

Domain Analysis and Description

1

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Danmark

DTU* Compute†, DK-2800 Kgs. Lyngby, Denmark

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

June 30, 2014: 10:17 am

*It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.*

Fyodor Dostoyevsky, The Idiot, 1868, Part 3, Sect. V

Abstract

2

We present a new approach to *domain analysis &¹ description*. A '**domain**'² is seen as a mapping from *entities* to *qualities*, that is, a mapping from manifest phenomena to usually non-manifest qualities. This idea is then concretised, considerably, in the form of analysing *entities* into either **endurant entities** and **perdurant entities**. We shall then analyse *endurant entities* into either **discrete entities**, and, when so, into either **atomic entities** or **composite entities**, or into **continuous entities**. For *discrete endurants* we shall analyse qualities into **unique identity**, **mereology** and a set of **attributes**. We shall distinguish between external qualities and internal qualities. External qualities are such which are observable: whether *endurant* or *perdurant*, whether discrete or continuous, whether atomic or composite. Internal qualities are (usually) not observable: unique identity, mereology and sets of attributes. The terms *endurant entities* and *perdurant entities* are terms we prefer in the context of domains. In the context of computing one may think of *endurant entities* as *data* and *perdurant entities* as *processes*.

3

4

5

6

The study behind this paper represents an ambitious undertaking: Within the relatively short span of some 51 pages³ it surveys a rather comprehensive set of techniques and tools for the description of non-trivial, non-“toy-example” domains. The reader must, please, be prepared, to cast aside previous conceptions of how domains may be specified; the present proposal “starts all over, from basic principles”. Perhaps a short monograph rather than a “longish” paper would be a better way to reach a reasonable audience, We shall see.

*DTU: Technical University of Denmark

†The former **DTU Informatics** was renamed, by Jan.1, 2013, into **DTU Compute**. It now consists of the former Departments of Mathematics, Computer Science & Engineering and Mathematical Modeling (i.a., Applied Math.). Its English long name is: **Department of Mathematics and Computer Science**.

¹When we write **A & B** we mean that the conjunction of the two concepts **A** and **B** form “a somehow” inseparable” concept.

²This is the first of a number of complementary characterisations of the concept of a domain. All these occurrences are marked by footnotes: Footnotes 2, 5 on the next page, 6 on the following page and 32 on page 42.

³Almost 16 of the paper's 51 pages are examples.

1 Introduction 7

We offer a solution to the problem of precisely describing, informally and formally, spatio-temporal⁴, human- and artifact-assisted domains.⁵

Example 1 . Domains: *Examples of domains are: air traffic, banks, container lines, documents, hospitals, manufacturing, pipelines, railways and road nets.* ■

1.1 The Problem Area 8

The problem area is broadly that of methods for software development and more narrowly that of methods for domain analysis & domain description.

1.1.1 Domains and Entities 9

By a '**domain**'⁶ we shall here understand an area of human activity characterised by observable phenomena⁷: entities, whether endurants (manifest parts and materials) or perdurants (actions, events and behaviours), whether discrete or continuous, and, if discrete, whether atomic or composite. and of their qualities⁸.

1.1.2 The TripTych Approach to Software Engineering 10

We suggest a TripTych view of software engineering: *before software can be designed and coded we must have a reasonable grasp of "its" requirements; before requirements can be prescribed we must have a reasonable grasp of "the underlying" domain.* To us, therefore, software engineering consists of three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

This paper focus on aspects of a methodology for domain analysis & domain description. References [12, 14] show how to "refine" domain descriptions into requirements prescriptions, and reference [16] indicates more general relations between domain descriptions and domain demos, domain simulators and more general domain specific software.

1.1.3 Method 13

By a '**method**' we shall understand a "somehow structured" set of principles for selecting and applying a number of techniques and tools for analysing problems and synthesizing solutions for a given domain.

The 'somehow structuring' amounts, in this treatise on domain analysis & description, to the techniques and tools being related to a set of domain analysis & description "prompts", "issued by the method", prompting the domain analyser, hence carried out by the domain analyser & describer⁹ — conditional upon the result of other prompts.

A Critique: There may be other 'definitions' of the term 'method'. The above is the one that will be adhered to in this paper. The main idea is that there is a clear understanding of what we mean by, as here, a software development method, in particular a *domain analysis & description method*.

⁴Spatio-temporal: Having both spatial and temporal properties

⁵This sentence loosely characterises the kind of domains for which our analysis & description techniques are well suited. It is our second characterisation of the concept of domain.

⁶This is our third characterisation of the concept of domain.

⁷As to what characterises an **observable phenomenon** we shall not speculate. This would entail a longer discourse [2, 84, 73, 99, 85, 98, 83, 28, 88, 67, 42, 90, 62, 100, 29].

⁸The highlighted terms are defined in Sect. 3–3.3.

⁹We shall use the term **domain engineer** to cover both the analyser & the describer.

• • •

A main contribution of this paper is therefore that of “painstakingly” elucidating the principles, techniques and tools of the domain analysis & description method.

1.1.1.4 Methodology

17

By ‘**methodology**’ we shall understand the study and knowledge about one or more methods.¹⁰

1.1.1.5 Towards a Methodology of Domain Analysis & Description

18

By a ‘**domain description**’ we shall understand a text which describes the entities of the domain: whether enduring or perdurant, and when enduring whether discrete or continuous, atomic or composite; or when perdurant whether actions, events or behaviours. as well as the qualities of these entities.

