A Philosophy of Domain Science & Engineering*

An Interpretation of Kai Sørlander's Philosophy

Dines Bjørner†
Fredsvej 11, DK-2840 Holte, Danmark
E--Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db‡

March 18, 2018: 18:35

1
2

Abstract

We suggest a **philosophy basis** for **domain science & engineering**. This paper is based on recent research [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] into methods for analysing and describing human-based universes of discourses such as transport nets, container lines, pipelines, drones, urban planning, etc. It is motivated by my speculations about the "interface" between domain analysis & description methods and the reality they model. A major section of the paper is based on 10 years of research into and experimental use of (the citation referenced) calculi for domain analysis & description. Another major segment of the paper is based on the philosophy of Kai Sørlander [13, 14, 15, 16].

Initial versions of this document are in the form of a report. As such it collects far more material than should be contained in a proper paper. Most of the "extra", i.e., the report material is collected from various sources but drastically edited by me. Most of the material of Sect. 8 is extracted from [16].

Contents

*First reading: The Victor Ivannikov Memorial Event, May 3–4, 2018, Yerevan, Armenia
†Margin numbers refer to slide numbers of a lecture (series) version of this paper.
‡This paper and its slide presentation version can be found at: http://www.imm.dtu.dk/~dibj/2018/-philosophy/filo.pdf, respectively http://www.imm.dtu.dk/~dibj/2018/philosophy/filo-oh.pdf

There are two aspects to this paper: (i) the analysis & description of fragments of the context in which software, to be developed, is to serve, (ii) and the general, basically philosophical, problem of the absolutely necessary conditions for describing the world.

In twelve papers we have put forward a method for analysing and describing the domains for which software is developed:

- [1, 2] Manifest Domains: Analysis & Description **FAoC, March 2017**
- [3, 4] Domain Facets: Analysis & Description
- [5, 6] Formal Models of Processes and Prompts
- [7, 8] To Every Manifest Domain Mereology a CSP Expression **LAMP, Jan. 2018**
- [9, 10] From Domain Descriptions to Requirements Prescriptions
- [11, 12] Domains: Their Simulation, Monitoring and Control

In four books the Danish philosopher Kai Sørlander has investigated the philosophical issues alluded to above.

- [13] Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions] Forord/Foreword: Georg Henrik von Wright*. Munksgaard · Rosinante, 1994. 168 pages.

- [14] Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, 1997. 200 pages.
- [15] Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, 2002. 187 pages.
- [16] Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, 2016. 233 pages.

1.2 The Thesis of This Paper 6

TO BE WRITTEN

1.3 The Computing Science Background 7

1.3.1 Computer & Computing Science

- By **computer science** I understand the study and knowledge
  of the "things" that can "exist inside" computing devices (i.e., data and computations)
  –
  and the study and knowledge of these computing devices.
- By **computing science** I understand the study and knowledge
  of how to construct "those things", i.e., **programming methodology**.

I consider myself a computing scientist primarily interested in programming methodology.

1.3.2 Formal Methods 8

- By a **method** I understand
  a set of **principles** for **selecting** and **applying** a set of **techniques** and **tools**
  for the **construction** of an artifact, as here software.
- By a **formal method** I understand I understand
  a method whose principles, techniques and tools
  can be understood in a mathematical framework –
  for example where, among the tools the **specification languages** can be given
  a **mathemtical syntax**, a **mathematical semantics** and a **mathematical proof system**.

I consider myself to have primarily contributed to the area of formal methods, i.e., VDM and RAISE.

1.3.3 A Triptych of Engineering 9

- Before software can be designed we must be familiar with its requirements.
- Before requirements can be prescribed we must be familiar with the context of the software to be developed, that is, the domain. 10
- Hence the triptych of software development:

- first (ideally) the domain engineering of an appropriate domain description;
- then (ideally) the requirements engineering of the requirements prescription – formally related to the domain description;
- finally the software design "derived" from the requirements prescription and (ideally) formally reasoned to meet customers' expectations, that is, to satisfy the domain description and be correct wrt. the requirements prescription.

11

My contributions in the last many years has been to establish a proper domain science & engineering. My main focus, since 1977, has been on the development of "large" software: compilers (like for `CHILL` and `Ada`), and infrastructure software – for pipelines, railways, health care, banking, road traffic, etc.

1.4 Domains, their Analysis & Description, and a Method

12

In this section we shall brief outline what we mean by a **domain**, what we mean by **domain analysis & description**, and what we mean by a **method for analysing & describing domains**.

1.4.1 Domains

13

Definition 1: **Domain:** By a **domain** we shall understand a human-centered aggregation of endurants, i.e., spatially enduring entities, and perdurants, i.e., temporally perduring entities that together serve a purpose in solving problems ■

1.4.2 Domain Analysis & Description

Definition 2: **Domain Analysis and Description:** By **domain analysis and description** we shall understand the analysis & description of domains ■

1.4.3 A Domain Analysis & Description Method

14

Definition 3: **A Domain Analysis and Description Method:** By a **domain analysis and description method** we shall understand a set of principles, techniques and tools for the construction, i.e., analysis & description of a domain model ■

The terms *description* and *model* are here considered synonymous.

1.5 Structure of This Paper

15

The present, March 18, 2018: 18:35 , version of this document contains "far more" material than an expected paper will contain. The writing of Sects. 9–11 began on March 12, 2018. Thus the present version shall serve as a basis for further research, further written
16 formulations, presentations, etc. before the editing of a final paper version is begun.

The paper has two main segments: Segment I and Segment III. Segment II provides a bridge between these two segments.

Segment I summarises the *domain analysis & description method*. It itself has four sections. *Endurants* (Sect. 2), *A Transcendental Transformation* (Sect. 3), *Perdurants* (Sect. 4) and *"A*

17 *Coin has Two Sides"* (Sect. 5). Section 2 outlines the *domain analysis & description prompts* whose deployment aids the domain analyser cum describer to develop a model of domain endurants. Section 3 motivates the interpretation of domain parts as behaviours. Section 4 outlines a *"translation"* of parts into CSP [17] behaviours. Section 5 (one page !) reflects on the two issues: parts and behaviours. 18

Sections 2 and 4 (themselves) are without Examples. Instead Appendix A (Pages 42–52) offers a comprehensive illustrations of almost all the description prompts and so-called process schemes of Sects. 2 and 4. Subsections of Sect. 2 and 4 are carefully correlated to corresponding subsections of Appendix A, and vice versa. 19

Segment III provides the main reason for this paper. Intertwined with references to the *domain analysis & description calculi* – of Segment I – it outlines, in rather terse form, core issues of Kai Sørlander's philosophy. In doing so it attempts to establish *a basis in philosophy* for *domain science & engineering*. 20

Segment II bridges Segments I and III. In Segment I the analysis & description method did not "touch upon" the domain phenomena of space and time. Kai Sørlander's philosophy, in contrast to Immanuel Kant's [1724–1812] *Critique of Pure Reason*[1] (`Kritik der reinen Vernunft`, 1781, does not -suppose space and time but derive them as indispensable in any analysis & description of the world. Therefore we treat them separately, first in Segment II, then in Segment III's Sect. 9.2. 21

Segment I: The Domain Analysis & Description Calculi

2 Endurants – see Appendix A.2 Pg. 42 22

In a series of *definitions*, most of which are rather like *characterisations*[2], we shall *explicate* a number of domain concepts. These definitions will lead to the introduction of initially *domain analysis prompts*, subsequently also *domain description prompts*. Think of a **prompt** as a *cue*, 23 a *hint*, a *suggestion*, in German, a *stichwort, suchbegriff*, in French, a *signal théâtre*, that the domain analyser is told, by the principles of the domain analysis & description method, to act upon.

2.1 The Universe of Discourse – see Appendix A.1 Pg. 42 24

The domain-of-interest needs first be briefly narrated. Just a simple story. One that emphasises the "main players": the endurants and the perdurants.

2.2 Basic Domain Concepts 25

Definition 4 *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*

[1] http://www.gutenberg.org/files/4280/4280-0.txt

[2] Usually, in computer science papers, definitions are terse and based on more-or-less implicit reference to a mathematically precise model. Since domains do not have an a-priori mathematically precise model our definitions cannot be precise. Most of the definitions are taken from such dictionaries as [18, The Oxford Shorter English Dictionary] and from the Internet based [19, The Stanford Encyclopedia of Philosophy].

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 [18, Vol. I, pg. 665]* ■

Analysis Prompt 1 *is_entity:* *The domain analyser analyses "things" (θ) into either entities or non-entities. The method can thus be said to provide the* **domain analysis prompt***:*

- *is_entity* – *where* `is_entity(θ)` *holds if θ is an entity* ■[3]

Definition 5 *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" [18, Vol. I, pg. 656]. Were we to "freeze" time we would still be able to observe the entire endurant* ■

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

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

`is_entity` *is a* prerequisite prompt *for* `is_endurant` ■

Definition 6 *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, that is, 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 [18, Vol. II, pg. 1552]* ■

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` ■

Definition 7 *Discrete Endurant: By a* **discrete endurant** *we shall understand an endurant which is separate, individual or distinct in form or concept* ■

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* ■

Definition 8 *Continuous Endurant: By a* **continuous endurant** *we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern* ■

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',

2.3 An Upper Ontology Diagram of Domains – A Preview

Figure 1 shows a so-called upper ontology for manifest domains. So far we have covered only a fraction of this ontology, as noted. By ontologies we shall here understand *formal representations of a set of concepts within a domain and the relationships between those concepts*.

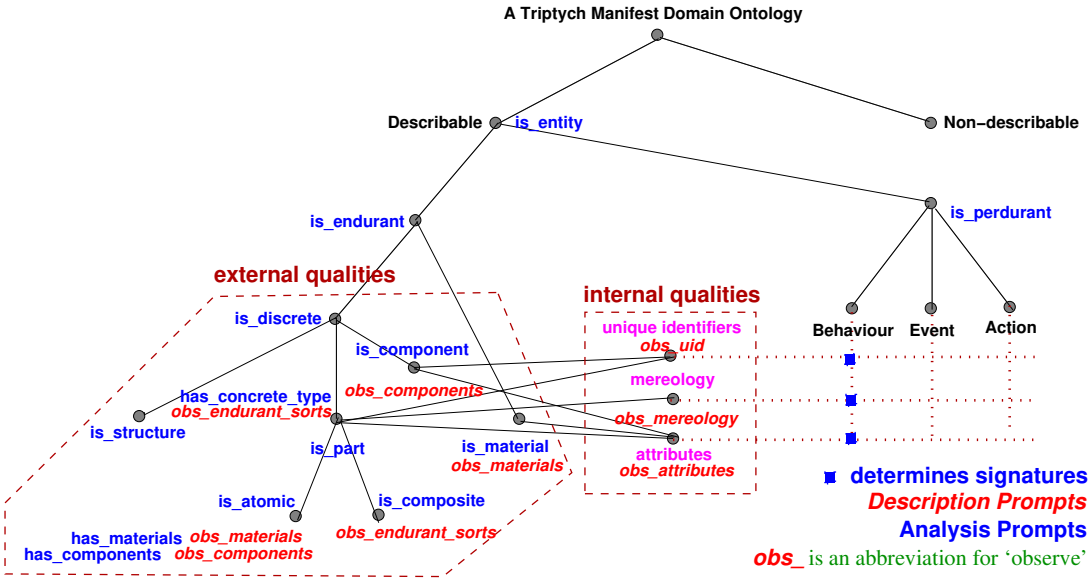


Figure 1: An Upper Ontology for Domains

2.4 Structures – see Appendix A.2.1 Pg. 42

Definition 9 *Structure: By a* **structure** *we shall understand a discrete endurant which the domain engineer chooses to describe as consisting of sub-structures, parts, components, but to* **not** *endow with* **internal qualities**: *unique identifiers, mereology or attributes* ■

[3] Analysis prompt definitions and description prompt definitions and schemes are delimited by ■.

We shall define the terms unique identification, mereology and attributes in Sects. A.2.7–A.2.10.

Analysis Prompt 6 *is_structure: The domain analyser analyse endurants, e, into structure entities as prompted by the* **domain analysis prompt***:*

- *is_structure* ■

2.5 Parts, Components and Materials – see Appendix A.2.2 Pg. 43

2.5.1 Parts – see Appendix A.2.3 Pg. 43

Definition 10 *Part: By a* **part** *we shall understand a discrete endurant which the domain engineer chooses to endow with all three* **internal qualities***: unique identification, mereology, and one or more attributes* ■

Analysis Prompt 7 *is_part: The domain analyser analyse endurants, e, into part entities as prompted by the* **domain analysis prompt***:*

- *is_part* ■

Definition 11 *Atomic Part:* **Atomic part***s 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 8 *is_atomic: The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:*

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

Definition 12 *Composite Part:* **Composite part***s are those which, in a given context, are deemed to indeed consist of meaningful, separately observable proper sub-parts* ■

Analysis Prompt 9 *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* ■

Analysis Prompt 10 *observe_endurant: The* **domain analysis prompt***:*

- *observe_endurants*

directs the domain analyser to observe the sub-endurants of an endurant e and to suggest their sorts.

