];
- bay stack delivery to gantry cranes [5] Item 219f [Page 473];
- gantry crane delivery to quay trucks [6] Item 220d [Page 473];
- quay truck delivery to quay cranes [7] Item 215d [Page 471];
- quay crane delivery to vessels [8] Item 222d [Page 474];
- vessel delivery to quay cranes [9] Item 227a [Page 475];
- quay cranes delivery to quay trucks [10] Item 227a [Page 475];
- quay truck delivery to gantry cranes [11] Item 223a [Page 474];
- gantry crane delivery to bay stacks [12] Item 216a [Page 472];
- bay stack delivery to gantry crane [4] Item 220d [Page 473];
- gantry crane delivery to land truck [13] Item 217d [Page 472]; and
- land truck “departure” [14] Item 212b [Page 470].

Also: the allocation and servicing (berthing) of any visiting vessel to quay positions

- [B] Item 255 [Page 480],
- [C] Item 235a [Page 478] and
- [D] Item 236a [Page 478];

respectively the servicing by quay cranes

- [C] Item 229a [Page 476] and
- [D] Item 230a [Page 476]

This implies that there are means for communication between a terminal command center and vessels, quay cranes, stack cranes, quay trucks, land trucks and terminal stacks.

⁷ We refer to Fig. G.3 [Page 457] for the significance of the boxed numerals and this itemised sequence.

Justification

We shall justify the concept of terminal monitoring & control, i.e., command centers. First, using the *domain analysis & description* approach of [81], we know that we are going, through a transcendental deduction, to model certain parts as behaviours. These parts, we decide, after some analysis (which we forego), to be vessels, quay cranes, quay trucks, stack cranes stacks and land trucks. Behaviours are usually like actors: they can instigate actions. But we decide, in our analysis, that some of these behaviours, quay cranes, quay trucks, stack cranes and stacks, are “*passive*” actors: are behaviourally not endowed with being able to initiate “own” actions. Instead, therefore, of all these behaviours, being able to communicate directly, pairwise, as indicated by Fig. G.3 [Page 457], we model them to *also* communicate *via* their terminal command centers.

This is how we justify the introduction of the concept of terminal command centers. They are an abstraction. In “*ye olde days*” you could observe, not one, but, perhaps, a hierarchy of terminal port offices, staffed by people, [each office, each group of staff] with its set of duties: communicating (by shouting, by hand-signals, by radio-phone) with approaching [and departing] vessels; scheduling quay positions, quay cranes and quay trucks; managing the operation of cranes and trucks; and, on a large scale, calculating stowage: on vessels and in terminals. Today, “*an age of ubiquitous computing*”, most of these offices and their staff are replaced by electronics: sensors, actuators, communication and computing, and with massive stowage data processing: where should containers be stowed on board vessels and in terminals so as to near-optimize all operations.

G.3.9 States, Global Values and Constraints

States

55. We postulate a container line industry *cli*:CLI.

From that we observe, successively, all parts:

56. the set, *tps*:TPs, of all terminal ports;

57. the set, *vs*:Vs, of all vessels;

- a. *vsas*:CSA-set, the set of all vessel stowage areas;
- b. *vbs*:BAY-set, the set of all bays of all vessels;
- c. the set, *vrs*:ROW-set, the set of all rows of all vessels;
- d. the set, *vss*:STK-set, the set of all stacks of all vessels;

58. the set, *lts*:LTs, of all land trucks;

59. the set, *gcs*:GCs, of all gantry cranes of all terminal ports;

- a. *tcsas*:CSA-set, the set of all terminal port container stowage areas of all terminal ports;
- b. *tbs*:BAY-set, the terminal port bays of all terminals;
- c. the set, *trss*:ROW-set, of all terminal bay stack rows of all terminal ports;
- d. the set, *tsss*:STK-set, of all terminal bay stack stacks of all terminal ports;

60. the set, *qts*:QTs, of all quay trucks of all terminal ports;

61. the set, *qcs*:QCs, of all quay cranes of all terminal ports;

62. the set, *ccs*:CCs, of all command centers of all terminal ports.

value

55 *cli*:CLI

56 *tps*:TP-set = $\text{obs_TPs}(\text{obs_TPS}(\text{cli}))$

57 *vs*:V-set = $\text{obs_Vs}(\text{obs_VS}(\text{cli}))$

57a *vsas*:CSA-set = $\{\text{obs_CSA}(v) \mid v:V \cdot v \in vs\}$

57b *vbs*:BAY-set = $\cup\{\text{rng}(\text{csa}) \mid \text{csa}:CSA \cdot \text{csa} \in vsas\}$

57c *vrs*:ROW-set = $\cup\{\text{rng}(\text{bs}) \mid \text{bs}:BS \cdot \text{bs} \in vbs\}$

```

57d vss:STK-set =  $\cup\{\mathbf{rng}(rs)|rs:RS\cdot rs \in vrs\}$ 
58 lts:LTs =  $\text{obs\_LTs}(\text{obs\_SLT}(cli))$ 
59 gcs:QC-set =  $\cup\{\text{obs\_GCs}(\text{obs\_SGC}(tp))|tp:TP\cdot tp \in tps\}$ 
59a tcas:CSA-set =  $\{\text{obs\_CSA}(tp)|tp:TP\cdot tp \in tps\}$ 
59b tbs:BAY-set =  $\cup\{\mathbf{rng}(csa)|csa:CSA\cdot csa \in csas\}$ 
59c trs:ROW-set =  $\cup\{\mathbf{rng}(bs)|bs:BS\cdot bs \in bss\}$ 
59d tss:STK-set =  $\cup\{\mathbf{rng}(rs)|rs:RS\cdot rs \in rss\}$ 
60 qts:QT-set =  $\cup\{\text{obs\_QTs}(\text{obs\_SQT}(tp))|tp:TP\cdot tp \in tps\}$ 
61 qcs:QC-set =  $\cup\{\text{obs\_QCs}(\text{obs\_SQC}(tp))|tp:TP\cdot tp \in tps\}$ 
62 ccs:CC-set =  $\{\text{obs\_CC}(tp)|tp:TP\cdot tp \in tps\}$ 

```

A Unique Identifier Functional

We can define a set of functions that extract from sets of parts their unique identifiers.

63. Let X be some part type, and
64. uid_X be any of the unique identifier observer functions uid_{TP} , uid_{CC} , uid_V , uid_{QC} , uid_{QT} , uid_{GC} , uid_{BAY} , uid_{LT} .
65. The xtr_UID is the functional which, given some unique identifier observer function, say X , and a set, $xs:X\text{-set}$, extracts the unique XI identifiers of the members x , of xs .

type

```
63 X
```

value

```

64  $\text{uid}_X: X \rightarrow XI$ 
65  $\text{xtr\_UID}: (X \rightarrow XI) \times X\text{-set} \rightarrow XI\text{-set}$ 
66  $\text{xtr\_UID}(f,xs) \equiv \{ f(x) \mid x:X \cdot x \in xs \}$ 

```

Unique Identifiers

Given the generic parts outlined in Sect. G.3.9 we can similarly define generic sets of unique identifiers.

66. the set, tp_uis , of all terminal port identifiers;
67. the set, cc_uis , of all terminal port command center identifiers;
68. the set, v_uis , of all vessel identifiers;
69. the set, qc_uis , of quay crane identifiers of all terminal ports;
70. the set, qt_uis , of quay truck identifiers of all terminal ports;
71. the set, sc_uis , of stack crane identifiers of all terminal ports;
72. the set, stk_uis , of stack identifiers of all terminal ports;
73. the set, lt_uis , of all land truck identifiers; and
74. the set, uis , of all vessel, quay crane, quay truck, gantry crane, terminal bay stack, and land truck identifiers.

value

```

66  $tp\_uis:TPI\text{-set} = \text{xtr\_UID}(\text{uid}_{TP},tps)$ 
67  $cc\_uis:TPI\text{-set} = \text{xtr\_UID}(\text{uid}_{CC},tccs) \equiv \{\text{uid}_{CC}(cc)|cc:CC\cdot cc \in tccs\}$ 
68  $v\_uis:VI\text{-set} = \text{xtr\_UID}(\text{uid}_V,vs) \equiv \{\text{uid}_V(v)|v:V\cdot v \in vs\}$ 
69  $qc\_uis:QCI\text{-set} = \text{xtr\_UID}(\text{uid}_{QC},qcs) \equiv \{\text{uid}_{QC}(qc)|qc:QC\cdot qc \in qcs\}$ 
70  $qt\_uis:QTI\text{-set} = \text{xtr\_UID}(\text{uid}_{QT},qts) \equiv \{\text{uid}_{QT}(qt)|qt:QT\cdot qt \in qts\}$ 
71  $gcs\_uis:GCI\text{-set} = \text{xtr\_UID}(\text{uid}_{GC},gcs) \equiv \{\text{uid}_{GC}(gc)|gc:GC\cdot gc \in gcs\}$ 
72  $stk\_uis:BI\text{-set} = \text{xtr\_UID}(\text{uid}_{BAY},stks) \equiv \{\text{uid}_{BAY}(stk)|stk:BAY\cdot stk \in stks\}$ 

```

73 $lt_uis: LTI\text{-set} = \text{xtr_UID}(\text{uid_LT}, \text{ltss}) \equiv \{\text{uid_LT}(lt) \mid lt: LT \cdot lt \in \text{ltss}\}$
 74 $uis: (V \mid QCI \mid QTI \mid GCI \mid BSI \mid LTI)\text{-set} = \cup \{ v_uis, qc_uis, qt_uis, sc_uis, stk_uis, lt_uis \}$

Unique CCI Identifier to Part-set Maps

75. the map, ccm , from command center identifiers into the identifiers of respective command centers;
 76. the map, qcm , from command center identifiers into the set of quay cranes of respective ports;
 77. the map, qtm , from command center identifiers into the set of quay trucks of respective ports;
 78. the map, qtm , from command center identifiers into the set of quay trucks of respective ports; and
 79. the map, bsm , from command center identifiers into the set of bays (i.e., “stacks”) of respective ports.

value

75 $ccm: (TPI \rightarrow_m CCI) = [\text{uid_TP}(tp) \mapsto \text{uid_CC}(\text{obs_CC}(tp)) \mid tp: TP \cdot tp \in \text{tps}]$
 76 $qcm: (CCI \rightarrow_m QC\text{-set})$
 76 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{obs_QCs}(\text{obs_QCS}(tp)) \} \mid tp: TP \cdot tp \in \text{tps}]$
 77 $qtm: (CCI \rightarrow_m QT\text{-set}) =$
 77 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{obs_QTS}(\text{obs_QTS}(tp)) \} \mid tp: TP \cdot tp \in \text{tps}]$
 78 $gcm: (CCI \rightarrow_m GC\text{-set})$
 78 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{obs_GCs}(\text{obs_SGC}(tp)) \} \mid tp: TP \cdot tp \in \text{tps}]$
 79 $bsm: (CCI \rightarrow_m BS\text{-set})$
 79 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{obs_BAYS}(\text{obs_BAYS}(\text{obs_CSA}(tp))) \} \mid tp: TP \cdot tp \in \text{tps}]$

Unique CCI Identifier to Part Identifier Maps

80. the map, qcm , from command center identifiers into the set of quay cranes of respective ports;
 81. the map, qtm , from command center identifiers into the set of quay trucks of respective ports;
 82. the map, qtm , from command center identifiers into the set of quay trucks of respective ports; and
 83. the map, bsm , from command center identifiers into the set of bays (i.e., “stacks”) of respective ports.

value

80 $iqcm: (CCI \rightarrow_m QCI\text{-set})$
 80 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{uid_QC}(qc) \mid qc: QC \cdot qc \in \text{obs_QCs}(\text{obs_QCS}(tp)) \} \mid tp: TP \cdot tp \in \text{tps}]$
 81 $iqtm: (CCI \rightarrow_m QTI\text{-set}) =$
 81 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{uid_QT}(qt) \mid qt: QT \cdot qt \in \text{obs_QTS}(\text{obs_QTS}(tp)) \} \mid tp: TP \cdot tp \in \text{tps}]$
 82 $igcm: (CCI \rightarrow_m GCI\text{-set})$
 82 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{uid_GC}(gc) \mid gc: GC \cdot gc \in \text{obs_GCs}(\text{obs_SGC}(tp)) \} \mid tp: TP \cdot tp \in \text{tps}]$
 83 $ibsm: (CCI \rightarrow_m BSI\text{-set})$
 83 $= [ccm(\text{uid_TP}(tp)) \mapsto \{ \text{uid_B}(b) \mid b: BAY \cdot b \in \text{obs_BAYS}(\text{obs_BAYS}(\text{obs_CSA}(tp))) \} \mid tp: TP \cdot tp \in \text{tps}]$

Some Axioms on Uniqueness

TO BE WRITTEN

G.3.10 Mereology

Land Truck Mereology

84. Land trucks are
- physically “connectable” to gantry cranes – of any port, and
 - conceptually connected to the command centers of any terminal port.

type

84 $\text{Mereo_LT} = \text{GCI-set} \times \text{CCI-set}$

value

84 $\text{mereo_LT}: \text{LT} \rightarrow \text{Mereo_LT}$

Gantry Crane Mereology

85. Gantry cranes are associated with
- bay stacks: each bay stack has its own gantry crane,
 - any quay truck,
 - and to the command center,
- all of the terminal at which they are located, and to
- any land truck.

type

85 $\text{Mereo_GC} = \text{s_bi}: \text{BI} \times \text{s_qtis}: \text{QTI-set} \times \text{s_cci}: \text{CCI} \times \text{s_ltis}: \text{LTI-set}$

axiom

85 $\forall (bi, qtis, cci, ltis): \text{Mereo_GC} \cdot qtis \neq \{\} \wedge ltis = ltis$

value

85 $\text{mereo_GC}: \text{GC} \rightarrow \text{Mereo_GC}$

axiom

85 $\forall tp: \text{TP} \cdot$

85 **let** $\text{csa} = \text{obs_CSA}(tp),$

85 $\text{bays} = \text{obs_BAYS}(\text{obs_BAYS}(\text{csa})),$

85 $\text{gts} = \text{obs_QTs}(\text{obs_SQT}(tp)),$

85 $\text{gcs} = \text{obs_GCs}(\text{obs_SCG}(tp))$ **in**

85 $\forall gc: \text{GC} \cdot gc \in \text{gcs} \Rightarrow$

85 **let** $(bi, qtis, cci, ltis) \text{ mereo_GC}(gc)$ **in**

85 $bi \in \text{xtr_BIs}(\text{bays}) \wedge qtis \subseteq \text{xtr_QTIs}(gts)$ **end end**

Bay Stack Mereology

86. Bay stacks, i.e., the elements of terminal stowage areas, are
- physically connected to a specific gantry crane and
 - conceptually connected to the command center
- of their terminal port.

type

86 $\text{Mereo_BS} = \text{GCI} \times \text{CCI}$

value

86 $\text{mereo_BS}: \text{BS} \rightarrow \text{Mereo_BS}$

axiom

86 $\forall \dots$

Quay Truck Mereology

87. Quay trucks are
- physically “connectable” to quay and gantry cranes and are
 - conceptually connected to the command center of the terminal port of which they are a part.

type

87 $\text{Mereo_QT} = \text{QCI-set} \times \text{GCI-set} \times \text{CCI}$

value

87 $\text{mereo_QT}: \text{QT} \rightarrow \text{Mereo_QT}$

Quay Crane Mereology

88. Quay cranes are
- conceptually related to the command center of the terminal in which they are located and are
 - physically “connectable” to any of the quay trucks of the terminal to which they belong and to any vessel of the container line industry.

type

88 $\text{Mereo_QC} = \text{QTI-set} \times \text{CCI} \times \text{VI-set}$

value

88 $\text{mereo_QC}: \text{QC} \rightarrow \text{Mereo_QC}$

Vessel Mereology

89. Container vessels can potentially visit any container terminal port, hence have as their mereology,
- a set of terminal port command center identifiers, and
 - a set of quay crane identifiers of any terminal port.

type

89 $\text{Mereo_V} = \text{CCI-set} \times \text{QCI-set}$

value

89 $\text{mereo_V}: \text{V} \rightarrow \text{Mereo_V}$

axiom

89 $\forall v:V \cdot \mathbf{let} (ccis, qcis) = \text{mereo_V}(v) \mathbf{in}$

89 $ccis = ccis \wedge qcis = qcis \mathbf{end}$

89

Command Center Mereology

Command centers are basically conceptual quantities. Hence we can expect the physical mereology to be the conceptual mereology.