19

Practicalities of Domain Analysis & Description How does one go about analysing & describing a domain? Well, for the first, one has to designate one or more domain analysers cum domain describers, i.e., trained domain scientists cum domain engineers. How does one get hold of a domain engineer? One takes a software engineer and *educates* and *trains* that person in domain science & domain engineering. A derivative purpose of this paper is to unveil aspects of domain science & domain engineering. The education and training consists in bringing forth a number of scientific and engineering issues of domain analysis and of domain description. Among the engineering issues are such as: *what do I do when confronted with the task of domain analysis?* and *with the task of description?* and *when, where and how do I select and apply which techniques and which tools?* Finally, there is the issue of *how do I, as a domain describer, choose appropriate abstractions and models?*

20

21

22

The Four Domain Analysis & Description “Players” We can say that there are four ‘players’ at work here. the domain, the domain analyser & describer, the domain analysis & description method, and the evolving domain analysis & description. (i) The *domain* is there. The domain analyser & describer cannot change the domain. Analysing & describing the domain does not change it¹¹. In a meta-physical sense it is inert. In the physical sense the domain will usually contain parts that are static (i.e., constant), and parts that are dynamic (i.e., variable). (ii) The *domain analyser & domain describer* is a human, preferably a scientist/engineer¹², well-educated and trained in domain science & engineering. The domain analyser & describer observes the domain, analyses it according to a method and thereby produces a domain description. (iii) As a concept the *method* is here considered “fixed”. By ‘fixed’ we mean that its principles, techniques and tools do not change during a domain analysis & description. The domain analyser & describer may very well apply these principles, techniques and tools more-or-less haphazardly during domain analysis & description, flaunting the method, but that method remains invariant. The method, however, may vary from one domain analysis & description (project) to another domain analysis & description (project). Domain analysers & describer, may, for example, have become wiser from a project to the next. (iv) Finally there is the evolving *domain analysis & description*. That description is a text, usually both informal and formal. Applying a *domain description synthesiser* to the domain yields an *additional domain description text* which is added to the thus evolving *domain description*. One may speculate of the rôle of the “input” domain description. Does it change? Does it help determine the additional domain description text? Etcetera. Without loss of generality we can

23

24

25

26

¹⁰Please note our distinction between method and methodology. We often find the two, to us, separate terms used interchangeably.

¹¹PHILOSOPHICAL NOTE: Observing domains, such as we are trying to encircle the concept of domain, is not like observing the physical world at the level of subatomic particles. The experimental physicists’ instruments of observation changes what is being observed.

¹²At the present time domain analysis appears to be partly an art, partly a scientific endeavour. Until such a time when domain analysis & description principles, techniques and tools have matured it will remain so.

assume that the “input” domain description is changed¹³ and that it helps determine the added text.

An Interactive Domain Analysis & Description Dialogue We see domain analysis & description as a process involving the above-mentioned four ‘players’, that is, as a dialogue between the domain analyser & describer and the domain, where the dialogue is guided by the method and the result is the description. We see the method as a ‘player’ which issues prompts: alternating between: “*analyse this*” (analysis prompts) and “*describe that*” (synthesis or, rather, description prompts).

Prompts In this paper we shall suggest a number of *domain analysis prompts* and a number of *domain description prompts*. The ‘**domain analysis prompt**’s, schematically `analyse_named_condition(p)`, direct the analyser to inquire as to the truth of whatever the prompt “names” at wherever part (or material), *p*, in the domain the prompt so designates. Based on the truth value of an analysed part the domain analyser may then be prompted to describe that part (or material). The ‘**domain description prompt**’s, schematically `describe_type_or_quality(e)`, direct the (analyser cum) describer to formulate both an informal and a formal description of the type or qualities of the entity (part or material) designated by the prompt.

The prompts form languages, and there are thus two languages at play here.

A Domain Analysis & Description Language The ‘Domain Analysis & Description Language’ thus consists of a number of meta-functions, the prompts. The meta-functions have names (say `is_endurant`) and types, but have no formal definition. They are not computable. They are “performed” by the domain analysers & describers. These meta-functions are systematically introduced and informally explained in Sect. 3.3.

The Domain Description Language The ‘Domain Description Language’ is RSL [44], the RAISE Specification Language [45]. With suitable, simple adjustments it could also be either of Alloy [53], Event B [1], VDM-SL [23, 24, 40] or Z [101]. We have chosen RSL because of its simple provision for defining sorts, expressing axioms, and postulating observers over sorts.

Domain Descriptions: Narration & Formalisation Descriptions *must* be readable and *should* be mathematically precise¹⁴. For that reason we decompose domain description fragments into clearly identified¹⁵ “pairs” of narrative texts and formal texts. We refer to Pages 10–11.

1.2 Our Contribution

33

The contribution, we *claim*, consists of three sets of concepts: [1] a set of domain description components, [2] a corresponding collection of domain observer functions, and [3] a set of domain analysis & description prompts. These are covered “triple-by-triple” in Sect. 3 on page 7.

1.2.1 Domain Engineering as a Separate Activity

34

There is, however, a more important contribution hidden here: it is that of promoting domain analysis & description as a prerequisite development activity separate from and prior to requirements prescription; in fact, more generally, as a “free-standing” activity: one that can be pursued whether or not we subsequently pursue requirements prescription. Publications [10, 11, 12, 13, 14, 16] discuss these issues.

¹³for example being “stylistically” revised.

¹⁴In order to be able to verify that domain descriptions satisfy a number of qualities not explicitly formulated as well as in order to verify that requirements prescriptions satisfy domain descriptions we must insist on formalised domain descriptions.

¹⁵The “clear identification” is here achieved by narrative text item and corresponding formula line numbers.

1.2.2 Domain Description Components

35

We suggest an approach to domain analysis & description which, in this paper, focus on domains as consisting of endurant entities, their atomicity or compositionality, the sorts (i.e., abstract types) of parts and materials, and subparts of parts; and their qualities: (i) identity, (ii) how composite parts are composed, that is, mereology, and (iii) their attributes. Perdurants will also be covered in this paper, but not systematically, with no enumeration of prompts and with no explicit listing of domain description schemas. 36

1.2.3 Domain Observer Functions

37

We consider as novel the following dual (a, b) aspects of this paper. First (a) the separation of parts and materials, hence (a_1) the notions of *part sorts* and *material sorts*, (a_2) the concepts of *part sort observer functions* and *material sort observer functions* (**obs_P** and **obs_M**), and (a_3) the concepts of concrete part types ($P = \text{Type_Expr}$). That is, firstly the focus on external qualities of endurants. Then (b) the systematic treatment of part and material qualities: (b_1) the notions of *unique part identifiers* (**uid_**) and *unique part identifier types* (ιP). (b_2) the notion of *mereology: part-hood relations* (**mereo_P**) and *mereology types* $(\mathcal{E}(\iota P_1, \iota P_2, \dots, \iota P_m))^{16}$, and (b_3) the notion of *part attribute types* (say $A, B, \dots, C$) and *part attribute type observers* (**attr_A**, **attr_B**, ..., **attr_C**). That is, secondly the focus on internal qualities of endurants. 38