Let `observe_endurants`$(e) = \{e_1{:}E_1, e_2{:}E_2, \ldots, e_m{:}E_m\}$ ■

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 schema

Narration:
[s] ... narrative text on sorts ...
[o] ... narrative text on sort observers ...
[η] ... narrative text on sort type observers ...
[i] ... narrative text on sort recognisers ...
[p] ... narrative text on proof obligations ...

Formalisation:

type
[s] P,
[s] E_i i:[1..m] **comment:** E_i i:[1..m] abbreviates E_1, E_2, ..., E_m

value
[o] **obs_endurant_sorts_**E_i: P $\rightarrow$ E_i i:[1..m]
[η] **if** is_part(e_i): η(e_i) $\equiv$ $\ll$ E_i $\gg$ i:[1..m]
[i] **is_**E_i: $(E_1|E_2|...|E_m)$ $\rightarrow$ **Bool** i[1..m]

proof obligation [Disjointness of endurant sorts]
[p] $\mathcal{PO}$: $\forall$ e:($E_1|E_2|...|E_m$) •
[p] $\bigwedge$ {**is_**E_i(e) $\equiv$ $\bigwedge$ {$\sim$**is_**E_j(p) | j:[1..m] \ {i}} | i:[1..m]}

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

- *observe_part_type(p)*[4]

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 schema

Narration:
[t_1] ... narrative text on sorts and types S_i ...
[t_2] ... narrative text on types T ...
[t_3] ... narrative text on type of value observer
[o] ... narrative text on type observers ...

Formalisation:

type
[t_1] S_1, S_2, ..., S_m, ..., S_n,

[4] has_concrete_type is a *prerequisite prompt* of observe_part_type.

[t2] T = $\mathcal{E}$(S1,S2,...,Sn)
[t3] η(si) ≡ ≪ S ≫ , i:[1..n],si:Si
value
[o] **obs_part_**T: P → T ■

2.5.2 Components – see Appendix A.2.4 Pg. 44 48

Definition 13 *Component: By a* **component** *we shall understand a discrete endurant which we, the domain analyser cum describer chooses to* **not** *endow with* **mereology** ■

Analysis Prompt 11 *is_component: The domain analyser analyse endurants e into component entities as prompted by the* **domain analysis prompt***:*

- *is_component* ■

49

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

- *observe_component_sorts_P(p)*

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:

50

3. observe_component_sorts_P schema

Narration:
[s] ... narrative text on component sorts ...
[o] ... narrative text on component observers ...
[i] ... narrative text on component sort recognisers ...
[u] ... narrative text on unique identifier ...
[p] ... narrative text on component sort proof obligations ...
Formalisation:
type
[s] K1, K2, ..., Kn
[s] K = K1| K2 | ... | Kn
[s] KS = K-**set**
value
[o] **obs_components_**P: P → KS
[i] **is_**Ki: (K1|K2|...|Kn) → **Bool** i:[1..n]
[u] **uid_**Ki
Proof Obligation: [Disjointness of Component Sorts]
[p] $\mathcal{PO}$: ∀ ki:(K1|K2|...|Kn) •
[p] ⋀ {**is_**Ki(ki) ≡ ⋀{∼**is_**Kj(kj)|j:[1..n] \ {i}}} i:[1..n] ■

2.5.3 Materials – see Appendix A.2.5 Pg. 44 51

Definition 14 *Material: By a* **material** *we shall understand a continuous endurant* ■

Analysis Prompt 12 *is_material: The domain analyser analyse endurants e into material entities as prompted by the* **domain analysis prompt***:*

- *is_material* ■

52

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

- *observe_material_sorts_P(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: 53

4. observe_material_sorts_P schema

Narration:
[s] ... narrative text on material sorts ...
[o] ... narrative text on material sort observers ...
[i] ... narrative text on material sort recognisers ...
[p] ... narrative text on material sort proof obligations ...
Formalisation:
type
[s] M1, M2, ..., Mn
[s] M = M1 | M2 | ... | Mn
[s] MS = M-**set**
[a] Ai = A11 | A12 | ... | A1n
value
[o] **obs_mat_sort**_M_i: P → M, [i:1..n]
[o] **obs_materials**_P: P → MS
[i] **is**_M_i: M → **Bool** [i:1..n]
[a] **attr**_A_{ij}: M_i → A_{ij} [i:...,j:...]
proof obligation [Disjointness of Material Sorts]
[p] $\mathcal{PO}$: ... ■

2.6 Unique Part and Component Identifiers – see Appendix A.2.7 Pg. 44 54

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 identifier*s, (ii) that *unique identifier*s (of parts and components p:P) are *abstract value*s (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 π_i:PI_i and π_j:PI_j are distinct, and (v) that the observer function **uid**_P applied to p yields the unique identifier, say π:PI, of p. 55

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 schema

Narration:
- [s] ... narrative text on unique identifier sort PI ...
- [u] ... narrative text on unique identifier observer **uid**_P ...
- [η] ... narrative text on type name, an RSL$^+$Text observer ...
- [a] ... axiom on uniqueness of unique identifiers ...

Formalisation:

type
- [s] PI

value
- [u] **uid**_P: P $\to$ PI
- [u] η PI $\to$ $\ll$ P $\gg$

axiom [Disjointness of Domain Identifier Types]
- [a] $\mathcal{A}$: $\mathcal{U}$(PI,PI_i,PI_j,...,PI_k) ■

2.7 Part Mereologies – see Appendix A.2.9 Pg. 45

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 [20, 21].

2.7.1 Part Relations

Which are the relations that can be relevant for part-hood? We give some examples. (i) Two otherwise distinct parts may *"share"* values.[5] By *'sharing'* values we shall, as a generic example, mean that two parts of different sorts has the same attributes but that one *'defines'* the attribute, like, for example *'programming'* its values, cf. Defn.8 Page20, whereas the other

[5] For the concept of attribute value see Sect. 2.8.2 on Page 18.

'uses' these values, like, for example considering them *'inert'*, cf. Defn.3 Page19. (ii) Two otherwise distinct parts may be said to, for example, be topologically "adjacent" or one "embedded" within the other. These examples are in no way indicative of the "space" of part relations that may be relevant for part-hood. The domain analyser is expected to do a bit of experimental research in order to discover necessary, sufficient and pleasing "mereology-hoods" ! 60

2.7.2 Part Mereology: Types and Functions 61

Analysis Prompt 13 *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). 62

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: 63

6. observe_mereology 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[6]
value
[m] **obs_mereo_**P: P → MT
axiom [Well−formedness of Domain Mereologies]
[a] $\mathcal{A}$: $\mathcal{A}$(MT)

2.8 Part Attributes – see Appendix A.2.10 Pg. 46 64

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 endurant qualities. Part attributes form more "free-wheeling" sets of **internal qualities**.

2.8.1 Inseparability of Attributes from Parts and Materials 65

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 touched[7], or seen[8], but can be objectively measured[9]. 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) !

2.8.2 Attribute Quality and Attribute Value 66

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 14 *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)* $= \{A_1, A_2, ..., A_m\}$.

2.8.3 Part and Material Attributes: Types and Functions 67

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,
68 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*
69 *following schema:*

[7] One can see the red colour of a wall, but one touches the wall.

[8] 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.

[9] 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.

7. observe_attributes schema

Narration:
[t] ... narrative text on attribute sorts ...
[o] ... narrative text on attribute sort observers ...
[v] ... narrative text on set **of** attribute **value** observers ...
[i] ... narrative text on attribute sort recognisers ...
[p] ... narrative text on attribute sort proof obligations ...

Formalisation:
type
[t] A_i [$1 \leq i \leq n$]
value
[o] **attr_**A_i:P→A_i i:[1..n]
[v] **obs_attrib_values_**P(p) ≡ { **attr_**A_1(p),**attr_**A_2(p),...,**attr_**A_n(p) }
[i] **is_**A_i:(A_1|A_2|...|A_n)→**Bool** i:[1..n]
proof obligation [Disjointness of Attribute Types]
[p] $\mathcal{PO}$: **let** P be any part sort **in** [the domain description]
[p] **let** a:(A_1|A_2|...|A_n) **in is_**A_i(a) ≠ **is_**A_j(a) **end end** [i≠i, i,j:[1..n]]

2.8.4 Attribute Categories

70

Michael A. Jackson [22] 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 [22, M.A. Jackson]. **Part attributes** are either constant or varying, i.e., **static** or **dynamic** attributes. 71

Attribute Category: 1 By a **static attribute**, a:A, 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, 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.

72

Attribute Category: 3 By an **inert attribute**, a:A, 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, is_reactive_attribute(a), we shall understand dynamic attributes whose value, if they vary, change in response to external stimuli, where these stimuli come from outside the domain of interest.

73

Attribute Category: 5 By an **active attribute**, a:A, `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, `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 onto themselves and their surroundings".

Attribute Category: 7 By a **biddable attribute**, a:A, `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, `is_programmable_attribute`(a), we shall understand a dynamic active attribute whose values can be prescribed.

Figure 2 captures an attribute value ontology.

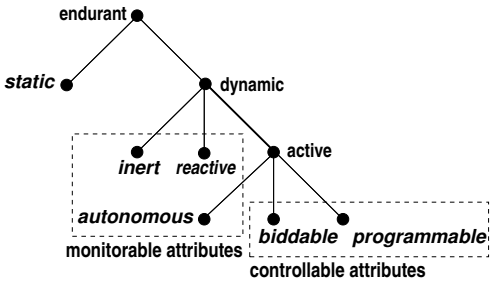


Figure 2: Attribute Value Ontology

1 Given a part p we can calculate its **static attributes**.

2 Given a part p we can calculate its **controllable attributes**, i.e., the biddable and programmable attributes.

3 And given a part p we can calculate its **monitorable attributes**, i.e., the inert, reactive and autonomous attributes.

4 These three sets make up all the attributes of part p.

```
value
1  stat_attr_typs: P → ⫷ SA1×SA2×...×SAs ⫸
2  ctrl_attr_typs: P → ⫷ CA1×CA2×...×CAc ⫸
3  mon_attr_typs: P → ⫷ MA1×MA2×...×MAm ⫸
```

```
axiom
4  ∀ p:P •
4      let ⫷ SA1×SA2×...×SAs ⫸ = stat_attr_typs(p),
4          ⫷ CA1×CA2×...×CAc ⫸ = ctrl_attr_typs(p),
4          ⫷ MA1×MA2×...×MAm ⫸ = mon_attr_typs(p) in
4      card{SA1,SA2,...,SAs}+card{CA1,CA2,...,CAc}+card{MA1,MA2,...,MAm}
4      = card{SA1,SA2,...,SAs,CA1,CA2,...,CAc,MA1,MA2,...,MAm} end
```

78

5 Given a part p we can calculate its static attribute values.

6 Given a part p we can calculate its controllable, i.e., the biddable and programmable attribute values.

```
value
5  stat_attr_vals: P → SA1×SA2×...×SAs
5  stat_attr_vals(p) ≡
5      let ⫷ SA1×SA2×...×SAs ⫸ = stat_attr_typs(p) in
5      (attr_SA1(p),attr_SA2(p),...,attr_SAs(p)) end

6  ctrl_attr_vals: P → CA1×CA2×...×CAc
6  ctrl_attr_vals(p) ≡
6      let ⫷ CA1×CA2×...×CAc ⫸ = ctrl_attr_typs(p) in
6      (attr_CA1(p),attr_CA2(p),...,attr_CAc(p)) end
```

3 A Transcendental Transformation – see Appendix A.3 Pg. 47

79

It should be clear to the reader that in domain analysis & description method we are reflecting on a number of philosophical issues. First and foremost on those of *epistemology* and *ontology*. In this section on a sub-field of epistemology, namely that of a number of issues of *transcendental* nature. We refer to [23, pp 878–880] [24, pp 807–810] [25, pp 54–55 (1998)].

80

Definition 15 *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 16 *Transcendental Transformation: By a* **transcendental transformation** *we shall understand the philosophical notion:* **a transcendental "conversion" of one kind of knowledge into a seemingly different kind of knowledge.**

Definition 17 *Transcendentality: By* **transcendentality** *we shall here mean the philosophical notion:* the state or condition of being transcendental.

81

Example 1 **Transcendentality**: We can speak of a bus in at least three *senses:*

(i) The bus as it is being `"assembled"` at an automobile plant;

(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**[10] ■

82

Example 1, 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.

4 Perdurants – see Appendix A.4 Pg. 48 83

So the transcendental deduction to be performed here is that of associating with each part – "existing" in space – a behaviour – "existing" in time.

Perdurants can thus be explained in terms of a notion of *state* and a notion of *time*. We 84 refer to Sect. 6.2 for a discussion of the concept of time.

To speak about behaviours, that is, to describe behaviours, we choose a model for behaviours. We choose that of `CSP` [17]. With `CSP` is associated the notions of *processes* (which serve to model behaviours), *channels*, `ch`, (which serve to model communication between behaviours), and *output*/*input* clauses: `ch ! v`, respectively `ch ?` which serves to express the offering of a value, `v` on channel `ch`, respectively the offering to accept such a value. We shall use these notions freely.