90. Command centers are physically and conceptually connected to all vessels, all quay and gantry cranes of the terminal port of the command center, all quay trucks of the terminal port of the command center, all stacks (i.e., bays) of the terminal port of the command center, all land trucks, and all containers.

```

type
90 Mereology = VI-set × QCI-set × QTI-set × GCI-set × BI-set × LTI-set × CI-set
value
90 mereology: CC → Mereology
axiom
90 ∀ tp:TP • tp ∈ tps •
90   let qcs:QC-set • qcs = obs_QCs(obs_QCS(tp)),
90       qts:QT-set • qts = obs_QTs(obs_QTS(tp)),
90       gcs:GC-set • gcs = obs_GCs(obs_SGC(tp)),
90       bs:BAY-set • bs = obs_Bs(obs_BS(obs_CSA(tp))) in
90   let vis:VI-set • vis = {uid_VI(v)|v:V•v ∈ vs},
90       qcis:QCI-set • qcis = {uid_QCI(qc)|qc:QC•qc ∈ qcs},
90       qtis:QTI-set • qtis = {uid_QTI(qt)|qt:QT•qt ∈ qts},
90       scis:SCI-set • scis = {uid_SCI(sc)|sc:SC•sc ∈ scs},
90       bis:iBAY-set • bis = {uid_BI(b)|b:iBAY•b ∈ bs},
90       ltis:LTI-set • ltis = {uid_LTI(lt)|lt:LT•lt ∈ lts},
90       cis:SCI-set • cis = {uid_CI(c)|c:C•c ∈ cs} in
90   mereology(obs_CSA(tp)) = (vis,qcis,scis,qtis,bis,ltis,cis) end end

```

Container Mereology

91. Containers are physically connectable to any

- land truck,
- gantry crane,
- bay stack,
- quay truck,
- quay crane, and
- vessel

and conceptually connected to any

- command center.

```

type
91 Mereology_C = LTI-set × GCI-set × BSI-set × QTI-set × QCI-set × VI-set × CCI-set
value
91 mereology_C: C → Mereology_C

```

G.3.11 Attributes

92. Common to all vessel and terminal stacks is that they all potentially hold tagged containers, usually several.

93. The predicate `is_C` applied to a hold yields **true** if what is ‘held’ is not `nil`.

94. The function `get_hi` applied to a hold yields a container history if what is ‘held’ is not `nil`.

95. The function `get_C` applied to a hold yields a container if what is ‘held’ is not `nil`.

96. The function `get_CI` applied to a hold yields the unique identifier of a container if what is ‘held’ is not `nil`.

Land trucks also hold tagged containers, zero or one! Gantry cranes, quay trucks and quay cranes temporarily holds tagged containers. All: trucks and cranes pass on tagged containers when they communicate with one another and gantry cranes with bay stacks: land trucks with gantry cranes, gantry cranes with bay stacks and quay trucks, quay trucks with quay cranes, and quay cranes with vessels, i.e., with vessel bays.

```

type
92 Hold == mkNil(s_c:"nil") | mkTagC(s_chi:C_Hi,s_c:C)
value
93 is_C: Hold → Bool
93 is_C(h) ≡ s_c(h) ≠ "nil"
94 get_hi: Hold  $\tilde{\rightarrow}$  C_Hi
94 get_hi(h) ≡ s_chi(h) pre is_C(h)
95 get_C: Hold  $\tilde{\rightarrow}$  C
95 get_C(h) ≡ s_c(h) pre is_C(h)
96 get_CI: Hold  $\tilde{\rightarrow}$  CI
96 get_CI(h) ≡ uid_C(get_C(h)) pre is_C(h)

```

Land Truck Attributes

A land truck

- 97. is *statically* either a 20" (twenty foot) or a 40" (forty foot) container.
- 98. *programmably* holds a tagged container or not.
- 99. Land trucks also possess a further undefined *programmable* land truck state.

Note that we do not here model the position of land trucks.

```

type
97 LTFeet == mkTwenty("twenty") | mkForty("forty")
98 LTHold = Hold
99 LTΣ
value
97 attr_LTFeet: LT → LTFeet
98 attr_LTHold: LT → LTHold
99 attr_LTΣ: LT → LTΣ

```

Summary:

```

value
  sta_attrs_LT(lt) ≡ ...
  pro_attrs_LT(lt) ≡ (attr_LTHold(lt),attr_LTΣ(lt))
  mon_attrs_LT(lt) ≡ ...

```

Gantry Crane Attributes

At any one time a gantry crane

- 100. Gantry cranes are *statically* associated with a terminal bay stack⁸.

Note that we do not here model the position of gantry cranes as they move along a bay stack.

```

type
100 GCPos = BI
value
100 attr_GCPos: GC → GCPos

```

⁸ Associated means: they are stationed at a specific bay stack, but can move along the rows of that bay stack.

Summary:**value**

```
sta_attrs_GC(gc) ≡ attr_GCPos(gc)
```

Bay Stack Attributes

101. A bay stack, bs:BS, is a bay.

102. Bay stacks have as a static attribute bay descriptors, cf. Item 39 [Page 445]; 49 [Page 447],

103. and as programmable attribute a map of row identifiers into rows, cf. Item 49 [Page 447].

type

101 BS = BAY

102 BD

103 Rows = RI \rightarrow ROW

value

101 attr_BD: BS \rightarrow BD

102 attr_Rows: BS \rightarrow Rows

Summary:**value**

```
sta_attrs_BS(bs) ≡ attr_BD(bs)
```

```
pro_attrs_BS(bs) ≡ attr_Rows(bs)
```

Quay Truck Attributes

We omit consideration of quay truck attributes (!).

Quay Crane Attributes

At any one time a quay crane

104. may *programmably* be positioned at a bay of a vessel.

type

104 QCPos == mkNilPos("nil") | mkVeBay(vi:VI,bi:BI)

value

104 attr_QCPos: QC \rightarrow QCPos

Summary:**value**

```
pro_attrs_QC(qc) ≡ attr_QCPos(qc)
```


Vessel Attributes

Container vessels have the following attributes:

- 105. a static attribute of container stowage description;
- 106. a programmable attribute of all containers carried: in the form of a map from each carried container identifier to its corresponding way bill and bill of lading;
- 107. a programmable attribute reflecting containers stowed;
- 108. and a programmable attribute state.

There may be other vessel attributes.

type

- 105 StowDescr = BD
- 106 ContDescr = CI \rightarrow (WB \times BoL)
- 107 Stowage = CSA
- 108 $V\Sigma$

value

- 105 attr_StowDescr: $V \rightarrow$ StowDescr
- 106 attr_ContDescr: $V \rightarrow$ ContDescr
- 107 attr_Stowage: $V \rightarrow$ Stowage
- 108 attr_ $V\Sigma$: $V \rightarrow V\Sigma$

Summary:

value

- sta_attrs_ $V(v) \equiv \dots$
- pro_attrs_ $V(v) \equiv (\text{attr_StowDescr}(v), \text{attr_ContDescr}(v), \text{attr_Stowage}(v), \text{attr_}V\Sigma(v))$
- mon_attrs_ $V(v) \equiv \dots$

Command Center Attributes

- 109. The syntactic description⁹ of the terminal state, i.e., the actual positions and deployment of vessels at quays, quay and stack cranes, quay and land trucks, and the actual container “contents” of these. $TP\Sigma\text{Descr}$, is a *programmable* attribute.

type

- 109 $TP\Sigma\text{Descr}$

value

- 109 attr_Term ΣDescr : $CC \rightarrow TP\Sigma\text{Descr}$

Summary:

value

- pro_attrs_ $CC(cc) \equiv \text{attr_Term}\Sigma\text{Descr}(cc)$

⁹ The syntactic description of the terminal state is, of course, not that state, but only its description. The terminal state is the combined states of all cranes, trucks and the container storage area.

Container Attributes

We have already, in Sect. G.3.5 [Page 444], dealt with some notion related to container attributes. For this report that should suffice.

G.4 Perdurants

G.4.1 A Diagram

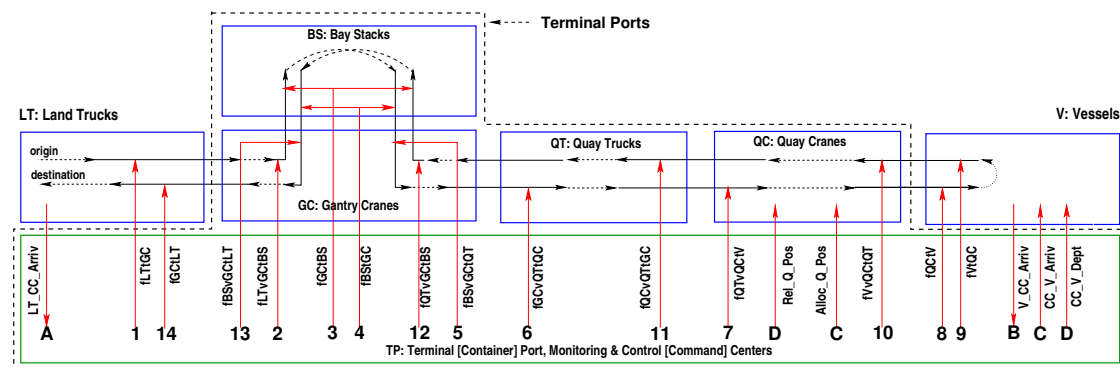


Fig. G.3. Terminal Port Behaviours

The flow of arrows, sometimes dashed (---), sometimes not dashed (—), left-to-right and right-to-left (←, →), even angled arrows, not emanating from or incident upon the command center, designate the flow (and hence transport) of containers. In Sect. G.3 (*Endurants*) we model containers as parts. In this, the *Perdurants* section, we do not model containers as behaviours, but as elements of programmable attributes of trucks, cranes, bay stacks and vessels. The diagram of Fig. G.3 [Page 457] shall further be understood as follows: There are two or more terminal ports and for each of these there are a number of quay cranes, quay trucks, one terminal container stowage area with one or more bay stacks and a corresponding number of gantry cranes, one per bay stack. There are a number of vessels and a number of land trucks.

The boxed numerals, 1–14, designate the ordered movements of containers: from land trucks onto vessels, 1–8, and from vessels onto land trucks, 9–14. The boxed letters, A–C, designate the arrival and departure of vessels, and their resulting actions.

G.4.2 Very Brief Overview

In Fig. G.3 [Page 457] the vertical arrows labeled 2–15, from the command center to any of the trucks, cranes, terminal port stowages or vessels, denote directives that the (arrow incident) behaviours are expected to fulfill. The vertical arrows 1 and A (incident upon the command center(s)) denote directives that the command center behaviour are expected to fulfill.

- Fulfillment of directive A is manifested in the sequenced issue of either directives
 - ⊗ 1, 2, 3, and in that order,
 - ⊗ resulting in the transfer of a specific container
 - ⊗ from a land truck to a terminal port stack position;
 - or directive:
 - ⊗ 3, 13, 14, and in that order,

- ⊗ resulting in the transfer of a specific container
 - ⊗ from a terminal port stack position to a land truck.
- The former results, sooner-or-later, from the arrival of a land truck with a container in its hold; the latter from the arrival of a land truck with no ("nil") container in its hold.
- Fulfillment of directive **B** is manifested in a set of zero, one or more sequenced issues of
 - ⊗ either directives **9, 10, 11, 12, 3,**
 - ⊗ or directives **4, 5, 6, 7, 8,**
 - ⊗ or both –
 - ⊗ resulting in the transfer of zero, one or more containers
 - ⊗ from a vessel to bay stacks, and/or
 - ⊗ from bay stacks to the vessel –
 - ⊗ where these directives (**4, 5, 6, 7, 8, 9, 10, 11, 12, 3**) occur in any interleaved order as directing different container transfers.

G.4.3 Short Behaviour Narratives

The vertical arrows of Fig. G.3 [Page 457] are labeled. The general form of these labels is **fAvBtC**, an abbreviation for **from A via B to C**.

110. Land Truck:

- a. **1 LT_CC_arriv.:** Land Truck to Command Center – Arrival:
 - a **land truck advises** a **[terminal port monitoring & control] command center** of its arrival (incl. WB and BoL¹⁰).
- b. **2 fLTtGC:** From Land Truck to Gantry Crane:
 - i. **container moves** (→) (from some origin) by **land truck**
 - ii. and is **transferred** () to a **gantry crane**.
- c. **15 fGCtLT:** From Gantry Crane to Land Truck:
 - i. **container** is **transferred** () from **gantry crane** to **land truck**;
 - ii. and **moves** (→) by it (to some destination)

111. Gantry Crane:

- a. **3 fLTvGCtBS:** From Land Truck via Gantry Crane to Bay Stack:
 - i. **container** is **transferred** (→) from **land truck** to **gantry crane**;
 - ii. gantry crane with container **moves** (→) from land truck to **bay stack** position;
 - iii. and is **transferred** () to **bay stack**.
- b. **14 fBSvGCtLT:** From Bay Stack via Gantry Crane to Land Truck:
 - i. **gantry crane** with **container moves** from **bay stack** position to **land truck**;
 - ii. **container** is **transferred** from **gantry crane** to **land truck**;
 - iii. and is **transferred** () to **land truck**.
- c. **13 fQTvGCtBS:** From Quay Truck via Gantry Crane to Bay Stack:
 - i. **container** is **transferred** from **quay truck** to **gantry crane**;
 - ii. gantry crane with container **moves** from quay truck to **bay stack** position;
 - iii. and is **transferred** () to **bay stack**.
- d. **6 fBSvGCtQT:** From Bay Stack via Gantry Crane to Quay Truck:
 - i. **container** is **transferred** () from **bay stack** to **gantry crane**;
 - ii. **gantry crane** with **container moves** (→) from **bay stack** position to **quay truck**;
 - iii. and **container** is **transferred** () from **gantry crane** to **quay truck**.

112. Bay Stack:

- a. **4 fGCtBS:** From Gantry Crane to Bay Stack:
 - **container** is **transferred** from **gantry crane** to **bay stack**.
- b. **4 fBSStGC:** From Bay Stack to Gantry Crane:

¹⁰ WB: Way Bill, BoL: Bill of Lading

- **container** is **transferred** from **bay stack** to **gantry crane**.
113. **Quay Truck:**
- a. **7 fGCvQTtQC:** From Gantry Crane via Quay Truck to Quay Crane:
 - i. **container** is **transferred** from **stack crane** to **quay truck**;
 - ii. then **moves** by **quay truck**;
 - iii. and is **transferred** from **quay truck** to **quay crane**.
 - b. **12 fQCvQTtGC:** From Quay Crane via Quay Truck to Gantry Crane:
 - i. **container** is **transferred** from **stack crane** to **quay truck**;
 - ii. then **moves** by **quay truck**;
 - iii. and is **transferred** from **quay truck** to **gantry crane**.
114. **Quay Crane:**
- a. **8 fQTvQCtV:** From Quay Truck via Quay Crane to Vessel:
 - i. **container** is **transferred** from **quay truck** to **quay crane**;
 - ii. then **moves** by **quay crane**;
 - iii. and is **transferred** from **quay crane** to **vessel**.
 - b. **11 fVvQCtQT:** From Vessel via Quay Crane to Quay Truck:
 - i. **container** is **transferred** from **vessel** to **quay crane**;
 - ii. then **moves** by **quay crane**;
 - iii. and is **transferred** from **quay crane** to **quay truck**.
115. **Vessel:**
- a. **A V_CC_arriv.:** Vessel to Command Center – Arrival:
 - a **vessel** informs **command center** of its pending arrival – providing a full account of its containers
 - b. **B CC_V_arriv.:** Command Center to Vessel, Arrival:
 - **command center** informs the **vessel** of its **quay position**
 - c. **9 fQCtV:** From Quay Crane to Vessel:
 - **container** is **transferred** from **quay crane** to **vessel**.
 - d. **10 fVtQC:** From Vessel to Quay Crane:
 - **container** is **transferred** from **vessel** to **quay crane**.
 - e. **C CC_V_dept.:** Command Center to Vessel – Departure:
 - **command center** informs **vessel** that it can leave

G.4.4 Actions

Land Truck Actions

116. , cf. Item 208 [Page 469].
 117. , cf. Item 208a [Page 469].
 118. , cf. Item 208c [Page 469].
 119. , cf. Item 211c [Page 470].
 120. , cf. Item 212c [Page 470].

116 approaching_terminal_port: $LT\Sigma \times \text{POSITION} \rightarrow \text{Bool}$
 116 approaching_terminal_port($lt\sigma, pos$) **as** tf; **pre ... post ...**

117 observe_terminal_CCI: $LT\Sigma \times \text{POSITION} \rightarrow \text{CCI}$
 117 observe_terminal_CCI($lt\sigma, pos$) **as** cci; **pre ... post ...**

118 arriv_update: $\text{CCI} \times LT\Sigma \rightarrow LT\Sigma$
 118 arriv_update($cci, lt\sigma$) $\equiv lt\sigma'$; **pre ... post ...**

```

119 lt_update_LT_to_GC:  $LT\Sigma \rightarrow LT\Sigma$ 
119 lt_update_LT_to_GC( $lt\sigma$ ) as  $lt\sigma'$ ; pre ... post ...

120 lt_gc_to_lt_update:  $GCI \times LT\Sigma \rightarrow LT\Sigma$ 
120 lt_gc_to_lt_update( $gci, lt\sigma$ ) as  $lt\sigma'$ ; pre ... post ...