1.2.4 The Domain Analysis & Description Prompts

39

We claim that the above structuring of the domain analysis is new: a clear separation of parts and materials (external quality) analysis from the (internal quality) analysis of their qualities; and a clear provision of tools and analysis techniques “manifest” in the form of (first) domain analysis prompts: **is_endurant**, **is_perdurant**, **is_discrete**, **is_continuous** (i.e., **is_part**, **is_material**), **is_atomic_part**, **is_composite_part**, **has_concrete_type**, **has_material**, and then (second) domain description prompts: **obs_part_sorts**, **obs_part_type**, **obs_material_sorts**, **obs_** 40
unique_identifier, **obs_mereology**, **attribute_names** and **obs_attributes**.

• • •

Section 5.2 reviews the above claims.

2 Formal Concept Analysis

41

2.1 A Formalisation

This section is a transcription of Ganter & Wille’s [43] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.

Some Notation: By $\mathcal{E}$ we shall understand the type of entities, i.e., an entity; by $\mathbb{E}$ we shall understand an entity of type $\mathcal{E}$; by $\mathcal{Q}$ we shall understand the type of qualities; by $\mathbb{Q}$ we shall understand a quality of type $\mathcal{Q}$; by $\mathcal{E}\text{-set}$ we shall understand the type of sets of entities; by $\mathbb{E}\mathcal{S}$ we shall understand a set of entities, i.e., a value of type $\mathcal{E}\text{-set}$; by $\mathcal{Q}\text{-set}$ we shall understand a set of qualities, i.e., the type of sets of qualities; and by $\mathbb{Q}\mathcal{S}$ we shall understand a value of type $\mathcal{Q}\text{-set}$. 42

Definition: 1 Formal Context: A ‘**formal context**’ $\mathbb{K} := (\mathbb{E}\mathcal{S}, \mathbb{I}, \mathbb{Q}\mathcal{S})$ consists of two sets; $\mathbb{E}\mathcal{S}$ of entities and $\mathbb{Q}\mathcal{S}$ of qualities, and a relation $\mathbb{I}$ between $\mathbb{E}$ and $\mathbb{Q}$. ■

To express that $\mathbb{E}$ is in relation $\mathbb{I}$ to a Quality $\mathbb{Q}$ we write $\mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}$, which we read as “entity $\mathbb{E}$ **has** quality $\mathbb{Q}$ ”. 43

¹⁶ $\mathcal{E}$ is a type expression over *unique part identifier types*

Example enduring entities are a specific vehicle, another specific vehicle, etcetera; a specific street segment (link), another street segment, etcetera; a specific road intersection (hub), another specific road intersection, etcetera, a monitor. Example enduring entity qualities are has mobility, has velocity (≥ 0), has acceleration (≥ 0), has length (> 0), has location, has traffic state, etcetera.

Definition: 2 Qualities Common to a Set of Entities: For any subset, $s\mathbb{E} \subseteq \mathbb{E}$, of entities we can define $\mathcal{DQ}$ for “derive set of qualities”.

$$\begin{aligned} \mathcal{DQ} : \mathcal{E}\text{-set} &\rightarrow (\mathcal{E}\text{-set} \times \mathcal{I} \times \mathcal{Q}\text{-set}) \rightarrow \mathcal{Q}\text{-set} \\ \mathcal{DQ}(s\mathbb{E})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) &\equiv \{Q \mid Q:Q, E:\mathcal{E} \bullet E \in s\mathbb{E} \wedge E \cdot \mathbb{I} \cdot Q\} \\ \text{pre: } s\mathbb{E} &\subseteq \mathbb{E} \end{aligned}$$

The above expresses: “the set of qualities common to entities in $s\mathbb{E}$ ”.

Definition: 3 Entities Common to a Set of Qualities: For any subset, $s\mathbb{Q} \subseteq \mathbb{Q}$, of qualities we can define $\mathcal{DE}$ for “derive set of entities”.

$$\begin{aligned} \mathcal{DE} : \mathcal{Q}\text{-set} &\rightarrow (\mathcal{E}\text{-set} \times \mathcal{I} \times \mathcal{Q}\text{-set}) \rightarrow \mathcal{E}\text{-set} \\ \mathcal{DE}(s\mathbb{Q})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) &\equiv \{E \mid E:\mathcal{E}, Q:Q \bullet Q \in s\mathbb{Q} \wedge E \cdot \mathbb{I} \cdot Q\}, \\ \text{pre: } s\mathbb{Q} &\subseteq \mathbb{Q} \end{aligned}$$

The above expresses: “the set of entities which have all qualities in $s\mathbb{Q}$ ”.

Definition: 4 Formal Concept: A ‘formal concept’ of a context $\mathbb{K}$ is a pair:

- $(s\mathbb{Q}, s\mathbb{E})$ where
 - $\bowtie \mathcal{DQ}(s\mathbb{E})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{Q}$ and
 - $\bowtie \mathcal{DE}(s\mathbb{Q})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{E}$;
- $s\mathbb{Q}$ is called the ‘intent’ of $\mathbb{K}$ and $s\mathbb{E}$ is called the ‘extent’ of $\mathbb{K}$.

2.2 Types Are Formal Concepts

47

Now comes the “crunch”: *In the TripTych domain analysis we strive to find formal concepts and, when we think we have found one, we assign a type (or a sort) and qualities to it!*

2.3 Practicalities

48

There is a little problem. To search for all those entities of a domain which each have the same sets of qualities is not feasible. So we do a combination of two things: (i) we identify a small set of entities all having the same qualities and tentatively associate them with a type, and (ii) we identify certain nouns of our national language and if such a noun does indeed designate a set of entities all having the same set of qualities then we tentatively associate the noun with a type. Having thus, tentatively, identified a type we conjecture that type and search for counterexamples, that is, entities which refutes the conjecture. This “process” of conjectures and refutations is iterated until some satisfaction is arrived at that the postulated type constitutes a reasonable conjecture.