4.1 States – see Appendix A.2.6 Pg. 44 85

Definition 18 *State: By a* **state** *we shall understand any collection of* **parts** *or* **components** *or* **materials** ■

[10] – in this case rather: as a fragment of a bus time table *attribute*

4.2 On Actions, Events, Behaviours and Actors 86

To us perdurants are further, pragmatically, analysed into *action*s, *event*s, and *behaviour*s. 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.

4.2.1 Actors 87

Definition 19 *Actor: By an* **actor** *we shall understand something that is capable of initiating and/or* **carrying out** *actions, events or behaviours* ■

4.2.2 Discrete Actions 88

Definition 20 *Discrete Action: By a* **discrete action** *[26, 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, and for which an actor can be made responsible* ■

4.2.3 Discrete Events 89

Definition 21 *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, for which no particular domain actor can be made responsible* ■

4.2.4 Discrete Behaviours 90

Definition 22 *Discrete Behaviour: By a* **discrete behaviour** *we shall understand a set of sequences of potentially interacting sets of discrete actions, events and behaviours* ■

• • •

In this paper we shall omit consideration of concepts of continuous actions, events and behaviours.

4.3 Channels – see Appendix A.4.2 Pg. 48 91

The fact that a part, p of sort P with unique identifier p_i, has a mereology, for example the set of unique identifiers $\{q_a, q_b, ..., q_d\}$ identifying parts $\{qa, qb, ..., qd\}$ of sort Q, **may mean** that parts p and $\{qa, qb, ..., qd\}$ may wish to exchange – for example, attribute – values, one way (from p to the qs) or the other (vice versa) or in both directions. Figure 3 Pg. 24 shows 92
(left) two dotted rectangle box (part) and (right) two corresponing, rounded box (behaviour and channel) diagrams. 93

The **fact** that a part, p of sort P with unique identifier p_i, has a mereology, for example the set of unique identifiers $\{q_a, q_b, ..., q_d\}$ identifying parts $\{qa, qb, ..., qd\}$ of sort Q, **may mean** that parts p and $\{qa, qb, ..., qd\}$ may wish to exchange – for example, attribute – values, one way (from p to the qs) or the other (vice versa) or in both directions. The left fragment of 94

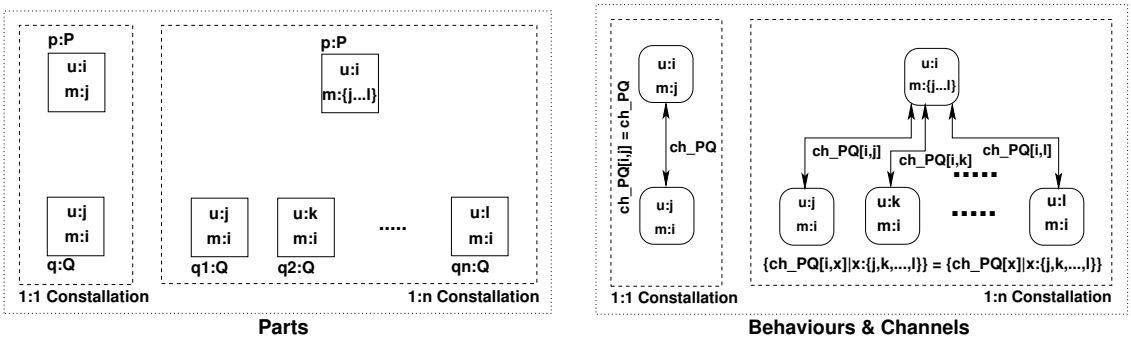


Figure 3: Two Part and Behaviour/Channel Constellations: *u:p* unique id. *p*; *m:p* mereology *p*

the figure intends to show a `1:1 Constellation` 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 Constellation` 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
95 have ascribed channel message types MPQ to all channels.[11] Figure 4 shows an arrangement similar to that of Fig. 3, but for an `m:n Constellation`.

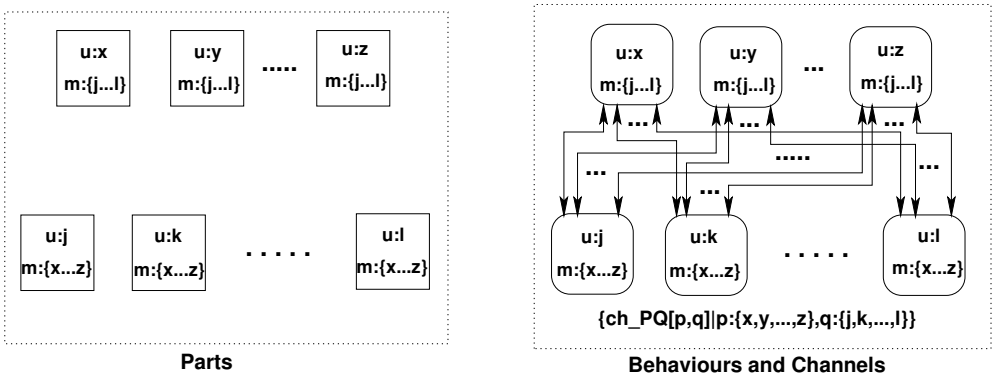


Figure 4: Multiple Part and Behaviour/Channel Constellations: *u:p* unique id. *p*; *m:p* mereology *p*

96 The channel declarations corresponding to Figs. 3 and 4 are:

```
    channel
[1]     ch_PQ[i,j]:MPQ
[2]     { ch_PQ[i,x]:MPQ | x:{j,k,...,l} }
```

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

[3] { ch_PQ[p,q]:MPQ | p:{x,y,...,z}, q:{j,k,...,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]:

channel
[1] ch_PQ:MPQ
[2] { ch_PQ[x]:MPQ | x:{j,k,...,l} }

97

7 The following description identities holds:

7 { ch_PQ[x]:MPQ | x:{j,k,...,l} } ≡ ch_PQ[j],ch_PQ[k],...,ch_PQ[l],

7 { ch_PQ[p,q]:MPQ | p:{x,y,...,z}, q:{j,k,...,l} } ≡
7 ch_PQ[x,j],ch_PQ[x,k],...,ch_PQ[x,l],
7 ch_PQ[y,j],ch_PQ[y,k],...,ch_PQ[y,l],
7 ...,
7 ch_PQ[z,j],ch_PQ[z,k],...,ch_PQ[z,l]

4.4 Behaviours

98

4.4.1 Behaviour Signatures – see Appendix A.4.3 Pg. 48

We associate with each part, p:P, a behaviour name $\mathcal{M}_P$. Behaviours have as first argument their unique part identifier: **uid**_P(p). Behaviours evolves around a state, or, rather, a set of values: its possibly changing mereology, mt:MT and the attributes of the part.[12] A behaviour signature is therefore: 99

$\mathcal{M}_P$: ui:UI×me:MT×stat_attr_typs(p) → ctrl_attr_typs(p) → calc_i_o_chn_refs(p) **Unit**

where (i) ui:UI is the unique identifier value and type of part p; (ii) me:MT is the value and type mereology of part p, me = **obs_mereo**_P(p); (iii) stat_attr_typs(p): *static attribute* types of part p:P; (iv) ctrl_attr_typs(p): *controllable attribute* types of part p:P; (v) calc_i_o_chn_refs(p) calculates channel references to the **in**put channels reflecting the *monitorable attributes* of p and the **in**put/**out**put and the **out**put channels designated in the mereology, me, of p.

4.4.2 Behaviour Definitions – see Appendix A.4.4 Pg. 49

100

Let P be a composite sort defined in terms of endurant[13] sub-sorts E_1, E_2, ..., 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 $\beta_1, \beta_2, \ldots, \beta_n$ 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. 101

[12] We leave out consideration of possible components and materials of the part.
[13] – structures or composite

Behaviour Schema 1

Abstract `is_composite(p)`

```
value
  Translate_P: P → RSL+Text
  Translate_P(p) ≡
    let ui = uid_P(p), me = obs_mereo_P(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
        M_P: P_UI × MT × ST  CT  IOR  Unit
        M_P(ui,me,sta)(pa) ≡ B_P(ui,me,sta)ca
          ,⫸ Translate_P1(obs_endurant_sorts_E1(p))
          ⫷⫸ Translate_P2(obs_endurant_sorts_E2(p))
          ⫷⫸ ...
          ⫷⫸ Translate_Pn(obs_endurant_sorts_En(p))
    end
```

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 ≡. Endurant sorts E_1, E_2, ..., E_n are obtained from the `observe_endurant_sorts` prompt, Page 13. We informally explain the **Translate**$_{P_i}$ function. It takes endurants and produces RSL$^+$Text. Resulting texts are bracketed: ⫷rsl_text⫸ For the case that an endurant is a structure there is only its elements to compile; otherwise Schema 2 is as Schema 1.

Behaviour Schema 2

Abstract `is_structure(e)`

```
value
  Translate_P(p) ≡
              Translate_P1(obs_endurant_sorts_P1(p))
    ⫷⫸ Translate_P2(obs_endurant_sorts_P2(p))
    ⫷⫸ ...
    ⫷⫸ Translate_Pn(obs_endurant_sorts_Pn(p))
```

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 paral-

lel with processes q:**obs_part_**Qs(p). The domain description "compilation" schematic below "formalises" the above. 105

Behaviour Schema 3

Concrete `is_composite(p)`

```
type
  Qs = Q-set
value
  qs:Q-set = obs_part_Qs(p)
  Translate_P(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
        M_P: P_UI×MT×ST  CT  IOR  Unit
        M_P(ui,me,sa)ca ≡ B_P(ui,me,sa)ca  ⫸
        { ⫷,⫸ Translate_Q(q)|q:Q•q ∈ qs }
    end
```

106

Behaviour Schema 4

Atomic `is_atomic(p)`

```
value
  Translate_P(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
        M_P: P_UI×MT×ST  PT  IOR  Unit
        M_P(ui,me,sa)ca ≡ B_P(ui,me,sa)ca  ⫸
    end
```

107

Behaviour Schema 5

Core Process

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

$\mathcal{B}_P$: uid:P_UI×me:MT×sa:SA
→ ct:CT
→ **in** in_chns(p) **in**,**out** in_out_chns(me) **Unit**
$\mathcal{B}_P$(p)(ui,me,sa)(ca) ≡ **let** (me′,ca′) = $\mathcal{F}_P$(ui,me,sa)ca **in** $\mathcal{M}_P$(ui,me′,sa)ca′ **end**

$\mathcal{F}_P$: P_UI×MT×ST → CT→ in_out_chns(me) → MT×CT

4.5 Initial Running Systems – see Appendix A.4.5 Pg. 51 108

To round it all off a narrative and a formalisation must be done of *"a running system"*. Up till now the behaviours for all relevant parts have been defined. Now a decision must be made as to which of these are the basis for an initial system. There may be several candidates for initial running systems, that is, collection of concurrently operating behaviours. So the domain analyser cum describer selects all or some candidates. For each the chosen behaviours are properly initialised. And that is that!

5 A Coin Has Two Sides 109

The transcendental deduction that "turns" parts into behaviours can also be interpreted as follows: The part and the "corresponding" behaviour "exist" at one and the same time: the part is characterised by its *internal qualities*, and these are the arguments, in one form or another of the behaviour. The properties of the internal qualities of parts, expressed, for example, in the form of **axiom**s, hold for all times (a concept not present in the treatment of endurants), and are to be maintained by the corresponding behaviours, as expressed, for
110 example, in **pre/post** conditions. Let us recall essential "features" of parts and behaviours. For parts, $p{:}P$, we can generally express the following:

Pg. 14:	**uid**_P: P → PI	
Pg. 17:	**obs_mereo**_P: P → $\mathcal{E}$(PI$_1$,PI$_2$,...,PI$_m$)	
Pg. 18:	**attr**_sA$_1$: P → sA$_1$	is_static_attribute
	...	is_static_attribute
	attr_sA$_n$: P → sA$_{n_s}$	is_static_attribute
	attr_cA$_1$: P → cA$_1$	is_controllable_attribute
	...	is_controllable_attribute
	attr_cA$_{n_c}$: P → cA$_{n_c}$	is_controllable_attribute
	attr_mA$_1$: P → mA$_1$	is_monitorable_attribute
	...	is_controllable_attribute
	attr_mA$_{n_m}$: P → mA$_{n_m}$	is_monitorable_attribute

111 where $n_s \geq 0$, $n_c \geq 0$, and $n_m \geq 0$. For "corresponding" behaviours, $\mathcal{M}_P$, we have (cf. Process Schema 1 on Page 26):

```
let ui = uid_P(p), me = obs_mereo_P(p),
  sv = stat_attr_vals(p), cv = 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
    M_P: ui:P_UI × me:MT × sv:ST  cv:CT  IOR  Unit
    M_P(ui,me,sv)(cv) ≡ B_P(ui,me,sv)(cv)  ⫸
end
```

We leave it to the reader to study these two sets of formulas. 112

Segment II: Space and Time

We have separated out a treatment of the notions of space and time as these are at the very basis of our ability to describe "the world". That is, has deep implications for our attempt to relate the mundane activity of analysing & describing domains to the philosophical issue of *"what can be described"*.