```

Container Stowage Actions

We define two operations on CSAs:

- 121. one of stacking (loading) a tagged container,
- 122. and one of unstacking (unloading) a tagged container;
- 123. both operations involving bay/row/stack and container references.

type

```
123 BRSI = BI  $\times$  RI  $\times$  SI
```

value

```

121 load_TagC: CSA  $\times$  BRSIP  $\times$  TagC  $\rightarrow$  Nat  $\times$  CSA
121 load_TagC( $csa, (bi, ri, si, pos), tagc$ ) as ( $pos, csa'$ )
121   pre:  $\mathcal{P}_{load}(csa, (bi, ri, si, pos), tagc)$ 
121   post:  $\mathcal{Q}_{load}(csa, (bi, ri, si), tagc)(pos', csa')$ 
121
122 unload_TagC: CSA  $\times$  BRSIP  $\xrightarrow{\sim}$  (Nat  $\times$  TagC)  $\times$  CSA
122 unload_CI( $csa, (bi, ri, si), ci$ ) as ( $(pos, tagc), csa'$ )
122   pre:  $\mathcal{P}_{unload}(csa, (bi, ri, si), tagc)$ 
122   post:  $\mathcal{Q}_{unload}(csa, (bi, ri, si), ci)((pos, tagc), csa')$ 

```

The Load Pre-/Post-Conditions

- 124. The $csa:CSA$ must be well-formed;
- 125. the tagged container, $tagc$, must not be embodied in that csa ; and
- 126. the bay/row/stack/position reference, (bi, ri, si) must be a top one of the container stowage area.

value

```

121  $\mathcal{P}_{load}(csa, (bi, ri, si, pos), tagc) \equiv$ 
124   well_formed( $csa$ )
125    $\wedge tagc \notin xtr\_TagCs(csa)$  note:  $xtr\_TagCs$  to be defined
126    $\wedge valid\_BRSI(bi, ri, si, pos)(csa)$ 

126 valid_BRS: BRSIP  $\rightarrow$  CSA  $\rightarrow$  Bool
126 valid_BRS( $bi, ri, si, pos$ )( $vir\_csa$ )  $\equiv$ 
126    $bi \in \mathbf{dom}csa \wedge ri \in \mathbf{dom}csa(bi) \wedge si \in \mathbf{dom}(csa(bi))(ri) \wedge pos = \mathbf{len}((csa(bi))(ri))(si)$ 

```

The Unload Pre-/Post-Conditions

- 127. The csa
- 128. must be wellformed; and
- 129. the bay/row/stack reference, (bi, ri, si) must be one of the container stowage area.

value

```

127  $\mathcal{P}_{unload}(csa, (bi, ri, si)) \equiv$ 
128   well_formed( $csa$ )
129    $\wedge valid\_BRSI(bi, ri, si)(csa)$ 

```

Common Aspects of the unload/load Actions

130. Common to the expression of post conditions for both the load and unload functions are
- that the “before” and “after” bay, row and stack identifiers are the same, and
 - that those bays, rows and stacks which are not designated by the load and unload functions are also the same.

value

$$\begin{aligned}
 130 \quad & \text{common_csa}(\text{csa}, (\text{bi}, \text{ri}, \text{si}), \text{tagc})(\text{csa}') \equiv \\
 130a \quad & \mathbf{dom} \text{ csa} = \mathbf{dom} \text{ csa}' \\
 130b \quad & \wedge \forall \text{ bi}' : \text{BI} \cdot \text{bi}' \in \mathbf{dom} \text{ csa} \setminus \{\text{bi}\} \cdot \text{csa}(\text{bi}') = \text{csa}'(\text{bi}') \\
 130a \quad & \wedge \mathbf{dom} \text{ csa}(\text{bi}) = \mathbf{dom} \text{ csa}'(\text{bi}) \\
 130b \quad & \wedge \forall \text{ ri}' : \text{RI} \cdot \text{ri}' \in \mathbf{dom} \text{ row} \setminus \{\text{ri}\} \cdot \text{row}(\text{ri}') = \text{row}'(\text{ri}') \\
 130a \quad & \wedge \mathbf{dom} (\text{csa}(\text{bi}))(\text{ri}) = \mathbf{dom} (\text{csa}')(\text{ri}) \\
 130b \quad & \wedge \forall \text{ si}' : \text{SI} \cdot \text{si}' \in \mathbf{dom} \text{ stk} \setminus \{\text{si}\} \cdot \text{stk}(\text{si}') = \text{stk}'(\text{si}')
 \end{aligned}$$

The load Action

131. The load **post** condition has
- the “before” and “after” csas satisfy the `common_csa` predicate, and
 - the identified “after” stack to be the “concatenation” of the tagged container, `tagc`, with the “before” stack.

value

$$\begin{aligned}
 131 \quad & \mathcal{Q}_{load}(\text{csa}, (\text{bi}, \text{ri}, \text{si}), \text{tagc})(\text{pos}, \text{csa}') \equiv \\
 131a \quad & \text{common_csa}(\text{csa}, (\text{bi}, \text{ri}, \text{si}), \text{tagc})(\text{csa}') \\
 131b \quad & \wedge ((\text{csa}'(\text{bi}))(\text{ri}))(\text{si}) = \langle \text{tagc} \rangle \wedge ((\text{csa}(\text{bi}))(\text{ri}))(\text{si}) \wedge \text{pos} = \mathbf{len} \text{ stk}
 \end{aligned}$$

The unload Action

132. The unload **post** condition is satisfied
- if the “before” and “after” csas satisfy the `common_csa` predicate, and
 - if the identified “before” stack is the “concatenation” of the tagged container, `tagc`, with the “after” stack.

$$\begin{aligned}
 132 \quad & \mathcal{Q}_{unload}(\text{csa}, (\text{bi}, \text{ri}, \text{si}), \text{tagc})(\text{pos}, \text{csa}') \equiv \\
 132a \quad & \text{common_csa}(\text{csa}, (\text{bi}, \text{ri}, \text{si}), \text{tagc})(\text{csa}') \\
 132b \quad & \wedge ((\text{csa}(\text{bi}))(\text{ri}))(\text{si}) = \langle \text{tagc} \rangle \wedge ((\text{csa}'(\text{bi}))(\text{ri}))(\text{si}) \wedge \text{pos} = \mathbf{len} \text{ stk}
 \end{aligned}$$

Vessel Actions

TO BE WRITTEN

Command Center Actions

The command center

133. either monitors *alerts* from land trucks and vessels:
134. **A LT_CC_arriv** [cf. Item 110a [Page 458]] The land truck informs a command center of its imminent arrival.

value

208 [Page 469] LT_Arriv arrival alert from land trucks

135. **B CC_V_Arriv** [cf. Item 115b [Page 459]] A vessel informs a command center of its imminent arrival.

value

232a [Page 477] V_CC_Arriv arrival alert from vessels

136. or controls trucks, cranes, terminal port bay stacks and vessels by *directives*.

137. The *next_move* action (internally non-deterministically) calculates a set of next responses to be sent to land trucks (Items 142–143), gantry cranes (Items 144–147), bay stacks (Items 148–149), quay trucks (Items 150–151), quay cranes (Items 152–155) and vessels (Items 158–159).

138. The *next_move* action takes the command center state and yields a pair of

- a new state (which records the fact that this calculation has taken place, including its result), and
- the truck, crane, vessel or stack targeted (cf. UI) *Directives*.

value

137 next_moves: $CC\Sigma \rightarrow CC\Sigma \times (UI \times Directive)\text{-set}$

138 next_moves($cc\sigma$) as ($cc\sigma', \text{next_dirs}$)

140 **pre:** $\mathcal{P}(cc\sigma)$

141 **post:** $\mathcal{Q}(cc\sigma)(cc\sigma', \text{next_dirs})$

139. The *Directive* consists of a pair of

- the *unique identifier* of the behaviour to which the *next move* is directed (a land truck, a gantry crane, a bay stack, a quay truck, a quay crane, or a vessel), and
- the *next move* details.

140. The pre-condition of the *next move* action expresses that the command center state is well-formed.¹¹

141. The post-condition of the *next move* action expresses a relationship between and “input” and [the well-formed] “output” command center states and the directive, including that the [new] directive is, somehow, recorded in the result (i.e., “output”) state.¹²

We discuss the *Directive* details in Items 134–159 below¹³.

142. **1 FLTtGC** [cf. Item 110b [Page 458]] The command center informs a land truck of the identity of the gantry crane to which to deliver its identified hold.

143. **14 fGCtLT** [cf. Item 110c [Page 458]] The command center informs a land truck of the identity of the gantry crane from which to fetch an identified container.

144. **2 fLTvGCtBS** [cf. Item 111a [Page 459]] The command center informs a gantry crane that it may fetch a designated container from a designated land truck and deliver it to a designated bay-row-stack-position of the bay stack served by the gantry crane.

145. **13 fBSvGCtLT** [cf. Item 111b [Page 459]] The command center informs a gantry crane that it may fetch a designated bay-row-stack-position of the bay stack served by the gantry crane and deliver it to a designated land truck.

146. **12 fQTvGCtBS** [cf. Item 111c [Page 459]] The command center informs a gantry crane that it may fetch a designated container from a designated quay truck and deliver it to a designated bay-row-stack-position of the bay stack served by the gantry crane.

147. **5 fBSvGCtQT** [cf. Item 111d [Page 459]] The command center informs a gantry crane that it may deliver a container from a designated bay-row-stack-position of the bay stack served by the gantry crane to a designated quay truck.

¹¹ We shall not detail the \mathcal{P} predicate.

¹² We shall also not detail the \mathcal{Q} predicate.

¹³ The reader may be helped in grasping Items 134–159 by tracing the mostly horizontal ($\rightarrow\text{---}\rightarrow$) arrows of Fig. G.3 [Page 457].

148. **3 fGCtBS** [cf. Item 112a [Page 459]] The command center informs a designated bay stack that it may accept a bay-row-stack-position designated and destined container from a designated gantry crane.
149. **4 fBStGC** [cf. Item 112b [Page 459]] The command center informs a designated bay stack that it may deliver, from a designated bay-row-stack-position bay stack to a designated gantry crane.
150. **6 fGCvQTtQC** [cf. Item 113a [Page 459]] The command center informs a designated quay truck that it may accept a designated container from a designated gantry crane and to be delivered to a designated quay crane.
151. **11 fQCvQTtGC** [cf. Item 113b [Page 459]] The command center informs a designated quay truck that it may deliver a designated container to a designated gantry crane from a designated quay crane.
152. **7 fQTvQCtV** [cf. Item 114a [Page 459]] The command center informs a designated quay crane that it may accept a designated container from a designated quay truck to a designated vessel.
153. **10 fVvQCtQT** [cf. Item 114b [Page 459]] The command center informs a designated quay crane that it may fetch a designated container from a designated vessel and deliver it to a designated quay truck.
154. **C Alloc_Q_Pos** The command center informs a designated quay cranes of the allocation of their vessel and bay position – with as many such messages as there are quay cranes to be allocated.
155. **D Rel_Q_Pos** The command center informs a designated quay crane of the release of their vessel and bay position – with as many such messages as there are quay cranes to be released.
156. **8 fQCtV** [cf. Item 115c [Page 459]] The command center informs a designated vessel that it may accept a designated container from a designated quay crane and stow it into a designated bay-row-stack-position of that vessel.
157. **9 fVtQC** [cf. Item 115d [Page 459]] The command center informs a designated vessel that it may deliver a designated container from a designated bay-row-stack-position of that vessel to a designated quay crane.
158. **C CC_V_Arriv** [cf. Item 115b [Page 459]] The command center informs a designated vessel of its quay crane position list¹⁴.
159. **D CC_V_Dept** [cf. Item 115e [Page 459]] The command center informs a designated vessel that it may depart from the terminal port

The next_move action is undoubtedly the most crucial action of the entire container line industry as here modelled. We shall not attempt to characterise this action nor the command center state other than indirectly as done in the narrative text Items 142–159 above. For the use of :: above we refer to the **Note** in the RSL Primer on Page 487.

type				
139	Directive ==	LT_CC_Arriv fLTtoGC fGCtLT	Bay Stacks	148 fGCtBS :: BRSIP × CI
139		fLTvGCtBS fBSvGCtLT		149 fBStGC :: BRSIP × CI
139		fQTvGCtBS fBSvGCtQT	Quay Trucks	
139		CC_V_Arriv CC_V_Dept		152 fGCvQTtQC :: GCI × QCI × CI
139		Alloc_Q_Pos Rel_Q_Pos		153 fQCvQTtGC :: QCI × GCI × CI
139		fQTvQCtV fVvQCtQT	Quay Cranes	
139		fQCtV fVtQC		150 fQTvQCtV :: QTI × VI × CI
	Land Trucks			151 fVvQCtQT :: VI × QTI × CI
134	LT_CC_Arriv	:: (WB mkNil("nil")) × CI		154 Alloc_Q_Pos :: QCI × VI × BI
142	fLTtGC	:: GCI × CI		155 Rel_Q_Pos :: QCI × VI × BI
143	fGCtLT	:: GCI × CI	Vessels	
	Gantry Cranes			156 fQCtV :: QCI × BRSIP × CI
144	fLTvGCtBS	:: LTI × CI		157 fVtQC :: BRSIP × QCI × CI
145	fBSvGCtLT	:: LTI × CI		135 CC_V_Arriv :: VI × StowDescr × ContDescr
146	fQTvGCtBS	:: QTI × CI		158 CC_V_Arriv :: QCI*
147	fBSvGCtQT	:: QTI × CI		159 CC_V_Dept :: StowDescr × ContDescr

¹⁴ Elements of such lists designated adjacent quay cranes

G.4.5 Channels

We decide to model the channels as one array channel where indices distinguish who is “connected” to who!

Channel Declarations

160. Land trucks communicate LTCC messages with the terminal port [monitoring & control] command center: informing of arrival and being informed of departure.
161. Land trucks, gantry cranes, terminal bay stacks, quay trucks, quay cranes and vessels receive messages from the terminal port [monitoring & control] command centers: informing them of their next actions.
162. Land trucks communicate containers with the gantry cranes of any terminal port.
163. Gantry cranes further communicate containers with the bay stacks of their terminal.
164. Gantry Cranes further communicate containers with the quay trucks of their terminal.
165. Quay trucks further communicate containers with the quay cranes of their terminal.
166. Quay cranes communicate containers with any vessels.
167. Vessels further communicate VCC messages with command centers of any terminal: informing of arrival and being informed of departure.

value

$uis:(LTI|GCI|BI|QTI|QCI|VI)\text{-set} = Itis \cup gcis \cup bis \cup qtis \cup qcis \cup vis$

channel

- 160 { ch[{ lti,cci }] | lti:LTI,cci:CCI•lti $\in Itis \wedge cci \in ccis$ } MSG_LTCC
- 161 { ch[{ ui,cci }] | ui:(LTI|GCI|BI|QTI|QCI|VI),cci:CCI•ui $\in uis \wedge cci \in ccis$ } Directive
- 162 { ch[{ lti,gci }] | tpi:TPI,lti:LTI,gci:GCI•tpi $\in tpics \wedge lti \in Itis \wedge gci \in gcim(tpi)$ } Hold
- 163 { ch[{ gci,bi }] | tpi:TPI,gci:GCI,bi:BI•gci $\in gcim(tpi) \wedge bi \in bsim(tpi)$ } Hold
- 164 { ch[{ gci,qti }] | tpi:TPI,qci:QCI,qti:QTI•gci $\in gtim(tpi) \wedge qti \in qcim(tpi)$ } Hold
- 165 { ch[{ qti,qci }] | tpi:TPI,qti:QTI,gci:QCI•qti $\in qtim(tpi) \wedge qci \in qtim(tpi)$ } Hold
- 166 { ch[{ qci,vi }] | tpi:TPI,qci:QCI,vi:VI•qci $\in qcim(tpi) \wedge vi \in vis$ } Hold
- 167 { ch[{ vi,cci }] | vi:VI,cci:CCI•vi $\in vis \wedge cci \in ccis$ } MSG_VCC

Channel Messages

- 168.
- 169.

type

- 168 MSG_LTCC = ...
- 169 MSG_VCC = ...

G.4.6 Behaviour Signatures

Land Truck Signature

170. The signature of land truck behaviour includes the usual arguments: land truck identifier, land truck mereology, land truck static attributes (omitted), and three programmable attributes: ‘hold’, route and state.

Land trucks

171. offer/accept containers to/from gantry cranes of any terminal port, and

172. offer/accept information to/from command centers of any terminal port.

value

```
170 land_truck: lti:LTI × (gcis:GCI-set × ccis:CCI-set) × ... → (LTHold × LTΣ)
171   → in,out { ch[{lti, gci}] | gci:GCI • gcis ∈ gcis }
172   { ch[{lti, cci}] | cci:CCI • cci ∈ ccis } Unit
```

The behaviours invoked by land_truck have the same signature:

- LT_CC_arriv, Item 208, pg. 469,
- land_truck_response, Item 205c, pg. 469,
- fLTtGC, Item 110b, pg. 458, and
- fGCtLT, Item 110c, pg. 458.

Gantry Crane Signature

173. The signature of gantry crane behaviour includes the usual arguments: gantry crane identifier, gantry crane mereology, [gantry crane static attributes omitted,] and two programmable gantry crane attributes: ‘hold’ and position.

Gantry cranes

174. accept/offer containers from/to any land trucks,
 175. accept/offer containers from/to bay stacks,
 176. accept/offer containers from/to quay trucks, and
 177. accept directives from their command center.

```
173 gantry_crane: gci:GCI × (bis,qtis,ltis,cci):(BI-set × QTI-set × LTI-set × CCI)
174   → in,out { ch[{gci, lti}] | lti:LTI • lti ∈ ltis },
175   { ch[{gci, bi}] | bi:BI • bi ∈ bis },
176   { ch[{gci, qti}] | qti:QTI • qti ∈ qtis },
177   ch[{gci, cci}] Unit
173 assert: bis=ibsm(cci) ∧ qtis=iqtm(cci) ∧ ltis=ltis
```

The behaviours invoked by gantry_crane have the same signature:

- gantry_crane_lt_gc, Item 214, pg. 471,
- gantry_crane_bs_qt, Item 215, pg. 471,
- gantry_crane_qt_bs, Item 216, pg. 472, and
- gantry_crane_bs_lt, Item 217, pg. 472.

Bay Stack Signature

178. The signature of bay stack behaviours includes the usual arguments: bay stack identifier, bay stack mereology, bay stack static attributes, and the programmable bay stack attribute.

Bay stacks

179. accept/offer containers from/to the local gantry cranes and
 180. accept directives from their command center.

value

```
178 bay_stack: bsi:BI × (gcis,cci):(GCI-set × CCI) × BD → BAY
179   → in,out { ch[{bsi, gci}] | gci:GCI • gcis ∈ gcis }
180   ch[{bsi, cci}] Unit
178 assert: gcis=igcm(cci)
```

The behaviours invoked by bay_stack have the same signature:

- bay_stack_gc_bs, Item 219, pg. 473, and
- bay_stack_bs_gc, Item 220, pg. 473.

Quay Truck Signature

181. The signature of quay truck behaviours includes the usual arguments: quay truck identifier and quay truck mereology.

Quay trucks

- 182. offers/accepts containers to/from quay cranes,
- 183. offers/accepts containers to/from gantry cranes, and
- 184. accept directives from their command center.

```

181 quay_truck: qti:QTI × (qci, qti):(QCI-set × GCI-set × CCI)
182   → in,out { ch[{qci,qti}] | qci:QTI • qci ∈ qcis },
183             { ch[{gci,qti}] | gci:QCI • gci ∈ gcis },
184             ch[{qti,cci}] Unit
181 assert: qcis=iqcm(cci) ∧ gcis=gcism(cci)

```

The behaviours invoked by quay_truck have the same signature:

- quay_truck_gc_qc, Item 222, pg. 474, and
- quay_truck_qc_gc, Item 223, pg. 474.

Quay Crane Signature

185. The signature of quay crane behaviours includes the usual arguments: quay crane identifier, quay crane mereology and the programmable quay crane position vis-a-vis a vessel's bay.

Quay cranes

- 186. offers/accepts containers to/from quay trucks,
- 187. offers/accepts containers to/from vessels, and
- 188. accept directives from their command center.

```

185 quay_crane: qci:GCI × (qtis, cci, vis):(QTI-set × CCI × VI-set) × QCPos →
186   → in,out { ch[{qci,qti}] | qti:QTI • qti ∈ qtis },
187             { ch[{qci,vi}] | vi:VI • vi ∈ vis },
188             ch[{qci,cci}] Unit
185 assert: qtis=iqtm(cci) ∧ vis=vis

```

The behaviours invoked by quay_crane have the same signature:

- quay_crane_c_xfer, Item 225, pg. 475,
- quay_crane_v_qt, Item 227, pg. 475, and
- quay_crane_position, Item 228, pg. 476.