2.4 Formal Concepts: A Wider Implication

50

The formal concepts of a domain form Galois Connections [43]. We gladly admit that this fact is one of the reasons that we emphasise formal concept analysis. At the same time we must admit that this paper does not do justice to this fact. We have experimented with the analysis & description of a number of domains and have noticed such Galois connections but it is, for us, too early to report on this. Thus we invite the reader to study this aspect of domain analysis.

3 Endurant Entities

51

3.1 General

Definition 1 . Entity: By an '**entity**' we shall understand a '**phenomenon**', i.e., something that can be observed, i.e., be seen or touched by humans, or that can be conceived as an abstraction of an entity ■

52

Endurant 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 ■

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

3.2 Endurants and Perdurants

53

Definition 2 . Endurant: By an '**endurant**' we shall understand an entity that can be observed or conceived, as a “complete thing”, at no matter which given snapshot of time. Were we to “freeze” time we would still be able to observe the entire endurant ■

54

Example 2 . Traffic System Endurants Examples of traffic system endurants are: traffic system, road nets, fleets of vehicles, sets of hubs (i.e., street intersections), sets of links (i.e., street segments [between hubs]) hubs, links, vehicles. ■

55

Definition 3 . Perdurant: By a '**perdurant**' we shall understand an entity for which only a fragment exists if we look at or touch them at any given snapshot in time, that is, where we to freeze time we would only see or touch a fragment of the perdurant.

56

Example 3 . Traffic System Perdurants Examples of road net perdurants are: insertion and removals of hubs or links (actions), disappearance of links (events), vehicles entering or leaving the road net (actions), vehicles crashing (events) and road traffic (behaviour). ■

57

Endurant Analysis Prompt 2 . is_endurant: The domain analyser analyses entities e into endurants as prompted by the '**domain analysis prompt**':

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

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

Endurant Analysis Prompt 3 . is_perdurant: The domain analyser analyses entities e into perdurants as prompted by the '**domain analysis prompt**':

- *is_perdurant* — ϕ is a perdurant if *is_perdurant*(ϕ) holds.

is_entity is a '**prerequisite prompt**' for *is_perdurant* ■

3.3 Endurants

58

Definition 4 . Parts: By a '**discrete endurant**' we shall understand something which is separate or distinct in form or concept, consisting of distinct or separate parts. We use the term '**part**' for discrete endurants.

59

Example 4 . Parts: Example 2 on the previous page illustrated, and examples 7 on the facing page and 8 on the next page illustrate discrete endurants. ■

60

Definition 5 . Material: By a '**continuous endurant**' we shall understand something which is prolonged without interruption, in an unbroken series or pattern ■ We use the term '**material**' for continuous endurants ■

61

Example 5 . Materials Examples of material endurants are: air of an air conditioning system, grain of a silo, gravel of a barge, oil (or gas) of a pipeline, sewage of a waste disposal system, and water of a hydro-electric power plant. ■

62

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

- `is_discrete` — e is discrete if `is_discrete(e)` holds ■

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

- `is_continuous` — e is continuous if `is_continuous(e)` holds ■

63

We shall call discrete endurants '**part**'s, i.e., `is_part(e) ≡ is_discrete(e)` and continuous endurants '**material**'s, i.e., `is_material(e) ≡ is_continuous(e)` Discrete endurants, i.e., parts, may contain continuous endurants, i.e., material.

Example 6 . Parts Containing Materials Pipeline units, u , are here considered discrete, i.e., parts. Pipeline units serve to convey material, i.e., continuous endurants. ■

64

So which is the result of applying the `is_discrete` prompt to a pipeline unit u ? The answer is: it all depends! That is, it is the domain analyser who decides on which one phenomenon is subordinated the other, that is, whether the material is an aspect of a part, or the part is subservient to the material!

• • •

In this chapter we shall focus mostly on discrete endurants. Section 3.11 on page 26 will cover parts containing materials in a bit more detail.

3.4 Atomic and Composite Parts

65

Definition 6 . Discrete Endurant: By a '**part**', to recall, we mean a discrete endurant ■

A distinguishing quality of discrete endurants, is whether they are atomic or composite.

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

67

Example 7 . Atomic Parts Examples of atomic parts of the above mentioned domains are: aircraft (of air traffic), demand/deposit accounts (of banks), containers (of container lines), documents (of document systems), hubs, links and vehicles (of road traffic), patients, medical staff and beds (of hospitals), pipes, valves and pumps (of pipeline systems), and rail units and locomotives (of railway systems). ■

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

Example 8 . Composite Parts Examples of atomic parts of the above mentioned domains are: airports and air lanes (of air traffic), banks (of a financial service industry), container vessels (of container lines), dossiers of documents (of document systems), routes (of road nets), medical wards (of hospitals), pipelines (of pipeline systems), and trains, rail lines and train stations (of railway systems). ■

Endurant Analysis Prompt 8 . is_atomic: The domain analyser analyses a discrete endurant, i.e., a part p into an atomic endurant:

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

Endurant Analysis Prompt 9 . is_composite: The domain analyser analyses a discrete endurant, i.e., a part p into a composite endurant:

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

$is_discrete$ is a 'prerequisite prompt' of both is_atomic and $is_composite$.

3.5 On Observing Part Sorts

71

3.5.1 Types and Sorts

We use the term 'sort' when we wish to speak of an abstract type [86], that is, a type for which we have no model¹⁷. We shall use the term 'type' to cover both abstract types and concrete types.

3.5.2 On Discovering Part Sorts

72

Recall from Sect. 2 that we "equate" a formal concept with a type (i.e., a sort). Thus, to us, a part sort is a set of all those entities which all have exactly the same qualities.