6 Space Time 113

The presentation of the domain analysis & description calculi avoided, in principle, references to space and time; but these concepts are there: "buried" as follows: endurants can be said to "exist" in space and perdurants to "exist" in time. We shall briefly examine these two concepts as they have been the concern of mathematicians. We shall not be interested in the physicists' *spacetime* mathematical model that fuses the three dimensions of space and the one dimension of time into a single four-dimensional continuum.

6.1 Space 114

> *Space is the boundless three-dimensional extent in which objects and events have relative position and direction*[14]*. Physical space is often conceived in three linear dimensions, although modern physicists usually consider it, with time, to be part of a boundless four-dimensional continuum known as spacetime. The concept of space is considered to be of fundamental importance to an understanding of the physical universe. However, disagreement continues between philosophers over whether it is itself an entity, a relationship between entities, or part of a conceptual framework*[15]*.*

To us *space* is a conceptual framework. That is, it is not an entity, hence neither an endurant or a perdurant. Here we shall primarily look at space as a mathematical construction. In Sect. 9 we shall widen that consideration considerably.

[14] https://www.britannica.com/science/space-physics-and-metaphysics
[15] https://en.wikipedia.org/wiki/Space

6.1.1 Topological Space 115

One notion of space, in mathematics, is that of a Hausdorf (or topological) space:

Definition 23 *Topological Space: A* **topological space** *is an ordered pair (X, τ), where X is a set and τ is a collection of subsets of X, satisfying the following axioms:*[16]

- *The empty set and X itself belong to τ.*
- *Any (finite or infinite) union of members ofτ still belongs to τ.*
- *The intersection of any finite number of members of τ still belongs to τ* ■

The elements of τ are called **open set**s *and the collection τ is called a* **topology** *on X.*

6.1.2 Metric Space 116

A metric spaces 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.

Definition 24 *Metric Space: A* **metric space** *is an ordered pair (M, d) where M is a set and d is a metric on M, i.e., a function*

- $d : M \times M \to \mathbb{R}$

such that for any $x, y, z : M$, the following holds:[17]

- *1.* $d(x, y) \geq 0$ *non-negativity or separation axiom*
- *2.* $d(x, y) = 0 \Leftrightarrow x = y$ *identity of indiscernibles*
- *3.* $d(x, y) = d(y, x)$ *symmetry*
- *4.* $d(x, z) \leq d(x, y) + d(y, z)$ *subadditivity or triangle inequality* ■

6.1.3 Euclidian Space 117

The notion of *Euclidian Space* is due to *Euclid of Alexandria* [325–265]. Euclid postulated

Example 2 **Euclid's Postulates**:

- To draw a straight line from any point to any point.
- To produce [extend] a finite straight line continuously in a straight line.
- To describe a circle with any centre and distance [radius].

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

[17] B. Choudhary (1992). The Elements of Complex Analysis. New Age International. p.20. ISBN 978-81-224-0399-2.

- That all right angles are equal to one another.
- [The parallel postulate] That, if a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which are the angles less than the two right angles ■

We refer to Euclidean space. Encyclopedia of Mathematics. URL: http://www.encyclopediaofmath.org/index.php?title=Euclidean_space&oldid=38673 The European Mathematical Society and Springer. 118

Example 3 **Euclid's Plane Geometry**: The Euclidean geometry informally described in Example 2 can be formally axiomatised by first introducing the sorts P and L:

type
  P, L
value
  [0] obs_Ps: L → P-**infset**
    parallel: L × L → **Bool**

Observe how the informal axiom in Example 2 has been modeled by the *observer function* obs_Ps. It applies to lines and yields possibly infinite sets of points. 119

Now we can introduce the axioms proper:

axiom
  [1] ∃ p,q:P • p ≠ q,
  [2] ∀ p,q:P • p ≠ q ⇒
      ∃! l:L • p ∈ obs_Ps(l) ∧ q ∈ obs_Ps(l),
  [3] ∀ l:L • ∃ p:P • p ∉ obs_Ps(l),
  [4] ∀ l:L • ∃ p:P • p ∉ obs_Ps(l) ⇒
      ∃ l′:L • l≠l′ ∧ p ∈ obs_Ps(l′) ∧ parallel(l,l′)

The concept of being parallel is modeled by the predicate symbol of the same name, by its signature and by axiom [4] ■

We leave it to the reader to reconcile the models of topological space, Defn. 23 on the preceding page, and metric space, Defn. 24 on the facing page, with the axiom systems of examples 2 on the preceding page and 3.

6.2 Time 120

```
                                (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.
```

[24, (i) Plato, (ii) Aristotle, (iii) Plotinus, (iv) Augustine].

6.2.1 Time — General Issues 121

In the next sections we shall focus on various models of time, and we shall conclude with a simple view of the operations we shall assume when claiming that an abstract type models time. These sections are far from complete. They are necessary, but, as a general treatment of notions of time, they are not sufficient. We refer the interested reader to special monographs: [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37].

6.2.2 "A-Series" and "B-Series" Models of Time 122

Colloquially, in ordinary, everyday parlance, we think of time as a dense series of time points. We often illustrate time by a usually horizontal line with an arrow pointing towards the right. Sometimes that line arrowhead is labeled with either a t or the word *time*, or some such name. J.M.E. McTaggart (1908, [29, 28, 37]) discussed theories of time around two notions:

- **"A-series":** has terms like "past", "present" and "future".
- **"B-series":** has terms like "precede", "simultaneous" and "follow".

123

McTaggart argued that the B-series presupposes the A-series: If t precedes t' then there must be a "thing" t'' at which t is past and t' is present. He argued that the A-series is incoherent: What was once 'future', becomes 'present' and then 'past'; and thus events 'will be events', 'are events' and 'were events', that is, will have all three properties.

6.2.3 A Continuum Theory of Time 124

The following is taken from Johan van Benthem [27]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering ($<$, $=$, $>$) relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the
125 application at hand, while also fitting our intuition.

Time is transitive: If $p<p'$ and $p'<p''$ then $p<p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time.
The following is taken from Johan van Benthem [27]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering ($<$, $=$, $>$) relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the
126 application at hand, while also fitting our intuition.

Time is transitive: If $p<p'$ and $p'<p''$ then $p<p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time.
One could designate a time axis as beginning at some time, that is, having no predecessor

times. And one can designate a time axis as ending at some time, that is, having no successor times. General, past and future successors (predecessors, respectively successors in daily talk) can be defined. Time can be dense: Given any two times one can always find a time between them. Discrete time can be defined. 127

axiom
[TRANS: Transitivity] ∀ p,p′,p″:P • p < p′ < p″ ⇒ p < p″
[IRREF: Irreflexitivity] ∀ p:P • p ≮ p
[LIN: Linearity] ∀ p,p′:P • (p=p′ ∨ p<p′ ∨ p>p′)
[L−LIN: Left Linearity] ∀ p,p′,p″:P • (p′<p ∧ p″<p) ⇒ (p′<p″ ∨ p′=p″ ∨ p″<p′)
[BEG: Beginning] ∃ p:P • ∼∃ p′:P • p′<p
[END: Ending] ∃ p:P • ∼∃ p′:P • p<p′
[SUCC: Successor]
[PAST: Predecessors] ∀ p:P,∃ p′:P • p′<p
[FUTURE: Successor] ∀ p:P,∃ p′:P • p<p′
[DENS: Dense] ∀ p,p′:P (p<p′ ⇒ ∃ p″:P • p<p″<p′)
[DENS: Converse Dense] ≡ [TRANS: Transitivity]
[DISC: Discrete]
∀ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p<p″ ∧ ∼∃ p‴:P • (p<p‴<p″))) ∧
∀ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p″<p′ ∧ ∼∃ p‴:P • (p″<p‴<p′)))

A strict partial order, SPO, is a point structure satisfying TRANS and IRREF. TRANS, IRREF and SUCC imply infinite models. TRANS and SUCC may have finite, "looping time" models.

6.3 Wayne D. Blizard's Theory of Space–Time 128

We now bring space and time together in an axiom system (Wayne D. Blizard, 1980 [38]) which relate abstracted entities to spatial points and time. Let $A, B, \ldots$ stand for entitites, $p, q, \ldots$ for spatial points, and t, τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p,q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator applicable, pairwise to entities, spatial locations and times, respectively. A^t_p expresses that entity A is at location p at time t. The axioms — where we omit (obvious) typings (of A, B, P, Q, and T): $'$ designates the time successsor function: t'. 129

(I)		$\forall A \forall t \exists p$	:	A^t_p	
(II)		$(A^t_p \wedge A^t_q)$	$\supset$	$p = q$	
(III)		$(A^t_p \wedge B^t_p)$	$\supset$	$A = B$	
$(IV)(?)$		$(A^t_p \wedge A^{t'}_p)$	$\supset$	$t = t'$	
$(V$	$i)$	$\forall p, q$	:	$N(p,q) \supset p \neq q$	Irreflexivity
$(V$	$ii)$	$\forall p, q$	:	$N(p,q) = N(q,p)$	Symmetry
$(V$	$iii)$	$\forall p \exists q, r$	:	$N(p,q) \wedge N(p,r) \wedge q \neq r$	No isolated locations 130
$(VI$	$i)$	$\forall t$	:	$t \neq t'$	
$(VI$	$ii)$	$\forall t$	:	$t' \neq 0$	
$(VI$	$iii)$	$\forall t$	:	$t \neq 0 \supset \exists \tau : t = \tau'$	
$(VI$	$iv)$	$\forall t, \tau$	:	$\tau' = t' \supset \tau = t$	
(VII)		$A^t_p \wedge A^{t'}_q$	$\supset$	$N(p,q)$	
$(VIII)$		$A^t_p \wedge B^t_q \wedge N(p,q)$	$\supset$	$\sim (A^{t'}_q \wedge B^{t'}_p)$	

We comment on these axioms:

- II–IV,VII–VIII: The axioms are universally 'closed'; that is: We have omitted the usual $\forall A, B, p, q, t$s.
- (I): For every entity, A, and every time, t, there is a location, p, at which A is located at time t.
- (II): An entity cannot be in two locations at the same time.
- (III): Two distinct entities cannot be at the same location at the same time.
- (IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture: Could be questioned.*
- (V): These three axioms define N.
- (V i): Same as $\forall p :\sim N(p, p)$. "Being a neighbour of", is the same as "being distinct from".
- (V ii): If p is a neighbour of q, then q is a neighbour of p.
- (V iii): Every location has at least two distinct neighbours.
- (VI): The next four axioms determine the time successor function $'$.
- (VI i): A time is always distinct from its successor: time cannot rest. There are no time fix points.
- (VI ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.
- (VI iii): Every non–begin time has an immediate predecessor.
- (VI iv): The time successor function $'$ is a one–to–one (i.e., a bijection) function.
- (VII): The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time (t'), then p and q are neighbours.
- (VIII): *No "switching":* If entities A and B occupy neighbouring locations at time t them it is not possible for A and B to have switched locations at the next time (t').

Except for Axiom (IV) the system applies both to systems of entities that "sometimes" rests, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities "occupy more" space volume than others, then we may suitably "repair" the notion of the point space P (etc.). We do not show so here.

Segment III: A Philosophy Basis

7 A Task of Philosophy

8 From Ancient to Kantian Philosophy and Beyond !

The review of this section, i.e., Sect. 8, is based primarily on [13]. It is exclusively "slanted" towards those aspects of the thinking of these philosophers with respect to the *task of phi-*

losophy as we define it in Sect. 7. In this review we reject the contributions of these great philosophers that is contradictory. This presentational "bias" should in no way stand in way of our general admiration for their otherwise profound thinking.

8.1 Pre-Socrates

136

A number of pre-Socratian thinkers speculated on how the world was "constructed". The earlier thinkers were pre-occupied with *matter*, that is, *substance*; what did the world consist of, how was it constructed? We briefly review some of the pre-Socratian thinkers. **Thales of Miletus, 624–546 BC** claimed that all existing, i.e., *base matter*, derived from *water*; **Anaximander of Miletus, 610–546 BC** that *base matter* all came from *apeiron*, some further unspecified substance; **Anaximenes of Miletus, 585–528 BC** that *base matter* was *air*; **Heraklit of Efesos, a. 500 BC** claimed that *fire* was the *base matter*, extended the concern from *substance* to *permanence* and based the thinking not only on (*empirical*) *observations* but also on *logical reasoning* claiming that everything in the world was in a constant struggle, all the time changing – so since all is *changing*, i.e., that nothing is *stable*, he concludes that *nothing exists*; **Parmenides of Elea, 501–470 BC** counterclaimed that that which actually exists is *eternal* and *unchanging* – is logically impossible; **Zeno of Elea, 490–430 BC** supported Parmenindes' claim by claiming some paradox, i.e., the well-known Achilles and the tortoise – thereby introducing *dialectic reasoning* and *proof by contradiction* (*reductio ad absurdum*) **Demokrit, 460–370 BC** tried to unify Heraklit's concept of *changeability* and Parmenides' concept of *permanence* in a new way; everything in the world is built from, consists of *atoms* and change is due to movement of atoms. **The Sophists, 5th Century BC** doubted, or even refuted, that we can arrive at universal truths about the world purely through reasoning. They refute that there is an objectively true reality which we can obtain knowledge about. So, instead, skepticism reigned. What is interesting, to us, is that, this thinking delineates the realms of religion and mythology on one side, and as well as those of science and philosophy, on the other side.