Vessel Signature

189. The signature of vessel behaviours includes the usual arguments: vessel identifier, vessel mereology, the static bay description and the programmable container stowage area.

Vessels

- 190. offer/accept directives to/from command centers, and
- 191. offer/accept containers to/from quay cranes.

value

```

189 vessel: vi:VI × (ccis:CCI-set × qcis:QCI-set) × ... → (StowDescr × ContDescr × Stowage × VΣ)
190   out,in { ch[{vi,cci}] | cci:CCI • cci ∈ ccis }
191         { ch[{vi,qci}] | qci:QCI • qci ∈ qcis } Unit

```

The behaviours invoked by vessel have the same signature:

- vessel_act, Item 232, pg. 477,
- V_CC_Arriv, Item 232a, pg. 477,
- vessel_response, Item 234, pg. 477,
- CC_V_Arriv, Item 235, pg. 478,
- CC_V_Dept, Item 236, pg. 478,
- fQCtV, Item 237, pg. 478, and
- fVtQC, Item 238, pg. 479.

Command Center Signature

192. The signature of command center behaviours includes the usual arguments: command center identifier, command center mereology, the static attributes (-) and the programmable state.

Vessels

193. accept directives from trucks and vessels, and
 194. offer directives to trucks, cranes, bay stacks and vessels.

```

192 command_center: cci:CCI
192   × (vis:VI-set × qcis:QCI-set × qtis:QTI-set × gcis:GCI-set × bis:BI-set × ltis:LTI-set) → CCΣ
193   → in { ch[{ui,cci}] | ui:UI, cci:CCI • ui ∈ uis ∧ cci ∈ ccis },
194   out { ch[{cci,ui}] | cci:CCI, ui:UI • cci ∈ ccis ∧ ui ∈ uis } Unit
192   assert: UI = VI | QCI | QTI | GCI | BI | LTI ∧ uis = vis ∪ qcis ∪ qtis ∪ gcis ∪ bis ∪ ltis

```

The behaviours invoked by cmd_ctr have the same signature:

- cc_ctl, Item 243, pg. 479,
- cc_mon, Item 247, pg. 480,
- LT_Arriv, Item 249, pg. 480,
- V_CC_Arriv, Item 249, pg. 480, and
- CC_V_Arriv, Item 258, pg. 480.

G.4.7 A Running System

A running system is the

195. parallel composition of all land trucks in parallel with the
 196. parallel composition of all gantry cranes of all terminal ports in parallel with the
 197. parallel composition of all bay stacks of all terminal ports in parallel with the
 198. parallel composition of all quay trucks of all terminal ports in parallel with the
 199. parallel composition of all quay cranes of all terminal ports in parallel with the
 200. parallel composition of all vessels in parallel with the
 201. parallel composition of all command centers of all terminal ports.

value

```

195 || { land_truck(uid_LT(lt), mereo_LT(lt), sta_attr_LT(lt))(progr_attr_LT(lt))
195   | lt:LT • lt ∈ lts } ||
196 || { gantry_crane(uid_GT(gc), mereo_GC(gc), sta_attr_GC(gc))(progr_attr_GC(gc))
196   | gc:GC • gc ∈ gcs } ||
197 || { bay_stack(uid_BS(bs), mereo_BS(bs), sta_attr_BS(bs))(progr_attr_BS(bs))
197   | bs:BS • bs ∈ bss } ||

```

```

198 || { quay_truck(uid_QT(qt),mereo_QT(qt),sta_attrs_QT(qt))(progr_attrs_QT(qt))
198   | qt:QT • qt ∈ qts } ||
199 || { quay_crane(uid_QC(qc),mereo_QC(qc),sta_attrs_QC(qc))(progr_attrs_QC(qc))
199   | qc:QC • qc ∈ qcs } ||
201 || { command_center(uid_CC(cc),mereo_CC(cc),sta_attrs_CC(cc))(progr_attrs_CC(cc))
201   | cc:CC • cc ∈ ccs }
200 || { vessel(uid_V(v),mereo_V(v),sta_attrs_V(v))(progr_attrs_V(v))
200   | v:V • v ∈ vs }

```

The *lts*, *gcs*, *bss*, *qts*, *qcs*, *vs*, *ccs* are defined in Sect. G.3.9 [Page 449].

G.4.8 Behaviour Definitions

Moves

202. Trucks and cranes move. We omit consideration of any specifics of “*from where to where*”.

203. Some arbitrary time interval is chosen.

204. And the behaviour “*waits*” that “*long*”!

value

202 Move: **Unit** → **Unit**

202 Move() ≡

203 **let** $\tau t : \mathbb{T}\mathbb{I}$ **in**

204 **wait**(τt) **end**

Land Truck Behaviour

We refer to Fig. G.3 [Page 457], the leftmost fifth of that figure – i.e., the leftmost six vertical arrows between the *land truck* and the *command center* behaviours.

205. Land trucks

- a. either act on their own will
- b. or, internal non-deterministically,
- c. respond to directives from the command center.

value

205 land_truck(*lti*, (*gcis*, *ccis*), *_*)(*lthold*, *ltσ*) ≡

205a land_truck_act(*lti*, (*gcis*, *ccis*), *_*)(*lthold*, *ltσ*)

205b \sqcap

205c land_truck_response(*lti*, (*gcis*, *ccis*), *_*)(*lthold*, *ltσ*)

Land Truck Actions:

Land trucks internal non-deterministically chooses

206. to move (he \dashrightarrow s model moves);

207. or informs the assumedly nearby terminal port of its imminent arrival.

value

205a land_truck_act(*lti*, (*gcis*, *ccis*), *_*)(*lthold*, *ltσ*) ≡

206 Move()

207 \sqcap LT_CC_arriv(*lti*, (*gcis*, *ccis*), *_*)(*lthold*, *ltσ*)

1 Land Truck Arrival: Cf. V_CC_arriv, Items 231a and 233 – 233d [Page 477].

208. When a land truck is close to a destination terminal port
- it ascertains the identity of that port's command center
 - informs that command center of whether
 - it holds a tagged container (by presenting its waybill),
 - or not (by presenting a "nil" message, cf. Items 249 [Page 480] and 252 [Page 480],
 - whereupon it resumes being the land truck, albeit with a state updated to reflect the arrival.
 - When still "far away" it resumes being the land truck.

value

```

208 LT_CC_arriv(lti,(gcis,ccis),_)(hold,ltσ) ≡
208   if approaching_terminal_port(ltσ,observe.POINT()15)
208a   then let cci = observe_terminal_CCI(ltσ,observe.POINT()) in
208b       msg = if hold=mkNil("nil") then hold else attr_WB(s_c(hold)) end
208b       ch[{lti,cci}] ! LL_CC_Arriv(lti,msg) ;
208c       land_truck(lti,(gcis,ccis),_)(hold,arriv_update(cci,msg,ltσ)) end
208d   else land_truck(lti,(gcis,ccis),_)(hold,ltσ) end

```

Land Truck Responses:

Land trucks external non-deterministically offers to accept advice from that terminal's command center

209. or as to transport to a gantry crane its hold;
 210. or as to transport from a gantry crane its hold.

value

```

205c land_truck_response(lti,(gcis,ccis),_)(lthold,ltσ) ≡
209   □ fLTtGC(lti,(gcis,ccis),_)(lthold,ltσ)
210   □ fGCtLT(lti,(gcis,ccis),_)(lthold,ltσ)

```

1 From Land Truck to Gantry Crane Container Transfer:

211. The command center directs a land truck to transfer a designated¹⁶ containers to a designated gantry crane.
- After some move,
 - the time-stamped update of the hold is transferred from land truck to gantry crane,
 - the land truck state is updated to reflect this fact,
 - and the land truck resumes being so with a "nil" container;
 - where the land truck may receive such directives from any terminal port.

```

110b fLTtGC(lti,(gcis,ccis),_)(mkTagC(chi,c),ltσ) ≡
211   □ { let fLTtGC(gci,ci) = ch[{lti,cci}] ? in assert: ci = uid_C(c)
211a     Move() ;
211b     ch[{lti,gci}] ! mkTagC(⟨record.TIME()gci⟩^chi,c) ;
211c     let ltσ' = lt_update_LT_to_GC(ltσ) in
211d     land_truck(lti,(gcis,ccis),_)(mkNil("nil"),ltσ')
211e     end end | cci:CCI•cci ∈ ccis } ;
110b   assert: let (τ,ui) = hd_chi in ui:LTl∧τ<record.TIME()end

```

¹⁵ POINT(): We refer to [81, Sect. 7.1.4]. s_c(lthold) is a discrete entity.

¹⁶ The command center designated container designates the container being currently carried at that moment by the land truck.

14 From Gantry Crane to Land Truck Container Transfer:

212. The command center directs land trucks to accept designated containers from designated gantry cranes.
- After some move,
 - the tagged container is transferred from gantry crane to land truck;
 - the land truck state is updated to reflect this fact;
 - and the land truck resumes being so with an updated, tagged container;
 - where the land truck may receive such directives from any terminal port.

```

110c fGCtLT(lti,(gci,ccis),_)(noc,ltσ) ≡
212   [] { let fGCtLT(gci,ci) = ch[{lti,cci}] ? in
212a     Move() ;
212b     let mkTagC(chi,c) = ch[{lti,gci}] ? in
212c     let ltσ' = lt_gc_to_lt_update(gci,ltσ) in
212d     land_truck(lti,(gci,ccis),_)(mkTagC(⟨record_TIME()⟩lti^chi,c),ltσ')
212e     end end end | cci:CCI•cci ∈ ccis }
110c   assert: if 212 accepted, then noc ≡ mkNil("nil")
212b     ∧ ci = uid_C(c)
212b     ∧ let (τ,ui) = hd chi in ui:GCI ∧ τ < record_TIME() end

```

Further Land Truck Behaviours

We shall not model the “affairs” of *land trucks* outside terminal ports or when/whether leaving these.

Gantry Crane Behaviour

213. The gantry crane
- chooses external non-deterministically to transfer a container
 - either from a land truck to the gantry crane,
 - or from the gantry crane to a quay truck,
 - or from a quay truck to the gantry crane,
 - or from the gantry crane to a land truck.

```

value
213 gantry_crane(gci,(bi,qtis,cci,ltis),_) ≡
213b   gantry_crane_lt_gc(gci,(bi,qtis,cci,ltis),_)
213a   []
213c   gantry_crane_gc_qt(gci,(bi,qtis,cci,ltis),_)
213a   []
213d   gantry_crane_qt_gc(gci,(bi,qtis,cci,ltis),_)
213a   []
213e   gantry_crane_gc_lt(gci,(bi,qtis,cci,ltis),_)

```

2 Gantry Crane: Land Truck to Bay Stack

214. The gantry crane, when transferring a container from a land truck to a bay stack,
- offers to receive a directive, of the form $\text{mkfQTvGCtBS}(\langle\langle\text{qti,gci,(bsi,ri,st,p)}\rangle\rangle)$, from the command center;
 - once received the gantry crane awaits the container from a land truck;
 - once received it moves from the gantry crane position over the quay truck to the stack of the container;

- d. and then transfers that container to the bay stack;
- e. whereupon the gantry crane resumes being that.

value

```

214 gantry_crane_lt_gc(gci,(bi,qtis,cci,ltis),_) ≡
214a   let fLTvGCtBS(lti,ci) = ch[{cci,gci}] ? in
214b   let mkTagC(chi,c) = ch[{lti,gci}] ? in assert: ci = uid_C(c)
214c   Move() ;
214d   ch[{gci,bi}] ! mkTagC((record_TIME)(cci,bi))^chi,c) ;
214e   gantry_crane(gci,(bi,qtis,cci,ltis),_) end end
214b   assert: let (τ,ui) = hd chi in ui:LTl∧τ<record_TIME() end

```

5 Gantry Crane: Bay Stack to Quay Truck

215. The gantry crane, when transferring a container from a bay stack to a quay truck,
- a. offers to receive a directive, of the form fBSvGCtQT(gci,qti), from, the command center;
 - b. once received the gantry crane
 - c. awaits the container from quay truck;
 - d. once received it transfers the appropriately tagged container to the quay truck;
 - e. and resumes being a gantry crane.

```

215 gantry_crane_bs_qt(gci,(bi,qtis,cci,ltis),_) ≡
215a   let fBSvGCtQT(qti,ci) = ch[{gci,cci}] ? in
215b   let mkTagC(chi,c) = ch[{gci,bi}] ? in
215c   Move() ;
215d   ch[{bi,qti}] ! mkTagC((record_TIME)(cci,qti))^chi,c) ;
215e   gantry_crane(gci,(bi,qtis,cci,ltis),_) end end
215b   assert: ci = uid_C(c)
215b   ∧ let (τ,ui) = hd chi in ui:LTl∧τ<record_TIME() end

```

12 Gantry Crane: Quay Truck to Bay Stack

216. The gantry crane, when transferring a container from a quay truck to a bay stack,
- a. offers to receive a directive, of the form fQTvGCtBS(gci,qti), from, the command center;
 - b. once received the gantry crane the gantry crane
 - c. awaits the container from the quay truck;
 - d. once received it transfers the appropriately tagged container to the bay stack;
 - e. and resumes being a gantry crane.

```

216 gantry_crane_qt_bs(gci,(bi,qtis,cci,ltis),_) ≡
216a   let fQTvGCtBS(qti,bsi,ci) = ch[{qti,cci}] ? in
216b   Move() ;
216c   let mkTagC(chi,c) = ch[{qti,gci}] ? in assert: ci = uid_C(c)
216d   ch[{gci,bsi}] ! mkTagC((record_TIME)(cci,bsi))^chi,c) ;
216e   gantry_crane(gci,(bi,qtis,cci,ltis),_) end end

```


13 Gantry Crane: Bay Stack to Land Truck

217. The gantry crane, when transferring a container from a bay stack to a land truck,
- offers to receive a directive of the form `mkfBSvGCtLT(gci,lti)` from, the command center;
 - once received the gantry crane awaits the container from a bay stack;
 - once received it moves the gantry crane to and the position over the stack of the container from/to the land truck;
 - and then transfers that container to the designated land truck;
 - whereupon the gantry crane resumes being that.

value

```

217 gantry_crane_bs_lt(gci,(bsi,qtis,cci,ltis),_) ≡
217a   let fBSvGCtLT(lti,ci) = ch[{cci,gci}] ? in
217b   let mkTagC(chi,c) = ch[{bsi,gci}] ? in assert: ci = uid_C(c)
217c   Move() ;
217d   ch[{gci,lti}] ! mkTagC(⟨record_TIME()⟩lti)^chi,c) end end
217e   gantry_crane(gci,(bi,qtis,cci,ltis),_)

```

Bay Stack Behaviour

218. Bay stacks
- external non-deterministically transfer containers
 - either from gantry crane to designated stack tops,
 - or from designated stack tops to gantry cranes.

```

218 bay_stack(bsi,(gci,cci),bd)(bay) ≡
218a   bay_stack_gc_bs(bsi,(gci,cci),bd)(bay)
218b   □
218c   bay_stack_bs_gc(bsi,(gci,cci),bs)(bay)

```

4 Bay Stack Load: From Gantry Crane to Bay Stack

219. The gantry crane to bay stack stack top
- offers to receive a directive of the form `mkfGCtBS(bi,ri,si,pos)` from, the command center;
 - once received the bay stack awaits the container from the gantry crane;
 - spends some time moving from the base position of the gantry crane to the bay stack position
 - where it loads the time stamped container on to the top of the designated row stack – thereby updating the bay stack;
 - whereupon it resumes being (an updated) bay stack —
 - where a check is made that the designated stack top is indeed that of the designated stack, and
 - where the container history correctly references that stack top.

```

219 bay_stack_gc_bs(bsi,(gci,cci),bd)(bay) ≡
219a   let fGCtBS((bi,ri,si,pos),ci) = ch[{gci,cci}] ? in assert: bsi ≡ bi
219b   let mkTagC(chi,c) = ch[{gci,bsi}] ? in assert: ci = uid_C(c)
219c   Move() ;
219f   let pos' = len(bay(ri))(si) assert: pos = pos' + 1,
219g   chi' = ⟨record_TIME()⟩(cci,(bi,ri,si,pos))^chi in
219d   let (pos'',[bi→bay']) = loadTagC(mkTagC(chi',c),[bi→bay]) in assert: pos' = pos''
219e   bay_stack(bsi,(gci,cci),bd)(bay') end end end end

```

5 Bay Stack Unload: From Bay Stack to Gantry Crane

220. The bay stack to gantry crane bay stack
- offers to receive a directive of the form $\text{mkfBStGC}(bi,ri,si,pos)$ from, the command center;
 - the gantry crane then spends some time to move to the designated row stack;
 - from whose top it unloads a container whose unique identifier must be that provided;
 - whereupon that container, suitably time-stamped, is transferred to the gantry crane;
 - and the bay stack resumes being that in the updated row stack state.

```

220 bay_stack_bs_gc(bsi,(gci,cci),bd)(bay) ≡
220a   let fBStGC((bi,ri,si,pos),ci) = ch[{gci,cci}] ? in
220b   Move() ;
220c   let ((chi,c),[bi→bay']) = unloadTagC((bi,ri,si,pos),[bi→bay]) in assert: ci = uid_C(c)
220d   ch[{bsi,gci}] ! mkTagC((record_TIME()(cci,gci))^chi,c) ;
220e   bay_stack(bsi,(gci,cci),bd)(bay') end end

```

Quay Truck Behaviour

221. Quay trucks either
- transfer a container from a **gantry crane** via a **quay truck** to a **quay crane**;
 - or, externally non-deterministically
 - transfer a container from a **quay crane** via a **quay truck** to a **gantry crane**;

value

```

221 quay_truck(qti,(gcis,qcis,cci),_) ≡
221a   quay_truck_c_to_qc(qti,(gcis,qcis,cci),_)
221b   []
221c   quay_truck_qc_to_gc(qti,(gcis,qcis,cci),_)

```

6 Quay Truck: Gantry Crane to Quay Crane Transfer

222. The quay truck offers
- the external non-deterministic receipt of a directive from a command center on transferring a container from a **gantry crane** via a **quay truck** to a **quay crane**;
 - the receipt of a **container** from the **gantry crane** designated in the directive to the correspondingly designated **quay crane**;
 - a time, $\tau_{l_{qt'}}$, to move the **quay truck** from the **gantry crane** to the **quay crane**;
 - the transfer of that **container** with updated history; and
 - the resumption of being the **quay truck**.

value

```

222 quay_truck_gc_qc(qti,(gcis,qcis,cci),_) ≡
222a   let fGCvQTtQC(gci,qci,ci) = ch[{qti,cci}] ? in
222b   let mkTagC(chi,c) = ch[{gci,qti}] ? in assert: ci=uid_C(c)
222c   Move() ;
222d   ch[{qti,qci}] ! mkTagC((record_TIME()(cci,qci))^chi,c) end end ;
222e   quay_truck(qti,(gcis,qcis,cci),_)

```

11 Quay Truck: Quay Crane to Gantry Crane Transfer

223. The quay truck offers
- the external non-deterministic receipt of a directive from a command center on transferring a container from a **quay crane** via a **quay truck** to a **gantry crane**;
 - the receipt of a **container** from the **quay crane** designated in the directive to the correspondingly designated **gantry crane**;
 - a time, $\tau_{l_{qt}}$, to move the **quay truck** from the **quay crane** to the **gantry crane**;
 - the transfer of that **container** with updated history; and
 - the resumption of being the **land truck**.