Our aim is to present the basic principles that let the domain analyser decide on part sorts. We observe parts (i.e., discrete, manifest endurants), one-by-one. (α) *Our analysis of parts concludes when we have "lifted" our examination of a particular part instance to the conclusion that it is of a given sort, that is, reflects, or is, a formal concept.*

Thus there is, in this analysis, a "eureka", a step where we shift focus from the concrete to the abstract, from observing specific part instances to postulating a sort: from one to the many.

Endurant Analysis Prompt 10 . observe_parts: The 'domain analysis prompt':

- $observe_parts(p)$

directs the domain analyser to observe the subparts of p ; let us say they are: $\{p_1, p_2, \dots, p_m\}$ ■

(β) The analyser analyses, for each of these parts, p_{i_k} , which formal concept, i.e., sort it belongs to; let us say that it is of sort P_k ; thus the subparts of p are of sorts $\{P_1, P_2, \dots, P_m\}$. Some P_k may be atomic sorts, some may be composite sorts.

The domain analyser continues to examine a finite number of other composite parts: $\{p_j, p_\ell, \dots, p_n\}$. It is then "discovered", that is, decided, that they all consists of the same number of subparts $\{p_{i_1}, p_{i_2}, \dots, p_{i_m}\}$, $\{p_{j_1}, p_{j_2}, \dots, p_{j_m}\}$, $\{p_{\ell_1}, p_{\ell_2}, \dots, p_{\ell_m}\}$, ..., $\{p_{n_1}, p_{n_2}, \dots, p_{n_m}\}$, of the same, respective, part sorts. (γ) *It is therefore concluded, that is, decided, that $\{p_i, p_j, p_\ell, \dots, p_n\}$ are all of the same part sort P with observable part sub-sorts $\{P_1, P_2, \dots, P_m\}$.*

¹⁷for example, in terms of the concrete types: sets, Cartesians, lists, maps, or other.

Above we have *type-font-highlighted* three sentences: (α, β, γ) . When you analyse what they “prescribe” you will see that they entail a “depth-first search” for part sorts. The β sentence says it rather directly: “*The analyser analyses, for each of these parts, p_k , which formal concept, i.e., part sort it belongs to.*” To do this analysis in a proper way, the analyser must (“recursively”) analyse the parts “down” to their atomicity, and from the atomic parts decide on their part sort, and work (“recurse”) their way “back”, through possibly intermediate composite parts, to the p_k s.

3.5.3 Part Sort Observer Functions

77

The above analysis amounts to the analyser first “applying” the domain analysis prompt `is_composite(p)` to a discrete enduring, where we now assume that the obtained truth value is **true**. Let us assume that parts $p:P$ consists of subparts of sorts $\{P_1, P_2, \dots, P_m\}$. Since we cannot automatically guarantee that our domain descriptions secure that P and each P_i ($1 \leq i \leq m$) denotes disjoint sets of entities we must prove it.

Domain Description Prompt 1 *observe_part_sorts* `observe-part-sorts` If `is_composite(p)` holds, then the analyser “applies” the description language observer prompt

- `observe_part_sorts(p)`

resulting in the analyser writing down the *part sorts and part sort observers domain description text* according to the followig schema:

1. `observe_part_sorts` schema

Narration:

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

Formalisation:

type

- [s] P ,
- [s] P_i [$1 \leq i \leq m$] **comment:** P_i [$1 \leq i \leq m$] abbreviates $P_1, P_2, \dots, P_m$

value

- [o] **obs- P_i** : $P \rightarrow P_i$ [$1 \leq i \leq m$]
- [i] **is- P_i** : $P_i \rightarrow \mathbf{Bool}$ [$1 \leq i \leq m$]

proof obligation [*Disjointness of part sorts*]

- [p] $\forall p: (P_1 | P_2 | \dots | P_m) \bullet$
- [p] $\bigwedge \{ \mathbf{is-}P_i(p) \equiv \bigvee \sim \{ \mathbf{is-}P_j(p) \mid j \in \{1..m\} \setminus \{i\} \} \mid i \in \{1..m\} \}$

is_composite is a ‘prerequisite prompt’ of *observe_part_sorts* ■

We do not here state guidelines for discharging these kinds of proof obligations. But we will very informally sketch such discharges, see below.

Example 9 . Composite and Atomic Part Sorts of Transportation *The following example illustrates the multiple use of the `observe_part_sorts` function: first to δ , a specific transport domain, Item 1, then to an $n : N$, the net of that domain, Item 2, and then to an $f : F$, the fleet of that domain, Item 3.*

- 1 A transportation domain is viewed as composed from a net (of hubs and links), a fleet (of vehicles) and a monitor.
- 2 A transportation net is here seen as composed from a collection of hubs and a collection of links.

3 A fleet is here seen as a collection of vehicles.

The monitor is considered an atomic part.

81

type

1. N, F, M

value

1. $\text{obs}_N:\Delta \rightarrow N, \text{obs}_F:\Delta \rightarrow F, \text{obs}_M:\Delta \rightarrow M$

type

2. HC, LC

value

2. $\text{obs}_{HC}:N \rightarrow HC, \text{obs}_{LC}:N \rightarrow LC$

type

3. VC

value

3. $\text{obs}_{VC}:F \rightarrow VC$

82

A proof obligation has to be discharged, one that shows disjointness of sorts N, F and M . An informal sketch is: entities of sort N are composite and consists of two parts: aggregations of hubs, HS , and aggregations of links, LS . Entities of sort F consists of an aggregation, VS , of vehicles. So already that makes N and F disjoint. M is an atomic entity — where N and F are both composite. Hence the three sorts N, F and M are disjoint. Experimental research report [21] covers [road] transportation in some detail. ■

3.5.4 On Discovering Concrete Part Types

83

Endurant Analysis Prompt 11 . **has_concrete_type**: The domain analyser may decide that it is expedient, i.e., pragmatically sound, to render a part sort, P , whether atomic or composite, as a concrete type, T . That decision is prompted by the holding of the '**domain analysis prompt**':

- $\text{has_concrete_type}(p)$.

is_discrete is a '**prerequisite prompt**' of has_concrete_type ■

84

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

- $\text{observe_part_type}^{18}$

[b.]