137 138 139 140 141

8.2 Plato, Socrates and Aristotle

142

Socrates, 470–399 BC protested against the sophists' refusal of reason, common sense, sanity and prudence. We know of Socrates' thinking almost exclusively through **Plato, 427–347 BC**: We shall focus on Plato's *theory of ideas*. His argument is that non-physical (but substantial) *ideas* represent the most accurate reality. Abstract and common concepts obtain meaning through standing for ideas that are eternal and unchangeable. In contrast to *ideas* Plato considers the concept of a *phenomenon*. *Phenomena are instances of ideas. We recognize a phenomenon because it embodies an idea.* So, according to Plato, the changeable world that surrounds us, one which we experience through our senses, is only a reflection of a, or the, real world. That real world is unchangeable and "consists" of ideas. **Aristotle, 384–322 BC**. We shall look at several of the concept clusters that Aristotle made to our thinking of the world: (i) *modalities* and (ii) *explanations* – the latter also referred to as *causes*. The *modalities* are: (i.1) *necessity*, (i.2) *reality*, and (i.3) *possibility*. A few words on the *modalities*: (i.1) *necessity*:

143 144 145 146

(i.2) *reality*, and

MORE TO COME

(i.3) *possibility*.

MORE TO COME

147 The *causes* (or *explanations*) are: (ii.1) *matter* or *material cause*, (ii.2) *form cause*, (ii.3) *agent*
148 *cause* and (ii.4) *end cause or purpose cause*. (ii.1) By *material cause* Aristotle means the aspect of the change or movement which is determined by the material that composes the moving or changing things. (ii.2) By *form cause* Aristotle means a change or movement's *formal cause*, is a change or movement caused by the arrangement, shape or appearance of the thing changing
149 or moving. (ii.3) By *agent cause* Aristotle means a change or movement's efficient or moving cause, consists of things apart from the thing being changed or moved, which interact so as to be an agency of the change or movement. (ii.4) By *end cause* Aristotle means a change or movement's final cause, is that for the sake of which a thing is what it is.

8.3 The Stoics: 300 BC–200 AD 150

Chrysippus of Soli: 279–206 BC

8.4 The Rational Tradition: Descartes, 151

René Descartes: 1596–1650 rejected the splitting of *corporeal substance* into *matter* and *form*. His main focus was on the relations between *mind* and *form*: as thinking substance we recognize material substance. **Baruch Spinoza: 1632–1677** rejected Descartes's two substances: there is, he claims, is only one substance; for Spinoza God and nature was one and
152 the same. **Gottfried Wilhelm Leibniz: 1646–1716** introduced the *Law of the Indiscernability of Identicals*, It is still in wide use today. It states that if some object x is identical to some object y, then any property that x has, y will have as well.

8.5 The Empirical Tradition: Locke, Berkeley and Hume 153

John Locke: 1632–1704. We focus on Locke's ideas of *sensing*. He defines himself[18]:

> *as that conscious thinking thing, (whatever substance, made up of whether spiritual, or material, simple, or compounded, it matters not) which is sensible, or conscious of pleasure and pain, capable of happiness or misery, and so is concerned for itself, as far as that consciousness extends.*

154 According to Locke, humans obtain their knowledge about the world through sensory perception. At one level, he claims, the world is "mechanical", so our sensory apparatus is influenced mechanically, for example through tactile or visual means. This sense information is then communicated to our brains. First the mechanical sense data a become *sense ideas*, The sense ideas then become *reflection ideas*. In the "jargon" of our domain analysis & description
155 the *sense ideas* are *values* and the *reflection ideas* become *types*. So a central idea in Locke's

[18] Locke, John (1997), Woolhouse, Roger, ed., *An Essay Concerning Human Understanding*, New York: Penguin Books

theory is that all *cognition* builds on our *reflection* over *sense ideas*. In other words: *Can we conclude anything from our sense ideas to knowledge about those "outer" things which causes the sense ideas ?* [16, pg. 85] To answer that question Locke goes on to distinguish[19] between *primary qualities*[20] and *secondary qualities*[21]. In the jargon of domain analysis & description the 156
primary qualities correspond to "our" *external qualities*, the secondary qualities to "our" *internal qualities*, but not quite! Locke views primary qualities as measurable aspects of physical reality and secondary qualities as subjective aspects of physical reality, where "our" domain analysis & description takes both to be somehow measurable. We must therefore claim that our distinction is purely pragmatic. Locke now claims: (i) that we can, with respect to the 157
primary qualities, deduce from our sense ideas to the reality, the world behind these; (ii) that the primary qualities exist in reality independent of whether we "experience" them or not; and (iii) that this is not the case for the secondary qualities which exist only in our *consciousness*. **George Berkeley: 1685–1753** points out a problem in Locke's theory: namely that 158
Locke's distinction between primary qualities as being *objective* and secondary qualities as being *subjective* does not hold. He argues that primary qualities can be subjective. To solve 159
that problem Berkeley denied the existence of a reality "behind" the sense ideas: there is no material reality; reality is our sense ideas: *esse est precipi*[22] ! The material reality is there because it is continuously experience by 'God'. The problem now is can we. at all, determine fundamental characteristics about the world and our situation as humans in that world without assuming the concept of independently existing substance. **David Hume, 1711–1776**. 160
Where Berkeley eliminated *material substance* Hume also eliminated Berkeley's concepts of 'God' and 'Consciousness'. He claimed that the basic sense-impressions, which to Hume were the basis for all valid human recognition, made it impossible to arrive at a valid recognition of 'God' and a substantial 'I'. They must therefore be eliminated when trying to describe the world and our situation in it. According to Hume all that we know are *sense impressions* and 161
the *conceptions* derived from these. Hume further distinguishes between *composite* *simple* (not-composite) *sense impressions*. Correspondingly Hume distinguishes between *composite* *simple* (not-composite) *ideas*. As a consequence there is no *necessity* in the world, nor in 162
possible relations between *cause* and *effect* This renders Hume's thinking in this area very problematic.

8.6 Immanuel Kant: 1720–1804 163

8.7 Post-Kant 164

Johann Gottlieb Fichte, 1752–1824 tried to avoid Kant's *Das Ding an sich*/*Das Ding für uns* dualism by letting the subject, the I, determine the object, the not-I, but ends up in contradiction. **Georg Wilhelm Friedrich Hegel, 1770–1831** also dissolves the Kantian dualism. He

[19] https://en.wikipedia.org/wiki/Primary/secondary_quality_distinction

[20] Primary qualities are thought to be properties of objects that are independent of any observer, such as solidity, extension, motion, number and figure. These characteristics convey facts. They exist in the thing itself, can be determined with certainty, and do not rely on subjective judgments. For example, if an object is spherical, no one can reasonably argue that it is triangular.

[21] Secondary qualities are thought to be properties that produce sensations in observers, such as color, taste, smell, and sound. They can be described as the effect things have on certain people. Knowledge that comes from secondary qualities does not provide objective facts about things.

[22] *"to-be-is-to-be-perceived"*

builds an impressive theory. The basis for this theory is the assumption of a deep-seated identity between *reason* (*sense*) and *reality*. : *"the reasonable is real"* and *"the real is reasonable"*. Hegel saw his understanding of this duality in the light of *history*. Hegel saw truth, reason and reality historically. "Modern" dialectism was born. Now two contradictory philosophies could now be both true. Philosophy, according to Hegel, had to be seen in the light of history. From this Hegel developed an impressive "apparatus": From "nothingness" via "creation", "quality", quantity" to "essence", "cause", "reality", "causality", and on to "concept", "life" and "cognition" ending with the "absolute" ! And there we end ! We must reject Hegel's *thesis, antithesis, synthesis*. By relativising philosphy wrt. history Hegel has removed necessity. By thus postulating that *"it is an eternal truth that we cannot achieve eternal truths"*. Hegel's main contribution ends up in contradiction. **Friedrich Schelling, 1775–1854** goes further by removing the *subject/object* distinction claiming an underlying identity between these, that is, between mind and matter: nature is the visible mind, and mind is the invisible nature. Again this attempt brings Schelling's work into contradictions. **Friedrich Nitzsche, 1844–1900 Edmund Husserl, 1859–1938 Bertrand Russel, 1872–1970** amongst very many contributions put forward a *Philosophy of Logical Atomism [39]*. **Martin Heidegger, 1889–1976**

TO BE WRITTEN

Ludwig Wittgenstein, 1889–1951

• • •

We have surveyed ideas of 28 philosophers – ideas relevant to our quest:

MORE TO COME

Segment IV: Summing Up

12 Conclusion

180

TO BE WRITTEN

13 Bibliography

181

13.1 Bibliographical Notes

We list a number of reports all of which document descriptions of domains. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

182

1 *A Railway Systems Domain:* `racosy/domains.ps` (2003)

2 *Models of IT Security:* `it-security.pdf` (2006)

3 *A Container Line Industry Domain:* `container-paper.pdf` (2007)

4 *The "Market": Buyers, Sellers, Traders:* `themarket.pdf` (2007)

5 *What is Logistics ?:* `logistics.pdf` (2009)

6 *A Domain Model of Oil Pipelines:* `pipeline.pdf` (2009)

7 *Transport Systems:* `comet/comet1.pdf` (2010)

8 *The Tokyo Stock Exchange:* `todai/tse-1.pdf` and `todai/tse-2.pdf` (2010)

9 *On Development of Web-based Software:* `wfdftp.pdf` (2010)

10 *A Credit Card System:* `/2016/uppsala/accs.pdf` (2016)

11 *Documents:* `/2017/docs.pdf` (2017)

12 *A Context for Swarms of Drones:* `/2016/uppsala/accs.pdf` (2017)

13 *A Framework for Urban Planning:* `/2018/accs.pdf` (2018)
http://www.imm.dtu.dk/ dibj/2017/urban-planning.pdf

13.1 References 183

[1] Dines Bjørner. A Domain Analysis & Description Method – Principles, Techniques and Modeling Languages. Research Note based on [2], February 20 2018. http://www.imm.dtu.-dk/~dibj/2018/adam/2018daad.pdf.

[2] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, Online: July 2016, Journal: March 2017. DOI 10.1007/s00165-016-0385-z http://link.springer.com/article/10.1007/s00165-016-0385-z.

[3] Dines Bjørner. Domain Facets: Analysis & Description. 2016. Extensive revision of [4]. http://www.imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf.

[4] 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.

[5] Dines Bjørner. Domain Analysis and Description – Formal Models of Processes and Prompts. 2016. Extensive revision of [6]. http://www.imm.dtu.dk/~dibj/2016/process/process-p.pdf.

[6] Dines Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[7] Dines Bjørner. To Every Manifest Domain a CSP Expression — A Rôle for Mereology in Computer Science. *Journal of Logical and Algebraic Methods in Programming*, (94):91–108, January 2018.

[8] 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.

[9] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. 2016. Extensive revision of [10].

[10] 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.

[11] Dines Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. Technical report, Fredsvej 11, DK–2840 Holte, Denmark, 2016. Extensive revision of [12]. http://www.imm.dtu.dk/~dibj/2016/demos/faoc-demo.pdf.

[12] 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.

[13] Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, 1994. 168 pages.

[14] Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, 1997. 200 pages.

[15] Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, 2002. 187 pages.

[16] Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, 2016. 233 pages.

[17] C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.-com/cspbook.pdf (2004).

[18] 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.

[19] Edward N. Zalta. The Stanford Encyclopedia of Philosophy. 2016. Principal Editor: https://plato.stanford.edu/.

[20] R. Casati and A. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[21] 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.

[22] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[23] Ted Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.

[24] Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.

[25] 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.

[26] George Wilson and Samuel Shpall. Action. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[27] Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

[28] 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.

[29] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [30].

[30] Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[31] Arthur Prior. *Changes in Events and Changes in Things*, chapter in [30]. Oxford University Press, 1993.

[32] Arthur N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.

[33] Arthur N. Prior. *Formal Logic*. Clarendon Press, Oxford, UK, 1955.

[34] Arthur N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.

[35] Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.

[36] Arthur N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.

[37] 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.

[38] Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

[39] Bertrand Russell. The Philosophy of Logical Atomism. *The Monist: An International Quarterly Journal of General Philosophical Inquiry,*, xxxviii–xxix:495–527, 32–63, 190–222, 345–380, 1918–1919.

[40] 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.

[41] Ludwig Wittgenstein. *Tractatus Logico–Philosophicus*. Taylor & Francis Books Ltd, 1975.

A An Example: A Road Transport System 184

A.1 The Universe of Discourse – see Sect. 2.1 Pg. 9 185

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, (intersections) and links, L, (street segments between intersections); a fleet of automobiles, FA, of automobiles, A; et cetera. See Fig. 5 on the next page

A.2 Endurants – see Sect. 2 Pg. 9 186

A.2.1 Structures – see Sect. 2.4 Pg. 11

See Description Prompt 1, Pg. 12.