Items 222a – 222e mirrors Items 223a – 223e, one-by-one. Hence first five lines express:

```

value
223  quay_truck_qc_gc(qti,(gcis,qcis,cci),_) ≡
223a  let fQCvQTtGC(qci,gcis,ci) = ch[{qti,cci}] ? in
223b  let mkTagC(chi,c) = ch[{qci,qti}] ? in assert: ci = uid_C(c)
223c  Move() ;
223d  ch[{qti,gcis}] ! mkTagC((record.TIME)(cci,gcis))^chi,c) end end ;
223e  quay_truck(qti,(gcis,qcis,cci),_)

```

Quay Crane Behaviour

224. A quay crane
- alternates, external non-deterministically between
 - transferring containers, and
 - being allocated to or freed from quay positions vis-a-vis vessel bays.

```

value
224  quay_crane(qci,(qtis,cci,vis),_)(qpos) ≡
224b  quay_crane_c_xfer(qci,(qtis,cci,vis),_)(qpos)
224a  []
224c  quay_crane_position(qci,(qtis,cci,vis),_)(qpos)

```

Quay Crane: Container Transfers

225. The quay crane
- chooses external non-deterministically to transfer a container
 - either from a quay truck to a vessel or
 - or from a vessel to a quay truck.

```

value
225  quay_crane_c_xfer(qci,(qtis,cci,vis),_)(qpos) ≡
225b  quay_crane_qt_v(qci,(qtis,cci,vis),_)(qpos)
225a  []
225c  quay_crane_v_qt(qci,(qtis,cci,vis),_)(qpos)

```

7 Quay Crane: From Quay Truck to Vessel

226. The quay crane as `quay_crane_qt_v` offers
- the external non-deterministic receipt of a directive from a command center on transferring a container from a **quay truck** via a **quay crane** to a **vessel**;
 - the receipt of a **container** from the **quay truck** at the position designated in the directive to the **quay crane**;
 - a time, $\tau_{1_{qc}}$, to move the **quay truck** from whatever position it was in to the **quay crane**;
 - the transfer of that **container** with updated history; and
 - the resumption of being the **quay crane**.

value

```

226 quay_crane_qt_v(qci,(qtis,cci,vis),_)(qpos) ≡
226a   let fQTvQCtV(qti,vi,ci) = ch[{qci,cci}] ? in
226b   let mkTagC(chi,c) = ch[{qti,qci}] ? in assert: ci = uid_C(c)
226c   Move() ;
226d   ch[{qci,vi}] ! mkTagC((record_TIME()vi)^chi,c) ;
226e   quay_crane(qti,(gcis,qcis,cci),_)(qpos) end end

```

10 Quay Crane: From Vessel to Quay Truck: Items 227a – 227e mirrors Items 226a – 226e, one-by-one.

227. The quay crane as `quay_crane_v_qt` offers
- the external non-deterministic receipt of a directive from a command center on transferring a container from a **vessel** via the **quay crane** to a **quay truck**;
 - the receipt of a **container** from the **vessel** designated in the directive to the correspondingly designated **quay crane**;
 - a time, $\tau_{1_{qc}}$, to move the **quay crane** from whatever position it was in, via the **quay truck** position to a **quay truck**;
 - the transfer of that **container** with updated history; and
 - the resumption of being the **quay crane**.

value

```

227 quay_crane_v_qt(qci,(qtis,cci,vis),_)(qpos) ≡
227a   let fVvQCtQT(vi,qti,ci) = ch[{qti,cci}] ? in
227b   let mkTagC(chi,c) = ch[{vi,qci}] ? in assert: ci = uid_C(c)
227c   Move() ;
227d   ch[{qci,qti}] ! mkTagC((record_TIME()qti)^chi,c) end end ;
227e   quay_crane(qci,(qtis,cci,vis),_)(qpos)

```

Quay Crane Allocation and Freeing

228. Quay cranes are allocated to or freed from quay positions vis-a-vis vessel bays.
- -
 -

```

228 quay_crane_position(qci,(qtis,cci,vis),_)(qpos) ≡
228a   quay_crane_alloc(qci,(qtis,cci,vis),_)(qpos)
228b   □
228c   quay_crane_free(qci,(qtis,cci,vis),_)(qpos)

```

B Allocation of Quay Crane Position

229. a.
b.
c.

value

```
229 quay_crane_alloc(qci,(qtis,cci,vis),_)(qpos)
229a   let Alloc_Q_Pos(qti,vi,bi) = ch[{qci,cci}] ? in
229b   quay_crane(qci,(qtis,cci,vis),_)(mkVeBay(vi,bi)) end
229   pre: qpos = mkNilPos("nil")
```

C Freeing of Quay Crane Position

230. a.
b.
c.

value

```
230 quay_crane_free(qci,(qtis,cci,vis),_)(qpos)
230a   let Rel_Q_Pos(qti,vi,bi) = ch[{qci,cci}] ? in
230b   quay_crane(qci,(qtis,cci,vis),_)(mkNilPos("nil")) end
230   pre: qpos ≠ mkNilPos("nil")
```

Vessel Behaviour

231. Vessels
- alternate internal non-deterministically between
 - acting on their own behalf
 - and responding to directives (from a terminal port's) command center.

value

```
231 vessel(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
231b   vessel_act(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)
231a   □
231c   vessel_response(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)
```

Vessel Actions**B Vessel Arrival - From Vessel to Terminal Port**

232. Vessels
- announce a terminal ports' command center of their arrival.

value

```
232 vessel_act(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
232a   V_CC_Arriv(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)
```

233. When a vessel is close to a destination terminal port
- it ascertains the identity of that port's command center,
 - informs that command center of the layout and contents (way bills) of its hold (container stowage area),

- c. whereupon it resumes being the vessel, albeit with a state updated to reflect the arrival.
- d. When still “far away” it resumes being the vessel.

value

```

232a V_CC_Arriv(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
233   if approaching_terminal_port(vσ,observe_POINT())
233a   then let cci = observe_terminal_CCI(vσ,observe_POINT()) in
233b       ch[{vi,cci}] ! V_CC_Arriv(vi,sto_dscr,cnt_dscr) ;
233c       vessel(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,arriv_update(cci,vσ)) end
233d   else vessel(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) end

```

Vessel Responses

234. Vessels alternate external non-deterministically between being informed by a terminal ports’ command center
- a. as to where to park the vessel;
 - b. as to their departure from a terminal port;
 - c. of quay crane to vessel transfer of a container; and
 - d. of vessel to quay crane transfer of a container.

value

```

234 vessel_response(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
234a   CC_V_Arriv(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)
234b   □ CC_V_Dept(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)
234c   □ fQCtV(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)
234d   □ fVtQC(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ)

```

C Vessel Arrival - From Terminal Port to Vessel:

235. The vessel,
- a. external non-deterministically, over all command centers, offers to accept a command center to vessel directive as to it terminal port quay crane positions.
 - b. It updates its state with hat information
 - c. and resumes being a vessel – supposedly berthing now at this quay position.

value

```

235 CC_V_Arriv(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
235a   □ { let mkCC_V_Arriv(qcil) = ch[{vi,cci}] ? in
235b       let vσ' = update_state_GCIs(qcil)(vσ) in
235c       vessel(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ')
235       end end | cci:CCI • cci ∈ ccis }

```

D Vessel: Departure - From Terminal Port to Vessel:

236. The vessel,
- a. external non-deterministically over all command centers, offers to accept a command center to vessel directive as to its departure – with that directive conveying updated stowage and contents descriptions
 - b. It updates its state with hat information
 - c. and resumes being a vessel in that new state – supposedly leaving the terminal port.

```

value
236 CC_V_Dept(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
236a   □ { let mkCC_V_Dept(sto_dscr',cnt_dscr') = ch[{cci,vi}] ? in
236b     let vσ' = update_state_Stow_Cont(sto_dscr',cnt_dscr')(vσ) in
236c     vessel(vi,(ccis,qcis),_)(sto_dscr',cnt_dscr',csa,vσ')
236     end end | cci:CCI • cci ∈ ccis }

```

8 Vessel: Container Transfer - From Quay Crane to Vessel

237. The vessel,
- external non-deterministically over all command centers, offers to accept a directive from the command center as to the loading of a container received from a quay crane onto a designated stack of the vessel.
 - The vessel offers to receive that (or a) container from the designated quay crane, and
 - updates its history.
 - Then loads that container onto the vessel;
 - updates its state to reflect the fact that a container has been loaded;
 - and resumes being the vessel in the updated load (i.e., csa') and state.

```

value
237 fQCtV(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
237a   □ { let fQCtV((bi,ri,si,pos),qci,ci) = ch[{vi,cci}] ? in
237b     let mkTagC(chi,c) = ch[{qci,bi}] ? in assert: ci = uid_C(c)
237c     let tagc = mkTagC(record_TIME((bi,ri,si,pos),ci)^chi,c) in
237d     let ((chi,c),csa') = loadTagC((bi,ri,si,pos),tagc,csa) in
237e     let vσ' = update_state_CSA(csa')(vσ) in
237f     vessel(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa',vσ')
237     end end end end end | cci:CCI • cci ∈ ccis }

```

9 Vessel: Container Transfer - From Vessel to Quay Crane

238. The vessel
- offers to accept a directive from the command center as to the transfer of a container, from a designated position at the vessel to a quay crane.
 - The vessel offers (i.e., unloads) the designated container
 - and transfer the suitably time stamped container to the quay crane.
 - It then updates the vessel state as to the fact that an unload and transfer has taken place.
 - Whereupon it resumes being that vessel less the unloaded container.

```

value
238 fVtQC(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa,vσ) ≡
238a   □ { let fVtQC((bi,ri,si,pos),qci,ci) = ch[{cci,vi}] ? in
238b     let ((chi,c),csa') = unloadTagC((bi,ri,si,pos),ci,csa) in assert: ci = uid_C(c)
238c     ch[{vi,qci}] ! mkTagC(record_TIME((vi,qci))^chi,c) ;
238d     let vσ' = update_state_CSA(csa')(vσ)
238e     vessel(vi,(ccis,qcis),_)(sto_dscr,cnt_dscr,csa',vσ')
237     end end end | cci:CCI • cci ∈ ccis }

```

Further Vessel Behaviours

We shall not model the “affairs” of *vessels* while cruising the oceans.

Command Center Behaviour

239. The command center
 240. internally non-deterministically alternates between
 241. monitoring land trucks and vessels
 242. and controlling trucks, terminal port bay stacks, cranes and vessels.

```

239 cmd_ctr(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ) ≡
241   cc_mon(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ)
240   □
242   cc_ctl(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ)

```

1–14, C–D The Command Center Controller

243. The command center controls land trucks and vessels by
 244. first calculating a set of next responses;
 245. then, for each of these to communicate respective directives to designated trucks, cranes¹⁷ and bays;
 246. whereupon it resumes being the command center.

```

243 cc_ctl(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ) ≡
244   let (ccσ',nxt_dirs) = next_move(ccσ) in
245   || { ch[{cci,ui}] ! dir | (ui,dir):(U1×Directive)•(ui,dir)∈nxt_dirs }
246   || cmd_ctr(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ') end

```

Placing, formula line 246, `cmd_ctr` in parallel (`||`) with the communications of line 245 secures that if any of these communications are “hung up”, the overall behaviour will continue.

If a calculated directive informs a vessel of n quay positions then the set of directives calculated also contains exactly n quay crane directives as to their allocation to vessel and bay positions. If a calculated directive informs a vessel of departure then the set of directives calculated also contains exactly n quay crane directives as to their release from vessel and bay positions. We omit formalising this constraint.

The Command Center Monitor:

247. The command center monitors land trucks and vessels
 248. by externally non-deterministically offering to accept messages from
 249. arriving land trucks,
 250. arriving vessels, or
 251. departing vessels.

```

247 cc_mon(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ) ≡
249   LT_Arriv(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ)
248   □
250   V_Arriv(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ)
248   □
251   V_Dept(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ)

```

¹⁷ The text: ‘trucks, cranes’ abbreviates the fuller text ‘land and quay trucks and gantry and quay cranes’.

A Command Center Handling of Land Truck Arrival:

252. The command center offers to accept an `LT_CC_Arriv(lti,waybill)` message from any incoming truck (`lti`) – cf. Item 208b [Page 469].
253. Then updates its current state – now aware of the presence of the land truck and its possible hold (cf. Item 208b [Page 469]).
254. Whereupon it (i.e., the land truck) resumes being so in that updated state.

```

249 LT_Arriv(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ) ≡
252   [] { let LT_CC_Arriv(lti,nil_or_waybill) = ch[{cci,lti}] ? in
253       let ccσ' = update_CCΣ(LT_CC_Arriv(lti,nil_or_waybill))(ccσ) in
254       cmd_ctr(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ') end end | lti:LT|lti ∈ ltis }

```

B Command Center Handling of Vessel Arrival:

255. The command center offers to accept an `V_CC_Arriv(lti,all_waybills)` message from any incoming vessel (`vi`) – cf. Item 208b [Page 469].
256. Then updates its current state – now aware of the presence of the vessel and its hold.
257. Whereupon the command center resumes being so – in that updated state.

```

249 V_CC_Arriv(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ) ≡
255   [] { let V_CC_Arriv(vi,all_waybills) = ch[{cci,vi}] ? in
256       let ccσ' = update_CCΣ(V_CC_Arriv(lti,all_waybills))(ccσ) in
257       cmd_ctr(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ') end end | vi:V|vi ∈ vis }

```

258. The command center responds to a vessel arrival
259. by internal. non-deterministically calculating, for some incoming vessel, based on the current command center state, a relevant vessel's identifier, a list of quay cranes at which to park, and an update command center state;
260. communicates this list to the [calculated] vessel;
261. whereupon the command center resumes being so – in that updated state.

```

258 CC_V_Arriv(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ) ≡
259   let ((vi,qcl),ccσ') = calculate_Quay_Cranes(ccσ) in
260   ch[{cci,vi}] ! CC_V_Arrival(qcl) ;
261   cmd_ctr(cci,(vis,qcis,qtis,bsis,ltis),_)(ccσ') end

```

G.5 Conclusion**G.5.1 Variations of Container Terminal Descriptions**

The analysis & description of this report reflects one view of container terminal ports. Other views are equally, or perhaps even more interesting and relevant. For all such the analysis & description method is, and hence analysis & description steps are suggested to be the same:

- **Endurants**
 - ⊗ Structures, Parts and Materials
 - ⊗ Unique Identifiers
 - ⊗ Mereology
 - ⊗ Attributes
 - ⊗ Global Values and States
- **Perdurants**
 - ⊗ Actions and Events
 - ⊗ Channels
 - ⊗ Behaviour Signatures
 - ⊗ Behaviours
 - ⊗ System

Here are some variations to the behaviours of the terminal port behaviours:

- (i) land trucks may fetch or deliver containers from respectively to gantry cranes which then have fetched these from, respectively delivers these to quay trucks; or
- (ii) land trucks may fetch or deliver containers directly from, respectively to quay cranes.

We suggest that serious readers of this report try their hand at reformulating the behaviours of

- (i) land trucks and gantry cranes, respectively
- (ii) land trucks and quay cranes

corresponding to the two variations (i–ii).

G.5.2 A Proper Container Terminal Analysis & Description Project

We suggest the following possibility:

- 262. that a container shipping and/or a container terminal (owning & operating) company
 - a. decide to carry out a project
 - b. aimed at producing an appropriate, industrial scale analysis & description
 - c. of a realistic class of container terminal ports.
- 263.
- 264.

MORE TO COME

RSL

H

An RSL Primer

This is an ultra-short introduction to the RAISE Specification Language, RSL.

H.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

H.1.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of “that” type).

Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully “taken apart”.

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

Basic Types:

type

- [1] **Bool**
- [2] **Int**
- [3] **Nat**
- [4] **Real**
- [5] **Char**
- [6] **Text**

Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully “taken apart”.

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

Composite Type Expressions:

- [7] **A-set**
- [8] **A-infset**
- [9] $A \times B \times \dots \times C$
- [10] A^*
- [11] A^ω
- [12] $A \xrightarrow{\dot{m}} B$
- [13] $A \rightarrow B$
- [14] $A \xrightarrow{\sim} B$
- [15] (A)
- [16] $A \mid B \mid \dots \mid C$
- [17] $\text{mk_id}(\text{sel_a}:A, \dots, \text{sel_b}:B)$
- [18] $\text{sel_a}:A \dots \text{sel_b}:B$

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers $\dots, -2, -1, 0, 1, 2, \dots$.
3. The natural number type of positive integer values $0, 1, 2, \dots$.
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period (“.”), followed by a natural number (the fraction).
5. The character type of character values “a”, “bb”, ...
6. The text type of character string values “aa”, “aaa”, ..., “abc”, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
 - either a Cartesian $B \times C \times \dots \times D$, in which case it is identical to type expression kind 9,
 - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., $(A \xrightarrow{\dot{m}} B)$, or $(A^*)\text{-set}$, or $(A\text{-set})\text{list}$, or $(A \mid B) \xrightarrow{\dot{m}} (C \mid D \mid (E \xrightarrow{\dot{m}} F))$, etc.
16. The postulated disjoint union of types A, B, \dots , and C .
17. The record type of mk_id -named record values $\text{mk_id}(av, \dots, bv)$, where av, \dots, bv , are values of respective types. The distinct identifiers sel_a , etc., designate selector functions.
18. The record type of unnamed record values (av, \dots, bv) , where av, \dots, bv , are values of respective types. The distinct identifiers sel_a , etc., designate selector functions.