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

85

2. **observe_part_type** schema

Narration:

- $[t_1]$... narrative text on types ...
- $[t_2]$... narrative text on types ...
- $[o]$... narrative text on type observers ...

Formalisation:

type

$[t_1]$ $Q, R, \dots, S$

$[t_2]$ $T = \mathcal{E}(Q, R, \dots, S)$

value

$[o]$ $\text{obs}_T: P \rightarrow T$

where $Q, R, \dots, S$ may be any types, including part sorts and $\mathcal{E}(Q, R, \dots, S)$ is a type expression ■

The type names, T , of the concrete type, as well as those of the auxiliary types, $Q, R, \dots, S$, are chosen by the domain describer: they may have already been chosen for other sort-to-type descriptions, or they may be new.

Example 10 . Concrete Part Types of Transportation *We continue Example 9 on page 10:*

4 *A collection of hubs is here seen as a set of hubs and a collection of links is here seen as a set of links.*

5 *Hubs and links are, until further analysis, part sorts.*

6 *A collection of vehicles is here seen as a set of vehicles.*

7 *Vehicles are, until further analysis, part sorts.*

type

4. $Hs = H\text{-set}, Ls = L\text{-set}$

5. H, L

6. $Vs = V\text{-set}$

7. V

value

4. **obs_** $Hs:HC \rightarrow H\text{-set}, \text{obs_}Ls:LC \rightarrow L\text{-set}$

6. **obs_** $Vs:VC \rightarrow V\text{-set}$ ■

3.5.5 Forms of Part Types

88

Usually it is wise to restrict the part type definitions, $T_i = \mathcal{E}_i(Q, R, \dots, S)$, to simple type expressions. $T = A\text{-set}$ or $T = A^*$ or $T = ID \overrightarrow{mk} A$ or $T = A_t | B_t | \dots | C_t$ where ID is a sort of unique identifiers, $T = A_t | B_t | \dots | C_t$ defines the disjoint types $A_t == mkA_s(s:A_s), B_t == mkB_s(s:B_s), \dots, C_t == mkC_s(s:C_s)$, and where $A, A_s, B_s, \dots, C_s$ are sorts. Instead of $A_t == mkA(a:A_s)$, etc., we may write $A_t :: A_s$ etc.

3.5.6 Part Sort and Type Derivation Chains

89

Let P be a composite sort. Let $P_1, P_2, \dots, P_m$ be the part sorts “discovered” by means of `observe_part_sorts(p)` where $p:P$. We say that $P_1, P_2, \dots, P_m$ are (immediately) ‘**derived**’ from P . If P_k is derived from P_j and P_j is derived from P_i , then, by transitivity, P_k is ‘**derived**’ from P_i .

No Recursive Derivations We “mandate” that if P_k is derived from P_j then there can be no P derived from P_j such that P is P_j , that is, P_j cannot be derived from P_j .

That is, we do not allow recursive domain sorts.

It is not a question, actually of allowing recursive domain sorts. It is, we claim to have observed, in very many domain modelling experiments, that there are no recursive domain sorts!

3.5.7 Names of Part Sorts and Types

91

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

Example 11 . Container Line Sorts Our example is that of a container line with container vessels and container terminal ports.

- 8 A container line contains a number of container vessels and a number of container terminal ports, as well as other components.
- 9 A container vessel contains a container stowage area, etc.
- 10 A container terminal port contains a container stowage area, etc.
- 11 A container stowage area contains a set of uniquely identified container bays.
- 12 A container bay contains a set of uniquely identified container rows.
- 13 A container row contains a set of uniquely identified container stacks.
- 14 A container stack contains a stack, i.e., a first-in, last-out sequence of containers.
- 15 Containers are further undefined.

After a some slight editing we get:

```

type
  CL
  VS, VI, V, Vs = VI  $\overline{\mapsto}$  V,
  PS, PI, P, Ps = PI  $\overline{\mapsto}$  P
value
  obs_VS: CL  $\rightarrow$  VS
  obs_Vs: VS  $\rightarrow$  Vs
  obs_PS: CL  $\rightarrow$  PS
  obs_Ps: CTPS  $\rightarrow$  CTPs
type
  CSA
value
  obs_CSA: V  $\rightarrow$  CSA
  obs_CSA: P  $\rightarrow$  CSA

```

```

type
  BAYS, BI, BAY, Bays=BI  $\overline{\mapsto}$  BAY
  ROWS, RI, ROW, Rows=RI  $\overline{\mapsto}$  ROW
  STKS, SI, STK, Stks=SI  $\overline{\mapsto}$  STK
  C
value
  obs_BAYS: CSA  $\rightarrow$  BAYS,
  obs_Bays: BAYS  $\rightarrow$  Bays
  obs_ROWS: BAY  $\rightarrow$  ROWS,
  obs_Rows: ROWS  $\rightarrow$  Rows
  obs_STKS: ROW  $\rightarrow$  STKS,
  obs_Stks: STKS  $\rightarrow$  Stks
  obs_Stk: STK  $\rightarrow$  C*

```

Note that $\text{observe_part_sorts}(v:V)$ and $\text{observe_part_sorts}(p:P)$ both yield CSA ■

3.5.8 More On Part Sorts and Types

95

The above “experimental example” motivates the below. We can always assume that composite parts $p:P$ abstractly consists of a definite number of subparts.

Example 12. We comment on Example 9 on page 10: parts of type Δ and N are composed from three, respectively two abstract subparts of distinct types ■

Some of the parts, say p_{i_z} of $\{p_{i_1}, p_{i_2}, \dots, p_{i_m}\}$, of $p:P$, may themselves be composite.

Example 13. We comment on Example 9 on page 10: parts of type N , F , HC , LC and VC are all composite ■

There are, pragmatically speaking, two cases for such compositionality. Either the part, p_{i_z} , of type t_{i_z} , is composed from a definite number of abstract or concrete subparts of distinct types.

Example 14. We comment on Example 9 on page 10: parts of type N are composed from three subparts ■

Or it is composed from an indefinite number of subparts of the same sort.