8 There is the *universe of discourse*, UoD. It is structured into

9 a *road net*, RN, a structure, and

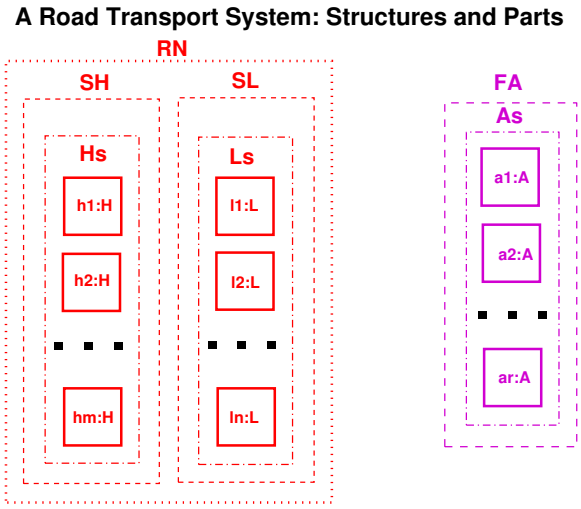


Figure 5: A Road Transport System

10 a *fleet of automobiles*, FA, a structure.

type
8 UoD **axiom** ∀ uod:UoD • is_structure(uod).
9 RN **axiom** ∀ rn:RN • is_strucure(rn).
10 FA **axiom** ∀ fa:FA • is_structure(fa).
value
9 obs_RN: UoD → RN
10 obs_FA: UoD → FA

A.2.2 Parts, Components and Materials – see Sect. 2.5 Pg. 12 187

See Description Prompt 1, Pg. 12.

11 The road net consists of
  a a structure, SH, of hubs and
  b a structure, SL, of links.

12 The fleet of automobiles consists of
  a a set, As of automobiles.

type
11a SH **axiom** ∀ sh:SH • is_structure(sh)
11b SL **axiom** ∀ sl:SL • is_structure(sl)
12a As = A-**set**
value
11a obs_SH: RN → SH
11b obs_SL: RN → SL
12a obs_As: FA → As

A.2.3 Parts – see Sect. 2.5.1 Pg. 12 189

See Description Prompt 2, Pg. 13.

13 The structure of hubs is a set, sH, of atomic hubs, H.

14 The structure of links is a set, sL, of atomic links, L.

15 The structure of automobiles is a set, sA, of atomic automobiles, A.

type
13 H, sH = H-**set** **axiom** ∀ h:H • is_atomic(h)
14 L, sL = L-**set** **axiom** ∀ l:L • is_atomic(l)
15 A, sA = A-**set** **axiom** ∀ a:A • is_atomic(a)
value
13 obs_sH: SH → sH
14 obs_sL: SL → sL
15 obs_sA: SA → sA

A.2.4 Components – see Sect. 2.5.2 Pg. 14

191

See Description Prompt 3, Pg. 14.

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

16 Hubs may contain components, but only if the hub is connected to exactly one link.

17 These "cul-de-sac" hub components may be such things as Sand, Gravel, Cobble Stones, Asphalt, Cement or other.

```
value
16  has_components: H → Bool
type
17  Sand, Gravel, CobbleStones, Asphalt, Cement, ...
17  KS = (Sand|Gravel|CobbleStones|Asphalt|Cement|...)-set
value
16  obs_components_H: H → KS
16  pre: obs_components_H(h) ≡ card mereo(h) = 1
```

A.2.5 Materials – see Sect. 2.5.3 Pg. 15

193

See Description Prompt 4, Pg. 15.

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

18 Links may contain material.

19 That material is water, W.

```
type
19  W
value
18  obs_material: L → W
18  pre: obs_material(l) ≡ has_material(h)
```

A.2.6 States – see Sect. 4.1 Pg. 22

194

20 Let there be given a universe of discourse, rts. It is an example of a state.

From that state we can calculate other states.

21 The set of all hubs, hs.

22 The set of all links, ls.

23 The set of all hubs and links, hls.

24 The set of all automobiles, as.

25 The set of all parts, ps.

```
value
20  rts:UoD
21  hs:H-set ≡ ≡ obs_sH(obs_SH(obs_RN(rts)))
22  ls:L-set ≡ ≡ obs_sL(obs_SL(obs_RN(rts)))
23  hls:(H|L)-set ≡ hs∪ls
24  as:A-set ≡ obs_As(obs_FV(rts))
25  ps:(H|L|BC|B|A)-set ≡ hls∪bcs∪bs∪as
```

A.2.7 Unique Identifiers – see Sect. 2.6 Pg. 15

196

See Description Prompt 5, Pg. 16

Part Identifiers

26 We assign unique identifiers to all parts.

27 By a road identifier we shall mean a link or a hub identifier.

28 Unique identifiers uniquely identify all parts.

- a All hubs have distinct [unique] identifiers.
- b All links have distinct identifiers.
- c All automobiles have distinct identifiers.
- d All parts have distinct identifiers.

```
type
26  H_UI, L_UI,  A_UI
27  R_UI = H_UI | L_UI
value
28a  uid_H: H → H_UI
28b  uid_L: L → L_UI
28c  uid_A: A → A_UI
```

Extract Parts from Their Unique Identifiers

29 From the unique identifier of a part we can retrieve, $\wp$, the part having that identifier.

```
type
29 P = H | L | A
value
29  ℘: H_UI→H | L_UI→L | A_UI→A
29  ℘(ui) ≡ let p:(H|L|A)•p∈ps∧uid_P(p)=ui in p end
```

Unique Identifier Constants: We can calculate:

30 the *s*et, $h_{ui}s$, of *u*nique *h*ub *i*dentifiers;

31 the *s*et, $l_{ui}s$, of *u*nique *l*ink *i*dentifiers;

32 the *m*ap, $hl_{ui}m$, from *u*nique *h*ub *i*dentifiers to the *s*et of *u*nique *l*ink *i*identifiers of the links connected to the zero, one or more identified hubs,

33 the *m*ap, $lh_{ui}m$, from *u*nique *l*ink *i*dentifiers to the *s*et of *u*nique *h*ub *i*identifiers of the two hubs connected to the identified link;

34 the *s*et, $r_{ui}s$, of all *u*nique hub and link, i.e., *r*oad *i*dentifiers;

35 the *s*et, $a_{ui}s$, of *u*nique *a*utomobile *i*dentifiers;

value
30. $h_{ui}s$:H_UI-**set** ≡ {uid_H(h)|h:H•h ∈ hs}
31. $l_{ui}s$:L_UI-**set** ≡ {uid_L(l)|l:L•l ∈ ls}
34. $r_{ui}s$:R_UI-**set** ≡ $h_{ui}s$∪$l_{ui}s$
32. $hl_{ui}m$:(H_UI $\overrightarrow{m}$ L_UI-**set**) ≡
32. [h_ui↦luis|h_ui:H_UI,luis:L_UI-set•h_ui∈$h_{ui}s$
32. ∧(_,luis,_)=mereo_H(η(h_ui))] [cf. Item 40]
33. $lh_{ui}m$:(L+UI $\overrightarrow{m}$ H_UI-**set**) ≡
33. [l_ui↦huis [cf. Item 41]
33. | h_ui:L_UI,huis:H_UI-set • l_ui∈$l_{ui}s$
33. ∧ (_,huis,_)=mereo_L(η(l_ui))]
35. $a_{ui}s$:A_UI-**set** ≡ {uid_A(a)|a:A•a ∈ as}

A.2.8 Uniqueness of Part Identifiers 201

See Sect. A.2.12 Pg. 47. We must express the following axioms:

36 All hub identifiers are distinct.

37 All link identifiers are distinct.

38 All automobile identifiers are distinct.

39 All part identifiers are distinct.

axiom
36 **card** hs = **card** $h_{ui}s$
37 **card** ls = **card** $l_{ui}s$
38 **card** as = **card** $a_{ui}s$
39 **card** {$h_{ui}s$∪$l_{ui}s$∪$a_{ui}s$}
39 = **card** $h_{ui}s$+**card** $l_{ui}s$+**card** $a_{ui}s$

A.2.9 Part Mereologies – see Sect. 2.7 Pg. 16 202

See Description Prompt 6, Pg. 17

40 The mereology of hubs is a triple: (i) the set of all automobile identifiers[23], (ii) the set of unique identifiers of the links that it is connected to and the set of all unique identifiers of all automobiles.[24], and (iii) an empty set.[25]

41 The mereology of links is a triple: (i) the set of all automobile identifiers, (ii) the set of the two distinct hubs they are connected to, and (iii) an empty set.

42 The mereology of an automobiles is a triple: (i) an empty set, (ii) an empty set, and (iii) the set of the unique identifiers of all links and hubs[26].

43 Empty sets are modeled as empty sets of tokens where tokens are further undefined.

type
43 ES = TOKEN-**set**
43 **axiom** ∀ es:ES•es={}
40 H_Mer = V_UI-**set**×L_UI-**set**×ES
40 **axiom** ∀ (vuis,luis,_):H_Mer • luis⊆$l_{ui}s$ ∧ vuis=$v_{ui}s$
41 L_Mer = V_UI-**set**×H_UI-**set**×ES
41 **axiom** ∀ (vuis,huis,_):L_Mer •
41 vuis=$v_{ui}s$ ∧ huis⊆$h_{ui}s$ ∧ **card**huis=2
42 A_Mer = ES×ES×R_UI-**set**
42 **axiom** ∀ (_,ruis,_):A_Mer • ruis=$r_{ui}s$

value
40 mereo_H: H → H_Mer
41 mereo_L: L → L_Mer
42 mereo_A: A → A_Mer

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

44 If hub, h, and link, l, are in the same road net,

45 and if hub h connects to link l then link l connects to hub h.

axiom
44 ∀ h:H,l:L • h ∈ hs ∧ l ∈ ls ⇒
44 **let** (_,luis,_)=mereo_H(h),(_,huis,)=mereo_L(l)
45 **in** uid_L(l) ∈ luis ⇒ uid_H(h) ∈ huis **end**

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

[23] This is just another way of saying that the meaning of hub mereologies involves the unique identifiers of all the automobiles that might pass through the hub `is_of_interest` to it

[24] ... its link identifiers designate the links, zero, one or more, that a hub is connected to `is_of_interest` to both the hub and that these links is `interested` in the hub.

[25] ... the hubs are not "proactive", i.e., that the universe of discourse have no parts that are `interested` in the hub.

[26] that the automobile might pass through

A.2.10 Part Attributes – see Sect. 2.8 Pg. 17

207

We treat part attributes, sort by sort. See Description Prompt 7, Pg. 18

Hubs: We show just a few attributes:

46 There is a hub state. It is a set of pairs, (l_f,l_t) 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_t) is an element, is that the hub is open, "green", for traffic *from* link l_f *to* link l_t. 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.

47 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.

48 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 automobile identifier, of the time ordered presence in the hub of these automobiles.

49 The link identifiers of hub states must be in the set, $l_{ui}s$, of the road net's link identifiers.

```
type
46 HΣ = (L_UI×L_UI)-set                [programmable, Df.8 Pg.20]
axiom
46 ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h)
type
47 HΩ = HΣ-set                         [static, Df.1 Pg.19]
48 H_Traffic                           [programmable, Df.8 Pg.20]
48 H_Traffic = A_UI ~m~> (T × VPos)*
axiom
48 ∀ ht:H_Traffic,ui:A_UI •
48    ui ∈ dom ht ⇒ time_ordered(ht(ui))
value
46 attr_HΣ: H → HΣ
47 attr_HΩ: H → HΩ
48 attr_H_Traffic: : → H_Traffic
axiom
49  ∀ h:H • h ∈ hs ⇒
49    let hσ = attr_HΣ(h) in
49    ∀ (lui,lui′):(L_UI×L_UI) •
49       (lui,lui′) ∈ hσ ⇒ {lui,l′ui} ⊆ luis end
value
48 time_ordered: T* → Bool
48 time_ordered(tvpl) ≡ ...
```

Links: We show just a few attributes:

50 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.

51 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".

52 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 automobile identifier, of the time ordered positions along the link (from one hub to the next) of these automobiles.

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

```
type
50 LΣ = H_UI-set                      [programmable, Df.8 Pg.20]
axiom
50 ∀ lσ:LΣ•card lσ=2
50 ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
type
51 LΩ = LΣ-set                        [static, Df.1 Pg.19]
52 L_Traffic                          [programmable, Df.8 Pg.20]
52 L_Traffic = A_UI ~m~> (T×(H_UI×Frac×H_UI))*
52 Frac = Real, axiom frac:Fract • 0<frac<1
value
50 attr_LΣ: L → LΣ
51 attr_LΩ: L → LΩ
52 attr_L_Traffic: : → L_Traffic
axiom
52 ∀ lt:L_Traffic,ui:A_UI•ui ∈ dom ht
52       ⇒ time_ordered(ht(ui))
53 ∀ l:L • l ∈ ls ⇒
53    let lσ = attr_LΣ(l) in
53    ∀ (hui i,hui i′):(H_UI×K_UI) •
53       (hui i,hui i′) ∈ lσ ⇒
53          {hui_i,h′ui_i} ⊆ huis end
```

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

54 Automobiles have a time attribute.

55 Automobiles have static number plate registration numbers.

56 Automobiles have dynamic positions on the road net:

   a either *at a hub* identified by some h_ui,
   b 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
54  𝒯                                          [inert, Df.3 Pg.19]
55  RegNo                                      [static, Df.1 Pg.19]
56  APos  == atHub | onLink [programmable, Df.8 Pg.20]
56a atHub   :: h_ui:H_UI
56b onLink  :: fh_ui:H_UI×l_ui:L_UI×frac:Fract×th_ui:H_UI
56b Fract   = Real, axiom frac:Fract • 0<frac<1
value
54  attr_T: A → 𝒯
55  attr_RegNo: A → RegNo
56  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.