H.1.2 Type Definitions**Concrete Types**

Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition:

type

$A = \text{Type_expr}$

Some schematic type definitions are:

Variety of Type Definitions:

- [1] Type_name = Type_expr /* without |s or subtypes */
- [2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
- [3] Type_name ==
 mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
 ... |
 mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
- [4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
- [5] Type_name = { | v:Type_name' • $\mathcal{P}(v)$ | }

where a form of [2–3] is provided by combining the types:

Record Types:

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

axiom

```

∀ a1:A_1, a2:A_2, ..., ai:Ai •
  s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
  ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
∀ a:A • let mk_id_1(a1',a2',...,ai') = a in
  a1' = s_a1(a) ∧ a2' = s_a2(a) ∧ ... ∧ ai' = s_ai(a) end
```

Note: Values of type A, where that type is defined by $A::B \times C \times D$, can be expressed $A(b,c,d)$ for $b:B$, $c:D$, $d:D$.

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate \mathcal{P} , constitute the subtype A:

Subtypes:**type**

```
A = { | b:B •  $\mathcal{P}(b)$  | }
```

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

Sorts:**type**

```
A, B, ..., C
```

H.2 The RSL Predicate Calculus

H.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, \dots, c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions:

false, true

$a, b, \dots, c \sim a, a \wedge b, a \vee b, a \Rightarrow b, a = b, a \neq b$

are propositional expressions having Boolean values. $\sim, \wedge, \vee, \Rightarrow, =$ and \neq are Boolean connectives (i.e., operators). They can be read as: *not, and, or, if then* (or *implies*), *equal* and *not equal*.

H.2.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, \dots, c designate Boolean values, let x, y, \dots, z (or term expressions) designate non-Boolean values and let i, j, \dots, k designate number values, then:

Simple Predicate Expressions:

false, true

a, b, \dots, c

$\sim a, a \wedge b, a \vee b, a \Rightarrow b, a = b, a \neq b$

$x = y, x \neq y,$

$i < j, i \leq j, i \geq j, i \neq j, i \geq j, i > j$

are simple predicate expressions.

H.2.3 Quantified Expressions

Let X, Y, \dots, Z be type names or type expressions, and let $\mathcal{P}(x), \mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

Quantified Expressions:

$\forall x:X \cdot \mathcal{P}(x)$

$\exists y:Y \cdot \mathcal{Q}(y)$

$\exists ! z:Z \cdot \mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are “read” as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

H.3 Concrete RSL Types: Values and Operations

H.3.1 Arithmetic

Arithmetic:

type

Nat, Int, Real

value

$+, -, *: \mathbf{Nat} \times \mathbf{Nat} \rightarrow \mathbf{Nat} \mid \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int} \mid \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$

$/: \mathbf{Nat} \times \mathbf{Nat} \rightarrow \mathbf{Nat} \mid \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int} \mid \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$

$<, \leq, =, \neq, \geq, > (\mathbf{Nat} \mid \mathbf{Int} \mid \mathbf{Real}) \rightarrow (\mathbf{Nat} \mid \mathbf{Int} \mid \mathbf{Real})$

H.3.2 Set Expressions

Set Enumerations

Let the below a 's denote values of type A , then the below designate simple set enumerations:

Set Enumerations:

$$\{\{\}, \{a\}, \{e_1, e_2, \dots, e_n\}, \dots\} \in \mathbf{A\text{-set}}$$

$$\{\{\}, \{a\}, \{e_1, e_2, \dots, e_n\}, \dots, \{e_1, e_2, \dots\}\} \in \mathbf{A\text{-infset}}$$

Set Comprehension

The expression, last line below, to the right of the \equiv , expresses set comprehension. The expression “builds” the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension:

type

A, B

$P = A \rightarrow \mathbf{Bool}$

$Q = A \rightsquigarrow B$

value

comprehend: $\mathbf{A\text{-infset}} \times P \times Q \rightarrow \mathbf{B\text{-infset}}$

comprehend(s, P, Q) $\equiv \{ Q(a) \mid a:A \cdot a \in s \wedge P(a) \}$

H.3.3 Cartesian Expressions

Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, \dots, C , then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations:

type

A, B, \dots, C

$A \times B \times \dots \times C$

value

(e_1, e_2, \dots, e_n)

H.3.4 List Expressions

List Enumerations

Let a range over values of type A , then the below expressions are simple list enumerations:

List Enumerations:

$$\begin{aligned} & \{ \langle \rangle, \langle e \rangle, \dots, \langle e_1, e_2, \dots, e_n \rangle, \dots \} \in A^* \\ & \{ \langle \rangle, \langle e \rangle, \dots, \langle e_1, e_2, \dots, e_n \rangle, \dots, \langle e_1, e_2, \dots, e_n, \dots \rangle, \dots \} \in A^\omega \\ & \langle a_{-i} \dots a_{-j} \rangle \end{aligned}$$

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j . If the latter is smaller than the former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

List Comprehension:**type**

$$A, B, P = A \rightarrow \mathbf{Bool}, Q = A \xrightarrow{\sim} B$$

value

$$\begin{aligned} & \text{comprehend: } A^\omega \times P \times Q \xrightarrow{\sim} B^\omega \\ & \text{comprehend}(l, P, Q) \equiv \\ & \quad \langle Q(l(i)) \mid i \text{ in } \langle 1..len\ l \rangle \cdot P(l(i)) \rangle \end{aligned}$$

H.3.5 Map Expressions**Map Enumerations**

Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

Map Enumerations:**type**

$$\begin{aligned} & T1, T2 \\ & M = T1 \xrightarrow{\sim} T2 \end{aligned}$$

value

$$\begin{aligned} & u, u1, u2, \dots, un: T1, v, v1, v2, \dots, vn: T2 \\ & [], [u \mapsto v], \dots, [u1 \mapsto v1, u2 \mapsto v2, \dots, un \mapsto vn] \quad \forall \in M \end{aligned}$$

Map Comprehension

The last line below expresses map comprehension:

Map Comprehension:**type**

$$\begin{aligned} & U, V, X, Y \\ & M = U \xrightarrow{\sim} V \\ & F = U \xrightarrow{\sim} X \\ & G = V \xrightarrow{\sim} Y \\ & P = U \rightarrow \mathbf{Bool} \end{aligned}$$

value

$$\begin{aligned} & \text{comprehend: } M \times F \times G \times P \rightarrow (X \xrightarrow{\sim} Y) \\ & \text{comprehend}(m, F, G, P) \equiv \\ & \quad [F(u) \mapsto G(m(u)) \mid u: U \cdot u \in \mathbf{dom}\ m \wedge P(u)] \end{aligned}$$

H.3.6 Set Operations

Set Operator Signatures

Set Operations:

value

- 19 $\in: A \times A\text{-infset} \rightarrow \mathbf{Bool}$
- 20 $\notin: A \times A\text{-infset} \rightarrow \mathbf{Bool}$
- 21 $\cup: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}$
- 22 $\cup: (A\text{-infset})\text{-infset} \rightarrow A\text{-infset}$
- 23 $\cap: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}$
- 24 $\cap: (A\text{-infset})\text{-infset} \rightarrow A\text{-infset}$
- 25 $\setminus: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset}$
- 26 $\subset: A\text{-infset} \times A\text{-infset} \rightarrow \mathbf{Bool}$
- 27 $\subseteq: A\text{-infset} \times A\text{-infset} \rightarrow \mathbf{Bool}$
- 28 $=: A\text{-infset} \times A\text{-infset} \rightarrow \mathbf{Bool}$
- 29 $\neq: A\text{-infset} \times A\text{-infset} \rightarrow \mathbf{Bool}$
- 30 $\mathbf{card}: A\text{-infset} \xrightarrow{\sim} \mathbf{Nat}$

Set Examples

Set Examples:

examples

- $a \in \{a,b,c\}$
- $a \notin \{\}, a \notin \{b,c\}$
- $\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
- $\cup\{\{a\},\{a,bb\},\{a,d\}\} = \{a,b,d\}$
- $\{a,b,c\} \cap \{c,d,e\} = \{c\}$
- $\cap\{\{a\},\{a,bb\},\{a,d\}\} = \{a\}$
- $\{a,b,c\} \setminus \{c,d\} = \{a,bb\}$
- $\{a,bb\} \subset \{a,b,c\}$
- $\{a,b,c\} \subseteq \{a,b,c\}$
- $\{a,b,c\} = \{a,b,c\}$
- $\{a,b,c\} \neq \{a,bb\}$
- $\mathbf{card} \{\} = 0, \mathbf{card} \{a,b,c\} = 3$

Informal Explication

- 19. \in : The membership operator expresses that an element is a member of a set.
- 20. \notin : The nonmembership operator expresses that an element is not a member of a set.
- 21. \cup : The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
- 22. \cup : The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
- 23. \cap : The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- 24. \cap : The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

25. \setminus : The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26. \subset : The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. \subsetneq : The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28. $=$: The equal operator expresses that the two operand sets are identical.
29. \neq : The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows (\equiv is the definition symbol):

Set Operation Definitions:

value

$$s' \cup s'' \equiv \{ a \mid a:A \cdot a \in s' \vee a \in s'' \}$$

$$s' \cap s'' \equiv \{ a \mid a:A \cdot a \in s' \wedge a \in s'' \}$$

$$s' \setminus s'' \equiv \{ a \mid a:A \cdot a \in s' \wedge a \notin s'' \}$$

$$s' \subseteq s'' \equiv \forall a:A \cdot a \in s' \Rightarrow a \in s''$$

$$s' \subsetneq s'' \equiv s' \subseteq s'' \wedge \exists a:A \cdot a \in s'' \wedge a \notin s'$$

$$s' = s'' \equiv \forall a:A \cdot a \in s' \equiv a \in s'' \equiv s' \subseteq s'' \wedge s'' \subseteq s'$$

$$s' \neq s'' \equiv s' \cap s'' \neq \{ \}$$

card $s \equiv$
if $s = \{ \}$ **then** 0 **else**
let $a:A \cdot a \in s$ **in** 1 + **card** ($s \setminus \{a\}$) **end end**
pre s /* is a finite set */
card $s \equiv$ **chaos** /* tests for infinity of s */

H.3.7 Cartesian Operations

Cartesian Operations:

type

A, B, C
 $g0: G0 = A \times B \times C$
 $g1: G1 = (A \times B \times C)$
 $g2: G2 = (A \times B) \times C$
 $g3: G3 = A \times (B \times C)$

value

$va:A, vb:B, vc:C, vd:D$

$(va,vb,vc):G0,$
 $(va,vb,vc):G1$
 $((va,vb),vc):G2$
 $(va3,(vb3,vc3)):G3$

decomposition expressions

let $(a1,b1,c1) = g0,$
 $(a1',b1',c1') = g1$ **in .. end**
let $((a2,b2),c2) = g2$ **in .. end**
let $(a3,(b3,c3)) = g3$ **in .. end**

H.3.8 List Operations

List Operator Signatures

List Operations:

value

hd: $A^\omega \rightsquigarrow A$
tl: $A^\omega \rightsquigarrow A^\omega$
len: $A^\omega \rightsquigarrow \mathbf{Nat}$
inds: $A^\omega \rightarrow \mathbf{Nat-infset}$
elems: $A^\omega \rightarrow \mathbf{A-infset}$
 $\cdot(\cdot)$: $A^\omega \times \mathbf{Nat} \rightsquigarrow A$
 $\hat{\cdot}$: $A^* \times A^\omega \rightarrow A^\omega$
 $=$: $A^\omega \times A^\omega \rightarrow \mathbf{Bool}$
 \neq : $A^\omega \times A^\omega \rightarrow \mathbf{Bool}$

List Operation Examples**List Examples:****examples**

hd $\langle a_1, a_2, \dots, a_m \rangle = a_1$
tl $\langle a_1, a_2, \dots, a_m \rangle = \langle a_2, \dots, a_m \rangle$
len $\langle a_1, a_2, \dots, a_m \rangle = m$
inds $\langle a_1, a_2, \dots, a_m \rangle = \{1, 2, \dots, m\}$
elems $\langle a_1, a_2, \dots, a_m \rangle = \{a_1, a_2, \dots, a_m\}$
 $\langle a_1, a_2, \dots, a_m \rangle(i) = a_i$
 $\langle a, b, c \rangle \hat{\cdot} \langle a, b, d \rangle = \langle a, b, c, a, b, d \rangle$
 $\langle a, b, c \rangle = \langle a, b, c \rangle$
 $\langle a, b, c \rangle \neq \langle a, b, d \rangle$

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i , gives the i th element of the list.
- $\hat{\cdot}$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- \neq : The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions**List Operator Definitions:****value**

is_finite_list: $A^\omega \rightarrow \mathbf{Bool}$

len $q \equiv$
case $\text{is_finite_list}(q)$ **of**
 true \rightarrow **if** $q = \langle \rangle$ **then** 0 **else** $1 + \text{len tl } q$ **end**,
 false \rightarrow **chaos end**

inds $q \equiv$
case $\text{is_finite_list}(q)$ **of**
 true $\rightarrow \{ i \mid i:\text{Nat} \cdot 1 \leq i \leq \text{len } q \}$,
 false $\rightarrow \{ i \mid i:\text{Nat} \cdot i \neq 0 \}$ **end**

elems $q \equiv \{ q(i) \mid i:\text{Nat} \cdot i \in \text{inds } q \}$

$q(i) \equiv$
 if $i=1$
 then
 if $q \neq \langle \rangle$
 then let $a:A, q':Q \cdot q = \langle a \rangle \wedge q'$ **in** a **end**
 else chaos end
 else $q(i-1)$ **end**

$fq \hat{=} iq \equiv$
 \langle **if** $1 \leq i \leq \text{len } fq$ **then** $fq(i)$ **else** $iq(i - \text{len } fq)$ **end**
 $\mid i:\text{Nat} \cdot$ **if** $\text{len } iq \neq \text{chaos}$ **then** $i \leq \text{len } fq + \text{len } iq$ **end \rangle
 pre $\text{is_finite_list}(fq)$**

$iq' = iq'' \equiv$
 inds $iq' = \text{inds } iq'' \wedge \forall i:\text{Nat} \cdot i \in \text{inds } iq' \Rightarrow iq'(i) = iq''(i)$

$iq' \neq iq'' \equiv \sim(iq' = iq'')$

H.3.9 Map Operations

Map Operator Signatures and Map Operation Examples

Map Operations

value

$m(a): M \rightarrow A \xrightarrow{\sim} B, m(a) = b$

dom: $M \rightarrow A\text{-infset}$ [domain of map]

dom $[a_1 \mapsto b_1, a_2 \mapsto b_2, \dots, a_n \mapsto b_n] = \{a_1, a_2, \dots, a_n\}$

rng: $M \rightarrow B\text{-infset}$ [range of map]

rng $[a_1 \mapsto b_1, a_2 \mapsto b_2, \dots, a_n \mapsto b_n] = \{b_1, b_2, \dots, b_n\}$

$\dagger: M \times M \rightarrow M$ [override extension]

$[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] \dagger [a' \mapsto bb'', a'' \mapsto bb'] = [a \mapsto b, a' \mapsto bb', a'' \mapsto bb']$

$\cup: M \times M \rightarrow M$ [merge \cup]

$[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] \cup [a''' \mapsto bb'''] = [a \mapsto b, a' \mapsto bb', a'' \mapsto bb'', a''' \mapsto bb''']$

$$\begin{aligned} \backslash: M \times \mathbf{A}\text{-infset} &\rightarrow M \text{ [restriction by]} \\ &[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] \setminus \{a\} = [a' \mapsto bb', a'' \mapsto bb''] \\ /: M \times \mathbf{A}\text{-infset} &\rightarrow M \text{ [restriction to]} \\ &[a \mapsto b, a' \mapsto bb', a'' \mapsto bb''] / \{a', a''\} = [a' \mapsto bb', a'' \mapsto bb''] \\ =, \neq: M \times M &\rightarrow \mathbf{Bool} \\ \circ: (A \xrightarrow{m} B) \times (B \xrightarrow{m} C) &\rightarrow (A \xrightarrow{m} C) \text{ [composition]} \\ &[a \mapsto b, a' \mapsto bb'] \circ [bb \mapsto c, bb' \mapsto c', bb'' \mapsto c''] = [a \mapsto c, a' \mapsto c'] \end{aligned}$$

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m .
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- \dagger : Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some “pairings” of the right operand map.
- \cup : Merge. When applied to two operand maps, it gives a merge of these maps.
- \backslash : Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- $/$: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- $=$: The equal operator expresses that the two operand maps are identical.
- \neq : The nonequal operator expresses that the two operand maps are *not* identical.
- \circ : Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1 , to the range elements of the right operand map, m_2 , such that if a is in the definition set of m_1 and maps into b , and if b is in the definition set of m_2 and maps into c , then a , in the composition, maps into c .

Map Operation Redefinitions

The map operations can also be defined as follows:

Map Operation Redefinitions:

value

$$\mathbf{rng} \ m \equiv \{ m(a) \mid a:A \cdot a \in \mathbf{dom} \ m \}$$

$$\begin{aligned} m1 \ \dagger \ m2 &\equiv \\ &[a \mapsto b \mid a:A, b:B \cdot \\ &\quad a \in \mathbf{dom} \ m1 \ \setminus \ \mathbf{dom} \ m2 \ \wedge \ bb = m1(a) \ \vee \ a \in \mathbf{dom} \ m2 \ \wedge \ bb = m2(a)] \end{aligned}$$

$$m1 \ \cup \ m2 \equiv [a \mapsto b \mid a:A, b:B \cdot \\ \quad a \in \mathbf{dom} \ m1 \ \wedge \ bb = m1(a) \ \vee \ a \in \mathbf{dom} \ m2 \ \wedge \ bb = m2(a)]$$

$$\begin{aligned} m \ \setminus \ s &\equiv [a \mapsto m(a) \mid a:A \cdot a \in \mathbf{dom} \ m \ \setminus \ s] \\ m \ / \ s &\equiv [a \mapsto m(a) \mid a:A \cdot a \in \mathbf{dom} \ m \ \cap \ s] \end{aligned}$$

$$m1 = m2 \equiv$$

$$\text{dom } m1 = \text{dom } m2 \wedge \forall a:A \cdot a \in \text{dom } m1 \Rightarrow m1(a) = m2(a)$$

$$m1 \neq m2 \equiv \sim(m1 = m2)$$

$$m \circ n \equiv$$

$$[a \mapsto c \mid a:A, c:C \cdot a \in \text{dom } m \wedge c = n(m(a))]$$

$$\text{pre rng } m \subseteq \text{dom } n$$

H.4 λ -Calculus + Functions

H.4.1 The λ -Calculus Syntax

λ -Calculus Syntax:

type /* A BNF Syntax: */

$\langle L \rangle ::= \langle V \rangle \mid \langle F \rangle \mid \langle A \rangle \mid (\langle A \rangle)$

$\langle V \rangle ::=$ /* variables, i.e. identifiers */

$\langle F \rangle ::= \lambda \langle V \rangle \cdot \langle L \rangle$

$\langle A \rangle ::= (\langle L \rangle \langle L \rangle)$

value /* Examples */

$\langle L \rangle$: e, f, a, ...