Example 15. We comment on Example 9 on page 10: parts of type HC , LC and VC are composed from an indefinite numbers of hubs, links and vehicles, respectively ■

Example 16 . Pipeline Parts

16 A pipeline consists of an indefinite number of pipeline units.

17 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.

18 All these unit sorts are atomic and disjoint.

type

16. $PL, U, We, Pi, Pu, Va, Fo, Jo, Si$

16. $Well, Pipe, Pump, Valv, Fork, Join, Sink$

value

16. **obs_Us**: $PL \rightarrow U\text{-set}$

type

17. $U == We \mid Pi \mid Pu \mid Va \mid Fo \mid Jo \mid Si$

18. $We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo::Fork, Jo::Join, Si::Sink$

The experimental research report [20] covers pipelines in some detail ■

Derivation Lattices Derivation chains start with the domain name, say Δ , and (definitively) end with the name of an atomic sort. Sets of derivation chains form join lattices [7].

Example 17 . Derivation Chains Figure 1 illustrates two part sort and type derivation chains. based on Examples 9 on page 10 and 11 on the preceding page, respectively.

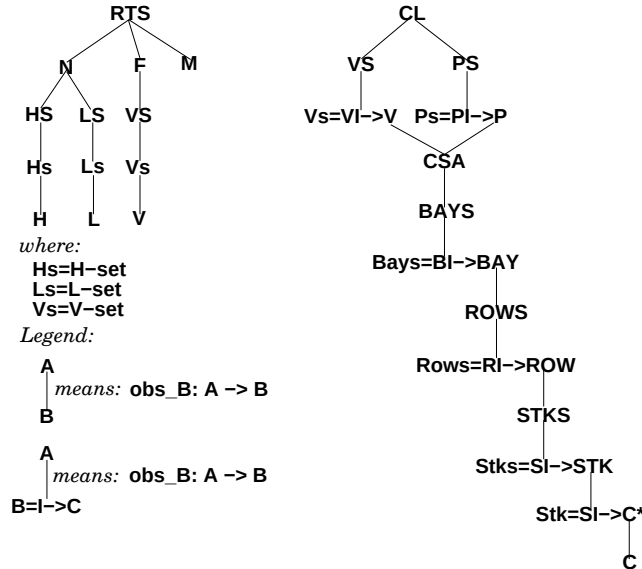


Figure 1: Two Domain Lattices: Examples 9 on page 10 and 11 on the preceding page

The “ $\rightarrow$ ” of Fig. 1 stands for $\overrightarrow{m}$ ■

¹⁸has_concrete_type is a ‘prerequisite prompt’ of observe_part_type.

3.6 Syntactic and Semantic Qualities of Endurants

100

By an '**external endurant quality**' we shall understand the *is atomic, is composite, is discrete and is continuous* qualities. We consider these to reflect '**syntactic qualities**'. By an '**internal endurant quality**' we shall understand the endurant qualities to be outlined in the next sections: *unique identification, mereology and attributes*. By '**part qualities**' we mean the sum total of *external endurant and internal endurant* qualities. We consider these to reflect '**semantic qualities**'.

3.7 Three Categories of Semantic Qualities

101

We suggest that the internal qualities of parts be analysed into three categories: (i) a category of unique part identifiers, (ii) a category of mereological quantities and (iii) a category of general attributes. Part mereologies are about *sharing* qualities between parts. Some such *sharing* expresses spatio-topological properties of how parts are organised. Other part *sharing* aspects express relations (like equality) of part attributes. We base our modelling of mereologies on the notion of unique part identifiers. Hence we cover semantic qualities in the order (i–ii–iii).

3.8 Unique Part Identifiers

103

We can assume, without any loss of generality, that all parts, p , of any domain P , have unique identifiers, that unique identifiers (of parts $p:P$) are abstract values (of the unique identifier sort PI of P), such that distinct part sorts, P_i and P_j , have distinctly named unique identifier sorts, say PI_i and PI_j , and that all $\pi_i:PI_i$ and $\pi_j:PI_j$ are distinct, and that the observer function **uid** _{P} applied to p yields the unique identifier, say $\pi:PI$, of p .

Domain Description Prompt 3 . *observe_unique_identifier:* We can therefore apply the '**domain description prompt**':

- *observe_unique_identifier*

[c.]

to parts $p:P$ resulting in the analyser writing down the unique identifier types and observers domain description text according to the followig schema:¹⁹

105

3. observe_unique_identifier schema

Narration:

- [s] ... narrative text on unique identifier sorts ...
- [u] ... narrative text on unique identifier observers ...
- [a] ... axiom on uniqueness of unique identifiers ...

Formalisation:

- type**
- [s] PI
- value**
- [u] **uid** _{P} : $P \rightarrow PI$
- axiom**
- [a] $\mathcal{U}$

$\mathcal{U}$ is a predicate over part sorts and unique part identifier sorts. The unique part identifier sort, PI , is unique, as are all part sort names, P .

106

Example 18 . Unique Transportation Net Part Identifiers We continue Example 9 on page 10.

¹⁹**Note:** Do we need to secure disjointness of all unique identifier sorts?

19 Links and hubs have unique identifiers

20 and unique identifier observers.

type

19. LI, HI

value

20. $\mathbf{uid_LI}: L \rightarrow LI$

20. $\mathbf{uid_HI}: H \rightarrow HI$

axiom [Well-formedness of Links, L , and Hubs, H]

19. $\forall l, l': L \bullet l \neq l' \Rightarrow \mathbf{uid_LI}(l) \neq \mathbf{uid_LI}(l')$,

19. $\forall h, h': H \bullet h \neq h' \Rightarrow \mathbf{uid_HI}(h) \neq \mathbf{uid_HI}(h')$ ■

3.9 Mereology

107

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 [64, 92, 94].

3.9.1 Part Relations

108

Which are the relations that can be relevant for part-hood? We give some examples.