A.2.11 Discussion of Edurants, I 216

In Items 48 Pg. 46 and 52 Pg. 46, 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 automobiles.[27]

A.2.12 Some Axioms and Proof Obligations 217

Examples of axioms are given in Items 36 – 39 Pg. 45, Items 44 – 45 Pg. 45, Item 48, and in Item 52. We shall give an example of a **proof obligation** expressed as a **post** condition, related to the last two of the above axioms, in Items 73g Pg. 50 and 80 Pg. 50

Those proof obligations reflect an aspect of the concept of **transcendental deduction:** that *axioms* over, as here, *internal qualities* of *endurants* via *post conditions* of *perdurants* become *proof obligations !*

A.2.13 Discussion of Endurants, II 219

- We have chosen to model some discrete endurants
  - as structures
  - others as parts (usually composite).
- Those choices are made mostly to illustrate that the domain analysis & description has a choice.
  - If a choice is made to model a discrete endurant as a structure
    - then it entails that the domain analysis & description does not wish to "implement" that discrete endurant as a behaviour separate from its sub-endurants;
  - If the choice is made to model a discrete endurant as a part
    - then it entails that the domain analysis & description wishes to "implement" that discrete endurant as a behaviour separate from its sub-endurants.
- The following discrete endurants which are modeled as structures above, could, instead, if modeled as parts, have the entailed behaviours reflect the following possibilities:
  - *road net*, rn:RN: The road net behaviour could be that of a road net authority charged with building, servicing, operating and maintaining the road net. Building and maintaining the road net could mean the insertion of new or removal of old links or hubs. Operating the road net could mean the gathering of automobile traffic statistics, the setting of hub states (traffic signal monitoring and control), etc.
  - *aggregate of automobiles*, ps:PA: The aggregate of automobiles could be that of one or more *automobile clubs*, etc.

A.3 Transcendentality – see Sect. 3 Pg. 21 221

We refer to Sect. A.3 Defn. 1 Page 22.

Example 4 **A Case of Transcendentality**: We refer to the following example: We can speak of an automobile in at least three *senses:*

- The automobile as it is being maintained, serviced, refueled;
- the automobile as it "speeds" down its route; and
- the automobile as it "appears" (listed) in car registries or advertisements.

The three *senses* are:

- as a part,
- as a behaviour, and
- as an attribute[28] ■

[27] 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.

[28] in this case rather: as a fragment of an attribute

A.4 Perdurants – see Sect. 4 Pg. 22 222

A.4.1 States

Constants: We refer to Sect. A.2.6 Pg. 44, and to App. 4.1 Pg. 22 We assume, as a constant, an arbitrarily selected universe of discourse, *uod*, and calculate from *uod* all its endurants.

value
20 rts:UoD
21 hs:H-**set** ≡:H-**set** ≡ obs_sH(obs_SH(obs_RN(rts)))
22 ls:L-**set** ≡:L-**set** ≡ obs_sL(obs_SL(obs_RN(rts)))
23 hls:(H|L)-**set** ≡ hs∪ls
24 as:A-**set** ≡ obs_As(obs_FV(rts))

Indexed States: We shall

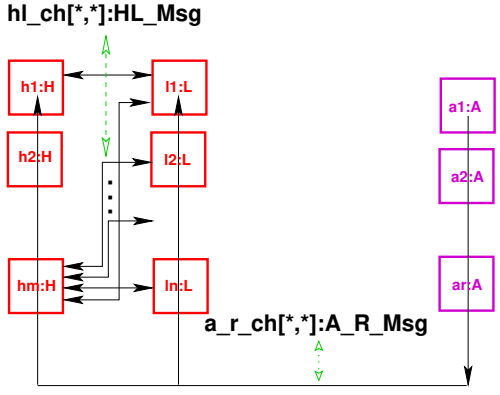
57 index automobiles

using the unique identifiers of these parts.

type
57 A_{ui}
value
57 ias:A_{ui}-**set** ≡
57 {a_{ui}|a:A,a:A_{ui}:A_{ui}•a∈as∧ui=uid_A(a)}

A.4.2 Channels – see Sect. 4.3 Pg. 23 224

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 below:



Channel Message Types: We ascribe types to the messages offered on channels.

58 Hubs and links communicate, both ways, with one another, over channels, `hl_ch`, whose indexes are determined by their mereologies.

59 Hubs send one kind of messages, links another.

60 Automobiles offer their current, timed positions to the road element, hub or link they are on, one way.

type
59 H_L_Msg, L_H_Msg
58 HL_Msg = H_L_Msg | L_F_Msg
60 A_R_Msg = T × APos

Channel Declarations: ...

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

channel
61 { hl_ch[h_ui,l_ui]:H_L_Msg
61 | h_ui:H_UI,l_ui:L_UI•i ∈ $h_{ui}s$∧j ∈ $lh_{ui}m$(h_ui) }
61 ∪
61 { hl_ch[h_ui,l_ui]:L_H_Msg
61 | h_ui:H_UI,l_ui:L_UI•l_ui ∈ $l_{ui}s$∧i ∈ $lh_{ui}m$(l_ui) }

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

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

channel
62 {a_r_ch[a_ui,r_ui]:A_R_Msg
62 |a_ui:A_UI,r_ui:R_UI•a_ui ∈ $a_{ui}s$∧r_ui ∈ $r_{ui}s$}

A.4.3 Behaviour Signatures – see Sect. 4.4.1 Pg. 25 229

We first decide on names of behaviours. In Sect. 4.4.2, Pages 25–27, we gave schematic names to behaviours of the form $\mathcal{M}_P$. We now assign mnemonic names: from part names to names of transcendentally interpreted behaviours and then we assign signatures to these behaviours.

63 $\mathsf{hub}_{h_{ui}}$:

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 automobile behaviours.

A.4.4 Behaviour Definitions – see Sect. 4.4.2

Pg. 25 236

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.

```
value
63  hub_{h_ui}:
63a   h_ui:H_UI×(auis,luis,_):H_Mer×HΩ
63b     → (HΣ×H_Traffic)
63c     → in,out { h_l_ch[ h_ui,l_ui ] | l_ui:L_UI:l_ui ∈ luis }
63d           { a_r_ch[ h_ui,a_ui ] | a_ui:A_UI•a_ui∈auis } Unit
63a     pre: auis = a_uis ∧ luis = l_uis
```

64 $\mathsf{link}_{l_{ui}}$:

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 automobile behaviours.

```
value
64  link_{l_ui}:
64a   l_ui:L_UI×(auis,huis,_):L_Mer×LΩ
64b     → (LΣ×L_Traffic)
64c     → in,out { h_l_ch[ h_ui,l_ui ] | h_ui:H_UI:h_ui ∈ huis }
64d           { a_r_ch[ l_ui,a_ui ] | a_ui:A_UI•a_ui∈auis } Unit
64a     pre: auis = a_uis ∧ huis = h_uis
```

65 $\mathsf{automobile}_{a_{ui}}$:

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: first the input time channel,

d then the input/output allowing communication between the automobile and the hub and link behaviours.

```
value
65  automobile_{a_ui}:
65a   a_ui:A_UI×(_,_,ruis):A_Mer×rn:RegNo
65b     → apos:APos
65c     → in attr_T_ch
65d       in,out {a_r_ch[ a_ui,r_ui ]
65d                | r_ui:(H_UI|L_UI)•r_ui∈ruis} Unit
65a     pre: ruis = r_uis ∧ a_ui ∈ a_uis
```

Automobiles:

66 We abstract automobile behaviour at a Hub (hui).

67 The automobile remains at that hub, "idling",

68 informing the hub behaviour,

69 or, internally non-deterministically,

a moves onto a link, tli, 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 automobile resumes the automobile behaviour positioned at the very beginning (0) of that link,

70 or, again internally non-deterministically,

71 the automobile "disappears — off the radar" !

```
66  automobile_{a_ui}(a_ui,({},(ruis,auis),{}),rn)
66            (apos:atH(fl_ui,h_ui,tl_ui)) ≡
67     (ba_r_ch[ a_ui,h_ui ] ! (attr_T_ch?,atH(fl_ui,h_ui,tl_ui));
68      automobile_{a_ui}(a_ui,({},(ruis,auis),{}),rn)(apos))
69   ⌈⌉
69a    (let ({fh_ui,th_ui},ruis′)=mereo_L(℘(tl_ui)) in
69a          assert: fh_ui=h_ui ∧ ruis=ruis′
66     let onl = (tl_ui,h_ui,0,th_ui) in
69b     (ba_r_ch[ a_ui,h_ui ] ! (attr_T_ch?,onL(onl)) ‖
69b      ba_r_ch[ a_ui,tl_ui ] ! (attr_T_ch?,onL(onl))) ;
69c      automobile_{a_ui}(a_ui,({},(ruis,auis),{}),rn)
69c            (onL(onl)) end end)
70   ⌈⌉
71     stop
```

72 We abstract automobile behaviour on a Link.

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**
   A then the automobile moves an arbitrary small, positive **Real**-valued *increment* along the link
   B informing the hub of this,
   C while resuming being an automobile ate the new position, or

ii **else**,
   A 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 automobile is about to leave),
   B the vehicle informs both the link and the imminent hub that it is now at that hub, identified by th_ui,
   C whereupon the automobile resumes the vehicle behaviour positioned at that hub;

c or

d the automobile "disappears — off the radar" !

```
72  automobile_{a_ui}(a_ui,({},ruis,{}),rno)
72                    (vp:onL(fh_ui,l_ui,f,th_ui)) ≡
72(a)ii   (ba_r_ch[ thui,aui ]!atH(lui,thui,nxt_lui) ;
72(a)i     automobile_{a_ui}(a_ui,({},ruis,{}),rno)(vp))
72b     []
72(b)i    (if not_yet_at_hub(f)
72(b)i      then
72(b)iA       (let incr = increment(f) in
66             let onl = (tl_ui,h_ui,incr,th_ui) in
72(b)iB        a−r_ch[ l_ui,a_ui ] ! onL(onl) ;
72(b)iC        automobile_{a_ui}(a_ui,({},ruis,{}),rno)
72(b)iC                          (onL(onl))
72(b)i         end end)
72(b)ii     else
72(b)iiA      (let nxt_lui:L_UI•nxt_lui ∈ mereo_H(℘(th_ui)) in
72(b)iiB       a_r_ch[ thui,aui ]!atH(l_ui,th_ui,nxt_lui) ;
72(b)iiC       automobile_{a_ui}(a_ui,({},ruis,{}),rno)
72(b)iiC                         (atH(l_ui,th_ui,nxt_lui)) end)
72(b)i    end)
72c     []
72d       stop
72(b)iA   increment: Fract → Fract
```

Hubs: We model the hub behaviour vis-a-vis automobiles.

73 The hub behaviour

   a non-deterministically, externally offers

   b to accept timed automobile positions —

   c which will be at the hub, from some vehicle, v_ui.

   d The timed automobile hub position is appended to the front of that automobile'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 automobile.

   g A **post** condition expresses what is really a **proof obligation**: that the hub traffic, ht′ satisfies the **axiom** of the endurant hub traffic attribute Item 48 Pg. 46.

```
value
73  hub_{h_ui}(h_ui,(,(luis,vuis)),hω)(hσ,ht) ≡
73a     []
73b       { let m = ba_r_ch[ h_ui,v_ui ] ? in
73c             assert: m=(_,atHub(_,h_ui,_))
73d           let ht′ = ht † [ a_ui ↦ ⟨m⟩⌢ht(a_ui) ] in
73e           hub_{h_ui}(h_ui,(,(luis,auis)),(hω))(hσ,ht′)
73f         | a_ui:A_UI•a_ui∈auis end end }
73g     post: ∀ a_ui:A_UI•a_ui ∈ dom ht′
73g             ⇒ time_ordered(ht′(a_ui))
```

Links: Similarly we model the link behaviour vis-a-vis automobiles.

74 The link behaviour non-deterministically, externally offers

75 to accept timed automobile positions —

76 which will be on the link, from some automobile, a_ui.

77 The timed automobile link position is appended to the front of that automobile's entry in the link's traffic table;

78 whereupon the link proceeds as a link behaviour with the updated link traffic table.