$\langle V \rangle$: x, ...

$\langle F \rangle$: $\lambda x \cdot e$, ...

$\langle A \rangle$: f a, (f a), f(a), (f)(a), ...

H.4.2 Free and Bound Variables

Free and Bound Variables: Let x, y be variable names and e, f be λ -expressions.

- $\langle V \rangle$: Variable x is free in x .
- $\langle F \rangle$: x is free in $\lambda y \cdot e$ if $x \neq y$ and x is free in e .
- $\langle A \rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

H.4.3 Substitution

In RSL, the following rules for substitution apply:

Substitution:

- $\text{subst}([N/x]x) \equiv N$;
- $\text{subst}([N/x]a) \equiv a$,
for all variables $a \neq x$;
- $\text{subst}([N/x](P Q)) \equiv (\text{subst}([N/x]P) \text{subst}([N/x]Q))$;
- $\text{subst}([N/x](\lambda x \cdot P)) \equiv \lambda y \cdot P$;
- $\text{subst}([N/x](\lambda y \cdot P)) \equiv \lambda y \cdot \text{subst}([N/x]P)$,
if $x \neq y$ and y is not free in N or x is not free in P ;
- $\text{subst}([N/x](\lambda y \cdot P)) \equiv \lambda z \cdot \text{subst}([N/z] \text{subst}([z/y]P))$,
if $y \neq x$ and y is free in N and x is free in P
(where z is not free in $(N P)$).

H.4.4 α -Renaming and β -Reduction

α and β Conversions:

- α -renaming: $\lambda x.M$
If x, y are distinct variables then replacing x by y in $\lambda x.M$ results in $\lambda y.\text{subst}([y/x]M)$. We can rename the formal parameter of a λ -function expression provided that no free variables of its body M thereby become bound.
- β -reduction: $(\lambda x.M)(N)$
All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. $(\lambda x.M)(N) \equiv \text{subst}([N/x]M)$

H.4.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures:

type

A, B, C

value

obs_B: $A \rightarrow B$,

obs_C: $A \rightarrow C$,

gen_A: $B \times C \rightarrow A$

H.4.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions:

value

f: Arguments \rightarrow Result

f(args) \equiv DValueExpr

g: Arguments $\xrightarrow{\sim}$ Result

g(args) \equiv ValueAndStateChangeClause

pre P(args)

Or functions can be defined implicitly:

Implicit Function Definitions:

value

f: Arguments \rightarrow Result

f(args) **as** result

post P1(args,result)

g: Arguments $\xrightarrow{\sim}$ Result

g(args) **as** result

pre P2(args)

post P3(args,result)

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

H.5 Other Applicative Expressions

H.5.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions:

let $a = \mathcal{E}_d$ **in** $\mathcal{E}_b(a)$ **end**

is an “expanded” form of:

$(\lambda a. \mathcal{E}_b(a))(\mathcal{E}_d)$

H.5.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive let Expressions:

let $f = \lambda a:A \cdot E(f)$ **in** $B(f,a)$ **end**

is “the same” as:

let $f = YF$ **in** $B(f,a)$ **end**

where:

$F \equiv \lambda g. \lambda a. (E(g))$ and $YF = F(YF)$

H.5.3 Predicative let Expressions

Predicative **let** expressions:

Predicative let Expressions:

let $a:A \cdot \mathcal{P}(a)$ **in** $\mathcal{B}(a)$ **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}(a)$ for evaluation in the body $\mathcal{B}(a)$.

H.5.4 Pattern and “Wild Card” let Expressions

Patterns and *wild cards* can be used:

Patterns:

```

let {a} ∪ s = set in ... end
let {a, _} ∪ s = set in ... end

let (a,b,...,c) = cart in ... end
let (a,_,...,c) = cart in ... end

let ⟨a⟩ℓ = list in ... end
let ⟨a,_,bb⟩ℓ = list in ... end

let [a↔bb] ∪ m = map in ... end
let [a↔b, _] ∪ m = map in ... end

```

H.5.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals:

```

if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /* same as: */
  if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
  choice_pattern_1 → expr_1,
  choice_pattern_2 → expr_2,
  ...
  choice_pattern_n_or_wild_card → expr_n
end

```

H.5.6 Operator/Operand Expressions**Operator/Operand Expressions:**

```

⟨Expr⟩ ::=
  ⟨Prefix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Suffix_Op⟩
  | ...
⟨Prefix_Op⟩ ::=
  - | ~ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=

```

$$= | \neq | \equiv | + | - | * | \uparrow | / | < | \leq | \geq | > | \wedge | \vee | \Rightarrow$$

$$| \in | \notin | \cup | \cap | \setminus | \subset | \subseteq | \supseteq | \supset | \hat{=} | \dagger | \circ$$

(Suffix_Op) ::= !

H.6 Imperative Constructs

H.6.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Statements and State Change:

Unit
value
 stmt: **Unit** \rightarrow **Unit**
 stmt()

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** \rightarrow **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as “only” arguments to a function “means” that () is an argument of type **Unit**.

H.6.2 Variables and Assignment

Variables and Assignment:

0. **variable** v:Type := expression
1. v := expr

H.6.3 Statement Sequences and skip

Sequencing is expressed using the ‘;’ operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and skip:

2. **skip**
3. stm_1;stm_2;...;stm_n

H.6.4 Imperative Conditionals

Imperative Conditionals:

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1 \rightarrow S_1(p_1), ..., p_n \rightarrow S_n(p_n) **end**

H.6.5 Iterative Conditionals

Iterative Conditionals:

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

H.6.6 Iterative Sequencing

Iterative Sequencing:

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

H.7 Process Constructs

H.7.1 Process Channels

Let A and B stand for two types of (channel) messages and $i:Kidx$ for channel array indexes, then:

Process Channels:

```
channel c:A
channel { k[i]:B • i:Idx }
channel { k[i,j,...,k]:B • i:Idx,j:Jdx,...,k:Kdx }
```

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

H.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

Process Composition:

```
P || Q   Parallel composition
P [] Q   Nondeterministic external choice (either/or)
P [] Q   Nondeterministic internal choice (either/or)
P ≡ Q   Interlock parallel composition
```

express the parallel (\parallel) of two processes, or the nondeterministic choice between two processes: either external (\square) or internal (\square). The interlock (\equiv) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

H.7.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events:

```
c ?, k[i] ?   Input
c ! e, k[i] ! e   Output
```

expresses the willingness of a process to engage in an event that “reads” an input, respectively “writes” an output.

H.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Process Definitions:

value

P: **Unit** → **in** c **out** k[i]

Unit

Q: i:KIdx → **out** c **in** k[i] **Unit**

P() ≡ ... c ? ... k[i] ! e ...

Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

H.8 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

Simple RSL Specifications:

type

...

variable

...

channel

...

value

...

axiom

...

Part **X**

Indexes

I

Indexes

I.1. Definitions	505
I.2. Concepts	510
I.3. Examples	516
I.4. Analysis Prompts	517
I.5. Description Prompts	517
I.6. Attribute Categories	517
I.7. RSL Symbols	517

I.1 Definitions

“being”, 10	part, 14, 106
“large”	Atomic Part, 14, 106
domain, 9	Attribute
“narrow”	active, 30
domain, 9	autonomous, 30
“small”	biddable, 30
domain, 9	dynamic, 30
action	inert, 30
derived, 197	programmable, 31
discrete, 43, 232	reactive, 30
active	static, 30
attribute, 30, 146	attribute
Active Parts, 231	active, 30, 146
Actor, 43	biddable, 30, 146
actor, 43, 232	dynamic, 30, 146
analysis	inert, 30, 146
language, 65	programmable, 31, 146
Animal, 16	reactive, 30, 146
Artifact, 14	static, 30, 146
artifact, 14	autonomous
Artifacts, 17	attribute, 30, 146
assumptions	axiom, 35
design, 170	behaviour
Atomic	continuous, 47

- discrete, 43, 232
- biddable
 - attribute, 30, 146
- Component, 16
- component, 16, 105
- Components, 105
- Composite
 - part, 15, 106
- Composite Part, 15, 106
- confusion, 36
- context of
 - the domain, 9
- continuous
 - behaviour, 47
 - endurant, 11, 105
- Continuous Domain Endurant, 105
- Continuous Endurant, 11
- definite
 - space, 39, 229
 - time, 40, 230
- Definite Space, 39, 229
- Definite Time, 40, 230
- derived, 22
 - action, 197
 - event, 198
 - perdurant, 197
 - requirements, 191, 197
- Derived Action, 197
- Derived Event, 198
- Derived Perdurant, 197
- description
 - domain, 7
 - prompt, 27, 109
 - tree, 118
 - language, 65
 - prompt
 - domain, 27, 109
 - tree
 - domain, 118
- design
 - assumptions, 170
 - requirements, 170
- Determination, 180
- determination
 - domain, 180
- development
 - software
 - trptych, 102
 - trptych
- software, 102
- discourse
 - universe of, 9
- discrete
 - action, 43, 232
 - behaviour, 43, 232
 - endurant, 11, 105
 - event, 233
- Discrete Action, 43
- Discrete Behaviour, 43
- Discrete Domain Endurant, 105
- Discrete Endurant, 11
- Domain, 6, 219
 - Engineering, 136
 - Science, 136
- domain
 - “large”, 9
 - “narrow”, 9
 - “small”, 9
 - description, 7
 - prompt, 27, 109
 - tree, 118
 - determination, 180
 - extension, 182
 - external
 - interfaces, 9
 - facet, 71
 - human behaviour, 95
 - instantiation, 176
 - interfaces
 - external, 9
 - management, 91
 - organisation, 91
 - partial
 - requirement, 191
 - prescription
 - requirements, 171
 - projection, 171
 - prompt
 - description, 27, 109
 - regulation, 79
 - requirement
 - partial, 191
 - shared, 191
 - requirements, 169
 - prescription, 171
 - rule, 79
 - script, 81
 - shared
 - requirement, 191
 - tree

- description, 118
- Domain Description, 7
- Domain Endurant, 104
- Domain Entity, 104
- Domain Instantiation, 176
- domain of
 - interest, 9
- Domain Perdurant, 105
- Domain Projection, 171
- Domain Requirements Prescription, 171
- dynamic
 - attribute, 30, 146
- Endurant, 10, 222
- endurant, 10, 104, 222
 - continuous, 11, 105
 - discrete, 11, 105
 - extension, 182
- Endurant Extension, 182
- Engineering
 - Domain, 136
- Entity, 10, 222
- entity, 10, 104, 222
- Epistemology, 220
- Event, 43
- event, 43
 - derived, 198
 - discrete, 233
- expression
 - function
 - type, 47
 - type
 - function, 47
- Extension, 182
- extension
 - domain, 182
 - endurant, 182
- external
 - domain
 - interfaces, 9
 - interfaces
 - domain, 9
 - part
 - quality, 108
 - quality
 - part, 108
- facet
 - domain, 71
- fitting
 - requirements, 190, 191
- formal
 - method, 102
 - software development, 102
 - software development
 - method, 102
- Formal Method, 102
- Formal Software Development, 102
- function
 - expression
 - type, 47
 - partial, 47
 - signature, 47
 - total, 47
 - type
 - expression, 47
- Function Signature, 47
- Function Type Expression, 47
- goal, 202
- harmonisation
 - requirements, 190
- Human, 16
- human behaviour
 - domain, 95
- Identity of Indiscernibles, 244
- Indefinite Space, 39, 228
- Indefinite Time, 40, 230
- Indiscernibility of Identicals, 244
- inert
 - attribute, 30, 146
- instantiation
 - domain, 176
- Intentional “Pull”, 226
- Intentional Pull, 34
- interest
 - domain of, 9
- interface
 - requirements, 170, 191
- interfaces
 - domain
 - external, 9
 - external
 - domain, 9
 - internal
 - system, 9
 - system
 - internal, 9
- internal
 - interfaces
 - system, 9

- part
 - quality, 108
- qualities, 28
- quality
 - part, 108
- system
 - interfaces, 9
- intrinsic, 72
- junk, 36
- language
 - analysis, 65
 - description, 65
- Living Species, I, 12
- Living Species, II, 15
- machine
 - requirements, 170
- Man-made Parts: Artifacts, 14
- management
 - domain, 91
- Material, 17, 105
- material, 17, 23, 105
- Mereology, 221
- mereology, 26
 - type, 26
- Metaphysics, 220
- Method, 102
- method, 71, 102, 158
 - formal, 102
 - software development, 102
 - software development
 - formal, 102
- Methodology, 102
- methodology, 102, 158
- Natural Part, 13
- natural part, 13
- Natural Parts, 13
- obligation
 - proof, 35
- Ontology, 221
- organisation
 - domain, 91
- Part, 105
- part, 105
 - Atomic, 14, 106
 - Composite, 15, 106
 - external
 - quality, 108
 - internal
 - quality, 108
- partial
 - domain
 - requirement, 191
 - function, 47
 - requirement
 - domain, 191
- Passive Parts, 231
- Perdurant, 10, 222
- perdurant, 10, 105, 222
 - derived, 197
- phenomenon, 10, 104, 222
- Philosophy, 220
- Physical Parts, 12
- prerequisite
 - prompt, 19, 27, 105, 107, 108
 - is_ entity, 10, 11
- prescription
 - domain
 - requirements, 171
 - requirements
 - domain, 171
- Proactive Parts, 231
- programmable
 - attribute, 31, 146
- projection
 - domain, 171
- prompt
 - description
 - domain, 27, 109
 - domain
 - description, 27, 109
 - prerequisite, 19, 27, 105, 107, 108
- proof
 - obligation, 35
- qualities
 - internal, 28
 - part, 108
- quality
 - external
 - part, 108
 - internal
 - part, 108
 - part

- external, 108
- internal, 108
- reactive
 - attribute, 30, 146
- regulation
 - domain, 79
- requirement
 - domain
 - partial, 191
 - shared, 191
 - partial
 - domain, 191
 - shared
 - domain, 191
- requirements
 - derived, 191, 197
 - design, 170
 - domain, 169
 - prescription, 171
 - fitting, 190, 191
 - harmonisation, 190
 - interface, 170, 191
 - machine, 170
 - prescription
 - domain, 171
- Requirements Fitting, 190
- Requirements Harmonisation, 190
- rule
 - domain, 79
- Science
 - Domain, 136
- script
 - domain, 81
- shared
 - domain
 - requirement, 191
 - requirement
 - domain, 191
- sharing, 191
- signature
 - function, 47
- software
 - development
 - tritych, 102
 - tritych
 - development, 102
- software development
 - formal
 - method, 102
 - method
 - formal, 102
- space
 - definite, 39, 229
- State, 18
- state, 226
- static
 - attribute, 30, 146
- Structure, 12
- structure, 12
- sub-part, 14, 106
- support
 - technology, 75
- system
 - interfaces
 - internal, 9
 - internal
 - interfaces, 9
- technology
 - support, 75
- the domain
 - context of, 9
- The Triptych Approach to Software Development, 102
- time
 - definite, 40, 230
- total
 - function, 47
- Transcendental, 36
- Transcendental Deduction, 36
- Transcendentality, 37
- tree
 - description
 - domain, 118
 - domain
 - description, 118
- tritych
 - development
 - software, 102
 - software
 - development, 102
- type
 - expression
 - function, 47
 - function
 - expression, 47
 - mereology, 26
- universe of
 - discourse, 9
- Universe of Discourse, 219
- Upper Ontology, 221
- Verification Paradigm, 170

I.2 Concepts

- [endurant]
 - analysis prompts
 - domain, 113
 - description prompts
 - domain, 113
 - domain
 - analysis prompts, 113
 - description prompts, 113
- “thing”, 10
- abstract
 - value, 24
- abstract type, 391
- abstraction, 10, 75, 104, 222
- accessibility, 200
- action, 42, 67, 88, 232
 - shared, 158, 169
- adaptive, 200
- analysed &
 - described, 9
- analysis
 - domain
 - prompt, 65
 - language, 65
 - prompt
 - domain, 65
- analysis &
 - description
 - domain, 9
 - prompts, 66
 - domain
 - description, 9
 - prompts
 - description, 66
- analysis prompts
 - [endurant]
 - domain, 113
 - domain
 - [endurant], 113
- assumptions
 - design, 170
- atomic part, 391
- attribute
 - embedded
 - sharing, 189, 196
 - external, 183, 185, 189
 - shared, 103
 - sharing
 - embedded, 189, 196
 - update, 103
- availability, 200
- axiom, 7
- behaviour, 7, 42, 67, 232
 - shared, 158, 169
- change
 - state, 47
- common
 - projection, 190
- communication, 57
- composite part, 391
- composite, 159
- computable
 - objects, 65
- computer
 - program, 157
- computer &
 - computing
 - science, 65, 66
 - science
 - computing, 65, 66
- computing
 - computer &
 - science, 65, 66
 - science
 - computer &, 65, 66
- conceive, 10, 104, 222
- concrete type, 391
- concurrency, 57
- conservative
 - extension, 182
 - proof theoretic, 75
 - proof theoretic
 - extension, 75
- Constraint, 202
- constructor
 - function
 - type, 47
 - type
 - function, 47
- continuous
 - time, 47
- control, 81
- corrective, 200
- deduction

- transcendental, 3, 67
- demonstration, 200
- dependability, 200
 - requirements, 200
- derivation
 - part, 143
- derived
 - requirements, 158, 169
- described
 - analysed &, 9
- description
 - analysis &
 - domain, 9
 - prompts, 66
 - domain, 157
 - analysis &, 9
 - facet, 201
 - prompt, 27, 65, 66, 109
 - tree, 118
 - facet
 - domain, 201
 - language, 66
 - prompt
 - domain, 27, 65, 66, 109
 - prompts
 - analysis &, 66
 - tree
 - domain, 118
- description prompts
 - [endurant]
 - domain, 113
 - domain
 - [endurant], 113
- design
 - assumptions, 170
 - requirements, 170
 - software
 - specification, 157
 - specification
 - software, 157
- determination, 169–171
- development, 200
 - domain
 - requirements, 170
 - interface
 - requirements, 170
 - requirements, 200
 - domain, 170
 - interface, 170
 - software
 - trptych, 102
- trptych
 - software, 102
- discrete endurant, 391
- documentation, 200
- domain, 9, 71
 - [endurant]
 - analysis prompts, 113
 - description prompts, 113
 - analysis
 - prompt, 65
 - analysis &
 - description, 9
 - analysis prompts
 - [endurant], 113
 - description, 157
 - analysis &, 9
 - facet, 201
 - prompt, 27, 65, 66, 109
 - tree, 118
 - description prompts
 - [endurant], 113
 - development
 - requirements, 170
 - engineering, 158, 201
 - extension
 - requirements, 192
 - external
 - interfaces, 9
 - facet, 71, 169
 - description, 201
 - intrinsic, 72
 - support technology, 75
 - intrinsic, 169
 - interfaces
 - external, 9
 - intrinsic, 72
 - facet, 72
 - manifest, 71
 - partial
 - requirement, 190, 191
 - prescription
 - requirements, 171
 - prompt
 - analysis, 65
 - description, 27, 65, 66, 109
 - requirement
 - partial, 190, 191
 - shared, 190, 191
 - requirements, 169
 - development, 170
 - extension, 192

- prescription, 171
 - semantic, 123
 - shared
 - requirement, 190, 191
 - support technology
 - facet, 75
 - syntactic, 123
 - tree
 - description, 118
- domain requirements
 - partial
 - prescription, 171
 - prescription
 - partial, 171
- embedded
 - attribute
 - sharing, 189, 196
 - sharing
 - attribute, 189, 196
- endurant, 67, 103, 391
 - discrete, 391
 - shared, 158, 169
- engineering
 - domain, 158, 201
 - requirements, 158, 201
 - software, 157
- entities, 67
- entity, 391
- entry, 186
- entry,, 184
- epistemology, 66
- Euclid of Alexandria, 38, 228
- event, 42, 67, 232
 - shared, 158, 169
- execution, 200
- exit, 186
- exit,, 184
- expression
 - function
 - type, 47
 - type, 47
 - function, 47
- extension, 75, 169–171
 - conservative, 182
 - proof theoretic, 75
 - domain
 - requirements, 192
 - proof theoretic
 - conservative, 75
 - requirements
 - domain, 192
- extensional, 200
- external
 - attribute, 183, 185, 189
 - domain
 - interfaces, 9
 - interfaces
 - domain, 9
 - part
 - quality, 108
 - qualities, 67
 - quality
 - part, 108
- facet, 71
 - description
 - domain, 201
 - domain, 71, 169
 - description, 201
 - intrinsic, 72
 - support technology, 75
 - intrinsic
 - domain, 72
 - machine
 - requirement, 200
 - requirement
 - machine, 200
 - specific, 72
 - support technology
 - domain, 75
- fitting, 169–171
- formal
 - method
 - software development, 102
 - software development
 - method, 102
 - specification, 157
- formalisation, 7
- function, 7
 - constructor
 - type, 47
 - expression
 - type, 47
 - name, 47
 - type
 - constructor, 47
 - expression, 47
- goal, 202
- guarantee, 201
- rely, 201

- has_ concrete_ type
 - prerequisite
 - prompt, 21
 - prompt
 - prerequisite, 21
- human behaviour, 71
- identifier
 - unique, 24, 25
- implementation
 - partial, 81
- instantiation, 169–171
- intrinsic, 169
 - domain, 169
- integrity, 200
- intensive, 72
- interface, 9
 - development
 - requirements, 170
 - requirements, 169, 183, 190, 191
 - development, 170
- interface
 - requirements, 158
- interfaces
 - domain
 - external, 9
 - external
 - domain, 9
 - internal
 - system, 9
 - system
 - internal, 9
- internal
 - interfaces
 - system, 9
 - part
 - quality, 108
 - qualities, 12–14, 67, 105
 - quality
 - part, 108
 - system
 - interfaces, 9
- interval
 - time, 43
- intrinsic, 71
 - domain, 72
 - facet, 72
 - facet
 - domain, 72
- language
 - analysis, 65
 - description, 66
 - license, 84, 88
 - license languages, 71
 - licensee, 84, 88
 - licensing, 88
 - licensor, 84, 88
- machine
 - facet
 - requirement, 200
 - requirement, 200
 - facet, 200
 - requirements, 169, 200
- maintenance, 200
 - requirements, 200
- management, 200
- management & organisation, 71
- manifest
 - domain, 71
- mathematical
 - object, 157
- mereology, 16
 - observer, 26
 - type, 26
 - update, 103
- method, 71
 - formal
 - software development, 102
 - software development
 - formal, 102
- modelling
 - requirements, 9
- monitor, 81
- name
 - function, 47
- narration, 7
- object
 - mathematical, 157
- objective
 - operational, 202
- objects
 - computable, 65
- obligation, 88
 - proof, 7
- observe, 10, 104, 222
- observe_ part_ type
 - prerequisite
 - prompt, 21
 - prompt

- prerequisite, 21
- observer
 - mereology, 26
- ontology, 66
- operational
 - objective, 202
- operations research, 99
- parallelism, 57
- part, 14, 106, 391
 - atomic, 391
 - composite, 391
 - derivation, 143
 - external
 - quality, 108
 - internal
 - quality, 108
 - quality
 - external, 108
 - internal, 108
 - sort, 18
 - sub-, 391
- partial
 - domain
 - requirement, 190, 191
 - domain requirements
 - prescription, 171
 - implementation, 81
 - prescription
 - domain requirements, 171
 - requirement
 - domain, 190, 191
- perdurant, 67, 103
- perfective, 200
- performance, 200
- permission, 88
- permit, 84
- phenomena
 - shared, 158
- philosophy, 66
- platform, 200
 - requirements, 200
- pragmatics, 67
- prerequisite
 - has_ concrete_ type
 - prompt, 21
 - observe_ part_ type
 - prompt, 21
 - prompt
 - has_ concrete_ type, 21
 - observe_ part_ type, 21
- prescription
 - domain
 - requirements, 171
 - domain requirements
 - partial, 171
 - partial
 - domain requirements, 171
 - requirements, 157, 201
 - domain, 171
- preventive, 200
- principles, 71
- process, 200
- program
 - computer, 157
- projection, 169–171
 - common, 190
 - specific, 190
- prompt
 - analysis
 - domain, 65
 - description
 - domain, 27, 65, 66, 109
 - domain
 - analysis, 65
 - description, 27, 65, 66, 109
 - has_ concrete_ type
 - prerequisite, 21
 - observe_ part_ type
 - prerequisite, 21
 - prerequisite
 - has_ concrete_ type, 21
 - observe_ part_ type, 21
- prompts
 - analysis &
 - description, 66
 - description
 - analysis &, 66
- proof
 - obligation, 7
- proof theoretic
 - conservative
 - extension, 75
 - extension
 - conservative, 75
- qualities
 - external, 67
 - internal, 12–14, 67, 105
- quality
 - external
 - part, 108

- internal
 - part, 108
- part
 - external, 108
 - internal, 108
- reliability, 200
- rely
 - guarantee, 201
- requirement
 - domain
 - partial, 190, 191
 - shared, 190, 191
 - facet
 - machine, 200
 - machine, 200
 - facet, 200
 - partial
 - domain, 190, 191
 - shared
 - domain, 190, 191
- requirements
 - dependability, 200
 - derived, 158, 169
 - design, 170
 - development, 200
 - domain, 170
 - interface, 170
 - domain, 169
 - development, 170
 - extension, 192
 - prescription, 171
 - engineering, 158, 201
 - extension
 - domain, 192
 - interface, 169, 183, 190, 191
 - development, 170
 - interface , 158
 - machine, 169, 200
 - maintenance, 200
 - modelling, 9
 - platform, 200
 - prescription, 157, 201
 - domain, 171
 - technology, 200
- robustness, 200
- rules & regulations, 71
- safety, 200
- science
 - computer &
 - computing, 65, 66
 - computing
 - computer &, 65, 66
 - scripts, 71
 - security, 200
 - semantic
 - domain, 123
 - semantics, 67
 - semiotic, 67
 - shared
 - action, 158, 169
 - attribute, 103
 - behaviour, 158, 169
 - domain
 - requirement, 190, 191
 - endurant, 158, 169
 - event, 158, 169
 - phenomena, 158
 - requirement
 - domain, 190, 191
 - sharing
 - attribute
 - embedded, 189, 196
 - embedded
 - attribute, 189, 196
 - simplification, 157, 172
 - simplify, 174
 - software
 - design
 - specification, 157
 - development
 - tritych, 102
 - engineering, 157
 - specification
 - design, 157
 - tritych
 - development, 102
 - software development
 - formal
 - method, 102
 - method
 - formal, 102
 - sort, 7, 391
 - part, 18
 - specific
 - facet, 72
 - projection, 190
 - specification
 - design
 - software, 157
 - formal, 157

- software
 - design, 157
- state, 41
 - change, 47
- sub-part, 14, 15, 106, 391
- support
 - technology, 190
- support technology, 71
 - domain
 - facet, 75
 - facet
 - domain, 75
- synchronisation, 57
- syntactic
 - domain, 123
- syntax, 67
- system
 - interfaces
 - internal, 9
 - internal
 - interfaces, 9
- techniques, 71
- technology
 - requirements, 200
 - support, 190
- time, 41, 43
 - continuous, 47
 - interval, 43
- tools, 71
- transcendental
 - deduction, 3, 67

- tree
 - description
 - domain, 118
 - domain
 - description, 118
- TripTych, 28, 203
- triptych
 - development
 - software, 102
 - software
 - development, 102
- type, 7, 391
 - abstract, 391
 - concrete, 391
 - constructor
 - function, 47
 - expression, 47
 - function, 47
 - function
 - constructor, 47
 - expression, 47
 - mereology, 26
- unique
 - identifier, 24, 25
- unique identifier, 394
- update
 - attribute, 103
 - mereology, 103
- value
 - abstract, 24

I.3 **Examples**

Domain Requirements

- Derived Action:
 - Tracing Vehicles (# 5.16), 197
- Derived Event:
 - Current Maximum Flow (# 5.17), 198
- Determination
 - Toll-roads (# 5.9), 180
- Endurant Extension (# 5.10), 183
- Fitting (# 5.11), 191
- Instantiation
 - Road Net (# 5.7), 176
 - Road Net, Abstraction (# 5.8), 179
- Projection (# 5.6), 172
- Projection:
 - A Narrative Sketch (# 5.5), 172

Interface Requirements

- Projected Extensions (# 5.12), 192
- Shared
 - Endurant Initialisation (# 5.14), 193
 - Endurants (# 5.13), 192
 - Shared Behaviours (# 5.15), 196

Road Pricing System

- Design Assumptions (# 5.2), 170
- Design Requirements (# 5.1), 170

Toll-Gate System

- Design Assumptions (# 5.4), 171
- Design Requirements (# 5.3), 171

I.4 Analysis Prompts

- a. is_entity, 10
- b. is_endurant, 11
- c. is_perdurant, 11
- d. is_discrete, 11
- e. is_continuous, 11
- f. is_physical_part, 12
- g. is_living_species, 12
- h. is_structure, 13
- i. is_part, 14
- j. is_atomic, 15
- k. is_composite, 15
- l. is_living_species, 15
- m. is_plant, 15
- n. is_animal, 16
- o. is_human, 16
- p. has_components, 17
- q. has_materials, 17
- r. is_artefact, 17
- s. observe_endurant_sorts, 18
- t. has_concrete_type, 21
- u. has_mereology, 26
- v. attribute_types, 29

I.5 Description Prompts

- [1] observe_endurant_sorts, 19
- [2] observe_part_type, 21
- [3] observe_component_sorts, 22
- [4] observe_material_sorts, 24
- [5] observe_unique_identifiers, 25
- [6] observe_mereology, 27
- [7] observe_attributes, 29

I.6 Attribute Categories

- is_active_attribute, 30, 146
- is_autonomous_attribute, 30, 146
- is_biddable_attribute, 30, 146
- is_dynamic_attribute, 30, 146
- is_inert_attribute, 30, 146
- is_programmable_attribute, 31, 146
- is_reactive_attribute, 30, 146
- is_static_attribute, 30
- is_static_attribute, 146

I.7 RSL Symbols

Literals, 478–488

Unit, 488

chaos, 478, 480

false, 472, 474

true, 472, 474

Arithmetic Constructs, 474

$a_i * a_j$, 474

$a_i + a_j$, 474

a_i / a_j , 474

$a_i = a_j$, 474

$a_i \geq a_j$, 474

$a_i > a_j$, 474

$a_i \leq a_j$, 474

$a_i < a_j$, 474

$a_i \neq a_j$, 474

$a_i - a_j$, 474

Cartesian Constructs, 475, 478

(e_1, e_2, \dots, e_n) , 475

Combinators, 484–487

... elsif ..., 485

case b_e **of** $pa_1 \rightarrow c_1, \dots, pa_n \rightarrow c_n$ **end**, 485, 486

do $stmt$ **until** be **end**, 487

for e **in** $list_{expr}$ **•** $P(b)$ **do** $stm(e)$ **end**, 487

if b_e **then** c_c **else** c_a **end** , 485, 486
let $a:A \bullet P(a)$ **in** c **end** , 484
let $pa = e$ **in** c **end** , 484
variable $v:Type := expression$, 486
while b_e **do** stm **end** , 487
 $v := expression$, 486

Function Constructs, 483

post $P(args, result)$, 483
pre $P(args)$, 483
 $f(args)$ **as** $result$, 483
 $f(a)$, 482
 $f(args) \equiv expr$, 483
 $f()$, 486

List Constructs, 475–476, 478–480

$\langle Q(l(i)) \mid i \text{ in } \langle 1..len \rangle \bullet P(a) \rangle$, 476
 $\langle \rangle$, 476
 $l(i)$, 479
 $l' = l''$, 479
 $l' \neq l''$, 479
 $l' \sim l''$, 479
elems l , 479
hd l , 479
inds l , 479
len l , 479
tl l , 479
 $e_1 \langle e_2, e_2, \dots, e_n \rangle$, 476

Logic Constructs, 473–474

$b_i \vee b_j$, 474
 $\forall a:A \bullet P(a)$, 474
 $\exists! a:A \bullet P(a)$, 474
 $\exists a:A \bullet P(a)$, 474
 $\sim b$, 474
false, 472, 474
true, 472, 474
 $b_i \Rightarrow b_j$, 474
 $b_i \wedge b_j$, 474

Map Constructs, 476, 480–482

$m_i \circ m_j$, 481
 $m_i \Gamma E30F m_j$, 481
 m_i / m_j , 481
dom m , 480
rng m , 480
 $m_i = m_j$, 481
 $m_i \cup m_j$, 480
 $m_i \dagger m_j$, 480
 $m_i \neq m_j$, 481
 $m(e)$, 480

$[\]$, 476
 $[u_1 \mapsto v_1, u_2 \mapsto v_2, \dots, u_n \mapsto v_n]$, 476
 $[F(e) \mapsto G(m(e)) \mid e:E \bullet e \in \text{dom } m \wedge P(e)]$, 476

Process Constructs, 487–488

channel $c:T$, 487
channel $\{k[i]:T \bullet i:\text{Idx}\}$, 487
 $c!e$, 487
 $c?$, 487
 $k[i]!e$, 487
 $k[i]?$, 487
 $p_i \square p_j$, 487
 $p_i \sqcap p_j$, 487
 $p_i \parallel p_j$, 487
 $p_i \dashv p_j$, 487
 $P: \text{Unit} \rightarrow \text{in } c \text{ out } k[i] \text{ Unit}$, 488
 $Q: i:\text{KIdx} \rightarrow \text{out } c \text{ in } k[i] \text{ Unit}$, 488

Set Constructs, 475, 477–478

$\cap \{s_1, s_2, \dots, s_n\}$, 477
 $\cup \{s_1, s_2, \dots, s_n\}$, 477
card s , 477
 $e \in s$, 477
 $e \notin s$, 477
 $s_i = s_j$, 477
 $s_i \cap s_j$, 477
 $s_i \cup s_j$, 477
 $s_i \subset s_j$, 477
 $s_i \subseteq s_j$, 477
 $s_i \neq s_j$, 477
 $s_i \setminus s_j$, 477
 $\{\}$, 475
 $\{e_1, e_2, \dots, e_n\}$, 475
 $\{Q(a) \mid a:A \bullet a \in s \wedge P(a)\}$, 475

Type Expressions, 471–472

$(T_1 \times T_2 \times \dots \times T_n)$, 472
Bool, 471
Char, 471
Int, 471
Nat, 471
Real, 471
Text, 471
Unit, 486
 $mk_id(s_1:T_1, s_2:T_2, \dots, s_n:T_n)$, 472
 $s_1:T_1 \ s_2:T_2 \ \dots \ s_n:T_n$, 472
 T^* , 472
 T^ω , 472
 $T_1 \times T_2 \times \dots \times T_n$, 472
 $T_1 \mid T_2 \mid \dots \mid T_1 \mid T_n$, 472
 $T_i \xrightarrow{m} T_j$, 472
 $T_i \xrightarrow{\sim} T_j$, 472

$T_i \rightarrow T_j$, 472

T-infset, 472

T-set, 472

Type Definitions, 472–473

$T = \text{Type_Expr}$, 472

$T = \{ | v: T' \bullet P(v) | \}$, 473

$T = TE_1 | TE_2 | \dots | TE_n$, 473