79 The link behaviour offers to accept from any automobile.

80 A **post** condition expresses what is really a **proof obligation**: that the link traffic, lt′ satisfies the **axiom** of the endurant link traffic attribute Item 52 Pg. 46.

```
74  link_{l_ui}(l_ui,(_,(huis,auis),_),lω)(lσ,lt) ≡
74    []
75      { let m = ba_r_ch[ l_ui,a_ui ] ? in
76            assert: m=(_,onLink(_,l_ui,_,_))
77          let lt′ = lt † [ a_ui ↦ ⟨m⟩⌢lt(a_ui) ] in
78          link_{l_ui}(l_ui,(huis,auis),hω)(hσ,lt′)
79      | a_ui:A_UI•a_ui∈auis end end }
80    post: ∀ a_ui:A_UI•a_ui ∈ dom lt′
80            ⇒ time_ordered(lt′(a_ui))
```

A.4.5 A Running System – see Sect. 4.5 Pg. 28

246

Preliminaries: We recall the *hub, link* and the *automobile states* first mentioned in Sect. A.2.6 Page 44.

```
value
21  hs:H-set ≡ ≡ obs_sH(obs_SH(obs_RN(rts)))
22  ls:L-set ≡ ≡ obs_sL(obs_SL(obs_RN(rts)))
24  as:A-set ≡ obs_As(obs_FA(rts))
```

Starting Initial Behaviours: We are reaching the end of this domain modeling example. Behind us there are narratives and formalisations 8 Pg. 42 – 80 Pg. 50. Based on these we now express the signature and the body of the definition of a *"system build and execute"* function.

81 The system to be initialised is

a the parallel composition (‖) of

b the distributed parallel composition (‖{...|...}) of

c all the hub behaviours,

d all the link behaviours, and

e all the automobile behaviours.

```
value
81  initial_system: Unit → Unit
81  initial_system() ≡
81c     ‖ { hub_{hui}(h_ui,me,hω)(htrf,hσ)
81c       | h:H•h ∈ hs,
81c         h_ui:H_UI•h_ui=uid_H(h),
81c         me:HMetL•me=mereo_H(h),
81c         hω:HΩ•hω=attr_HΩ(h),
81c         htrf:H_Traffic•htrf=attr_H_Traffic_H(h),
81c         hσ:HΣ•hσ=attr_HΣ(h)∧hσ ∈ hω
81c       }
81a     ‖
81d     ‖ { link_{lui}(l_ui,me,lω)(ltrf,lσ)
81d         l:L•l ∈ ls,
81d         l_ui:L_UI•l_ui=uid_L(l),
81d         me:LMet•me=mereo_L(l),
81d         lω:LΩ•lω=attr_LΩ(l),
81d         ltrf:L_Traffic•ltrf=attr_L_Traffic_H(l),
81d         lσ:LΣ•lσ=attr_LΣ(l)∧lσ ∈ lω
81d       }
81a     ‖
81e     ‖ { automobile_{aui}(a_ui,me,rn)(apos)
81e         a:A•a ∈ as,
81e         a_ui:A_UI•a_ui=uid_A(a),
81e         me:AMet•me=mereo_A(a),
81e         rn:RegNo•rno=attr_RegNo(a),
81e         apos:APos•apos=attr_APos(a)
81e       }
```

A.4.6 The End !

252

Yes, this is the end of the main example.

A.5 Example Index

253

A.5.1 Sorts

A.5.2 Types

A.5.3 Functions

A.5.4 Values

A.5.5 Channels

A.5.6 Behaviours

B RSL: The RAISE Specification Language – A Primer 254

B.1 Type Expressions

Type expressions are expressions whose value are types, that is, possibly infinite sets of values (of "that" type).

B.1.1 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.

type
[1] **Bool** **true**, **false**
[2] **Int** ... , −2, −2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., −5.43, −1.0, 0.0, 1.23· · · , 2,7182· · · , 3,1415· · · , 4.56, ...
[5] **Char** "a", "b", ..., "0", ...
[6] **Text** "abracadabra"

B.1.2 Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully "taken apart". There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types 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 the following are type expressions:

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω
[12] A $\xrightarrow[m]{}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following the meaning of the atomic and the composite type expressions:

1 The Boolean type of truth values **false** and **true**.

2 The integer type on integers ..., –2, –1, 0, 1, 2, ...

3 The natural number type of positive integer values 0, 1, 2, ...

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 × C × ... × 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 $\overrightarrow{m}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.

16 The postulated disjoint union of types A, B, ..., and C.

17 The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

18 The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

type
  A, B, C, ..., D
value
  obs_B: A → B, obs_C: A → C, ..., obs_D: A → D

The above expresses that values of type A are composed from at least three values — and these are of type B, C, ..., and D. A concrete type definition corresponding to the above presupposing material of the next section

type
  B, C, ..., D
  A = B × C × ... × D

B.2 Type Definitions

B.2.1 Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

type
  A = Type_expr

Some schematic type definitions are:

```
[19] Type_name = Type_expr /∗ without | s or subtypes ∗/
[20] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[21] 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)
[22] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[23] Type_name = {| v:Type_name′ • P(v) |}
```

256

where a form of [20]–[21] is provided by combining the 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**

B.2.2 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:

type
A = {| b:B • $\mathcal{P}$(b) |}

B.2.3 Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

type
A, B, ..., C

B.3 The RSL Predicate Calculus

B.4 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

B.4.1 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

B.4.2 Quantified Expressions

Let X, Y, ..., C 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:

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

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.

B.5 Concrete RSL Types: Values and Operations

B.5.1 Arithmetic

type
Nat, **Int**, **Real**

value
  +,−,∗: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\xrightarrow{\sim}$**Nat** | **Int**×**Int**$\xrightarrow{\sim}$**Int** | **Real**×**Real**$\xrightarrow{\sim}$**Real**
  <,≤,=,≠,≥,> (**Nat**|**Int**|**Real**) → (**Nat**|**Int**|**Real**)

B.5.2 Set Expressions

Set Enumerations Let the below a's denote values of type A, then the below designate simple set enumerations:

  {{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ...} ∈ A-**set**
  {{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ..., {e$_1$,e$_2$,...}} ∈ A-**infset**

Set Comprehension The expression, last line below, to the right of the ≡, 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.

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

B.5.3 Cartesian Expressions

Cartesian Enumerations Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e1,e2,...,en)

B.5.4 List Expressions

List Enumerations Let a range over values of type A, then the below expressions are simple list enumerations:

  {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
  {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ A$^\omega$

  ⟨ a_i .. a_j ⟩

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.

```
type
  A, B, P = A → Bool, Q = A ~→ B
value
  comprehend: A^ω × P × Q ~→ B^ω
  comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

B.5.5 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:

```
type
  T1, T2
  M = T1 m~→ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [ ], [ u↦v ], ..., [ u1↦v1,u2↦v2,...,un↦vn ] all ∈ M
```

Map Comprehension The last line below expresses map comprehension:

```
type
  U, V, X, Y
  M = U m~→ V
  F = U ~→ X
  G = V ~→ Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X m~→ Y)
  comprehend(m,F,G,P) ≡ [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

B.5.6 Set Operations

Set Operator Signatures

```
value
  19 ∈: A × A-infset → Bool
  20 ∉: A × A-infset → Bool
  21 ∪: A-infset × A-infset → A-infset
  22 ∪: (A-infset)-infset → A-infset
```

23 $\cap$: A-**infset** $\times$ A-**infset** $\to$ A-**infset**
24 $\cap$: (A-**infset**)-**infset** $\to$ A-**infset**
25 \: A-**infset** $\times$ A-**infset** $\to$ A-**infset**
26 $\subset$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
27 $\subseteq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
28 $=$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
29 $\neq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
30 **card**: A-**infset** $\xrightarrow{\sim}$ **Nat**

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} \ {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}
card {} = 0, **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. 257

25 \: 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 $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.

27 $\subset$: 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):

```
value
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A • a ∈ s in 1 + card (s \ {a}) end end
    pre s /∗ is a finite set ∗/
  card s ≡ chaos /∗ tests for infinity of s ∗/
```

B.5.7 Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × 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
```

B.5.8 List Operations

List Operator Signatures

value
  hd: $A^\omega \xrightarrow{\sim} A$
  tl: $A^\omega \xrightarrow{\sim} A^\omega$
  len: $A^\omega \xrightarrow{\sim}$ **Nat**
  inds: $A^\omega \rightarrow$ **Nat-infset**
  elems: $A^\omega \rightarrow$ **A-infset**
  .(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
  $\frown$: ~~A*~~ ~~A$^\omega$~~ ~~A$^\omega$~~ ~~A$^\omega$~~ ~~A$^\omega$~~ ~~BoBbool~~

List Operation Examples

examples
  hd⟨a1,a2,...,am⟩=a1
  tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
  len⟨a1,a2,...,am⟩=m
  inds⟨a1,a2,...,am⟩={1,2,...,m}
  elems⟨a1,a2,...,am⟩={a1,a2,...,am}
  ⟨a1,a2,...,am⟩(i)=ai
  ⟨a,b,c⟩$\frown$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
  ⟨a,b,c⟩=⟨a,b,c⟩
  ⟨a,b,c⟩ ≠ ⟨a,b,d⟩

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 ith element of the list. 258
- $\frown$: 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.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }
```

259

```
  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩⌢q′ in a end
          else chaos end
      else q(i−1) end

  fq ⌢ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
      | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

B.5.9 Map Operations

Map Operator Signatures and Map Operation Examples

```
value
  m(a): M → A ~→ B, m(a) = b

  dom: M → A-infset [domain of map]
    dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}
```

rng: M → B-**infset** [range of map]
  rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

†: M × M → M [override extension]
  [a↦b,a′↦bb′,a″↦bb″] † [a′↦bb″,a″↦bb′] = [a↦b,a′↦bb″,a″↦bb′]

260

∪: M × M → M [merge ∪]
  [a↦b,a′↦bb′,a″↦bb″] ∪ [a‴↦bb‴] = [a↦b,a′↦bb′,a″↦bb″,a‴↦bb‴]

\: M × A-**infset** → M [restriction by]
  [a↦b,a′↦bb′,a″↦bb″]\{a} = [a′↦bb′,a″↦bb″]

/: M × A-**infset** → M [restriction to]
  [a↦b,a′↦bb′,a″↦bb″]/{a′,a″} = [a′↦bb′,a″↦bb″]

=,≠: M × M → **Bool**

°: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]
  [a↦b,a′↦bb′] ° [bb↦c,bb′↦c′,bb″↦c″] = [a↦c,a′↦c′]

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.
- †: 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.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: 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.

261

- /: 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.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: 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:

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ bb=m1(a) ∨ a ∈ dom m2 ∧ bb=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ bb=m1(a) ∨ a ∈ dom m2 ∧ bb=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

B.6 λ-Calculus + Functions

B.6.1 The λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

B.6.2 Free and Bound Variables 262

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

B.6.3 Substitution 263

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λ*x*•*P*)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λ*y*• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

B.6.4 α-Renaming and β-Reduction 264

- α-renaming: λx•M

  If x, y are distinct variables then replacing x by y in λx•M results in λy•**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: (λx•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. (λx•M)(N) ≡ **subst**([N/x]M)

B.6.5 Function Signatures 265

For sorts we may want to postulate some functions:

```
type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: BB×C → A
```

B.6.6 Function Definitions 266

Functions can be defined explicitly:

```
value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments ~→ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)
```

267

Or functions can be defined implicitly:

```
value
  f: Arguments → Result
  f(args) as result
  post P1(args,result)

  g: Arguments ~→ Result
  g(args) as result
  pre P2(args)
  post P3(args,result)
```

The symbol $\tilde{\rightarrow}$ 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.

B.7 Other Applicative Expressions 268

B.7.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

B.7.2 Recursive let Expressions

Recursive **let** expressions are written as:

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

B.7.3 Predicative let Expressions

Predicative **let** expressions:

let a:A • $\mathcal{P}$(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).

B.7.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

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**

B.7.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

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
```

B.7.6 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⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | ∘
⟨Suffix_Op⟩ ::= !

B.8 Imperative Constructs

B.8.1 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.

```
  Unit
value
  stmt: Unit → Unit
  stmt()
```

- Statements accept no arguments.

- Statement execution changes the state (of declared variables).
- **Unit** → **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**.

B.8.2 Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

B.8.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

2. **skip**
3. stm_1;stm_2;...;stm_n

B.8.4 Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

B.8.5 Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

B.8.6 Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

B.9 Process Constructs

B.9.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

B.9.2 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:

```
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 (‖) of two processes, or the nondeterministic choice between two processes: either external ([]) or internal (⌈⌉). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

B.9.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

```
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.

B.9.4 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.

```
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.

B.10 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:

type
...
variable
...
channel
...
value
...
axiom
...

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[29]

B.11 RSL Index

[29] For schemes, classes and objects we refer to [40, Chap. 10]

C RSL+ 269

TO BE WRITTEN

D A Language of Domain Analysis & Description Prompts 270

TO BE WRITTEN

E A Description Narration Language 271

TO BE WRITTEN

F Indexes

272

F.1 Philosophy Index

Philosophers:

Ideas:

F.2 Domain Analysis Index

F.2.1 Concepts

F.2.2 Definitions

F.2.3 Analysis Predicates

F.2.4 Description Observers

F.2.5 Attribute Category

F.2.6 $\mathcal{P}$roof $\mathcal{O}$bligations and $\mathcal{A}$xioms

F.2.7 Observer Function Literals