- Two otherwise distinct parts may share attribute values.²⁰

Example 19 . Shared Attribute Mereology: *Examples: (i) two or more distinct public transport busses may run according to the same, thus “shared”, bus time table; (ii) all vehicles in a traffic participate in that traffic each with their “share”: that is, position on links or at hubs – as observed by the (thus postulated) traffic observer. etcetera.* ■

- Two otherwise distinct parts may be said to, for example, be topologically “adjacent” or one “embedded” within the other.

Example 20 . Topological Connectedness Mereology: *Examples: (i) two rail units may be connected (i.e., adjacent), (ii) a road link may be connected to two road hubs; (iii) a road hub may be connected to zero or more road links; etcetera.* ■

The above 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”!

3.9.2 Part Mereology: Types and Functions

110

If a composite part p consists of a definite number of sub-parts: $a, \dots, c$, then that is the mereology of any p with respect to any $a, \dots, c$, and we need not bother about unique identifiers for p . In this case we have the situation that:

type

$P, A, \dots, C$

value

$\mathbf{obs_A}: P \rightarrow A, \dots, \mathbf{obs_C}: P \rightarrow C.$

If, however, a composite part p consists of an indefinite number of sub-parts, $q_1, \dots, q_n$, that is, if we have the situation that:

²⁰For the concept of attribute value see Sect. 3.10.2 on page 20.

```

type
  P, Q, Ps = Q-set
value
  obs_Ps: P → Q-set,

```

then the mereology need to be further elaborated.

Endurant Analysis Prompt 12 . **has_mereology**: *To do so the analyser can be said to endow a truth value **true** to the 'domain analysis prompt':*

- *has_mereology*

112

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

We can define a '**mereology type**' as a type $\mathcal{E}$ expression over unique [part] identifier types. We generalise to unique [part] identifier over a definite collection of part sorts, $P_1, P_2, \dots, P_n$, where the parts $p_1:P_1, p_2:P_2, \dots, p_n:P_n$ are not necessarily (immediate) sub-parts of some part $p:P$.

```

type
  PI1, PI2, ..., PIn
  MT =  $\mathcal{E}(PI1, PI2, \dots, PIn)$ ,

```

113

Domain Description Prompt 4 . **observe_mereology**: *If $has_mereology(p)$ holds for parts p of type P , then the analyser can apply the 'domain description prompt':*

- *observe_mereology*

[d.]

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

114

4. **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]  $MT = \mathcal{E}(PI1, PI2, \dots, PIm)$ 
value
  [m] mereo_P:  $P \rightarrow MT$ 
axiom [ Well-formedness of Domain Mereologies]
  [a]  $\mathcal{A}(MT)$ 

```

115

Here $\mathcal{E}(PI1, PI2, \dots, PIm)$ is a type expression over possibly all unique identifier types of the domain description, and $\mathcal{A}(MT)$ is a predicate over possibly all unique identifier types of the domain description. To write down the concrete type definition for MT requires a bit of analysis and thinking. *has_mereology* is a '**prerequisite prompt**' for *observe_mereology* ■

116

Example 21 . **Road Net Part Mereologies** We continue Example 9 on page 10 and Example 18 on page 15.

21 Links are connected to exactly two distinct hubs.

- 22 Hubs are connected to zero or more links.
- 23 For a given net the link and hub identifiers of the mereology of hubs and links must be those of links and hubs, respectively, of the net.

117

type

21. $LM' = HI\text{-set}, LM = \{|his:HI\text{-set} \bullet \text{card}(his)=2|\}$

22. $HM = LI\text{-set}$

value

21. $\text{mereo_L}: L \rightarrow LM$

22. $\text{mereo_H}: H \rightarrow HM$

axiom [Well-formedness of Road Nets, N]

23. $\forall n:N, l:L, h:H \bullet l \in \text{obs_Ls}(\text{obs_LC}(n)) \wedge h \in \text{obs_Hs}(\text{obs_GC}(n))$

23. **let** $his=\text{mereology_H}(l)$, $lis=\text{mereology_H}(h)$ **in**

23. $his \subseteq \{\text{uid_H}(h) \mid h \in \text{obs_Hs}(\text{obs_HC}(n))\}$

23. $\wedge lis \subseteq \{\text{uid_H}(l) \mid l \in \text{obs_Ls}(\text{obs_LC}(n))\}$ **end** ■

The experimental research report [21] covers well-formedness of road nets.

Example 22 . Pipeline Parts Mereology We continue Example 16 on page 14. Pipeline units serve to conduct fluid or gaseous material. The flow of these occur in only one direction: from so-called input to so-called output.

- 24 Wells have exactly one connection to an output unit.
- 25 Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.
- 26 Forks have exactly one connection from an input unit and exactly two connections to distinct output units.
- 27 Joins have exactly one two connection from distinct input units and one connection to an output unit.
- 28 Sinks have exactly one connection from an input unit.
- 29 Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

type

29. $UM' = (UI\text{-set} \times UI\text{-set})$

29. $UM = \{|(iuis,ouis):UI\text{-set} \times UI\text{-set} \bullet iuis \cap ouis = \{\}\}| \}$

value

29. $\text{mereo_U}: UM$

axiom [Well-formedness of Pipeline Systems, $PLS(0)$]

$\forall pl:PL, u:U \bullet u \in \text{obs_Us}(pl) \Rightarrow$

let $(iuis,ouis)=\text{mereo_U}(u)$ **in**

case $(\text{card } iuis, \text{card } ouis)$ **of**

24. $(0,1) \rightarrow \text{is_We}(u),$

25. $(1,1) \rightarrow \text{is_Pi}(u) \vee \text{is_Pu}(u) \vee \text{is_Va}(u),$

26. $(1,2) \rightarrow \text{is_Fo}(u),$

27. $(2,1) \rightarrow \text{is_Jo}(u),$

28. $(1,0) \rightarrow \text{is_Si}(u)$

end end

Example 39 on page 28 (axiom Page 28), Example 40 on page 29 (axiom Page 29) and Example 41 on page 30 (axiom Page 31) illustrates the need to constrain the sets of enduring entities denoted by definitions of part sort, unique identifier and mereology attribute definitions. ■

3.9.3 Update of Mereologies

120

We normally consider a part's mereology to be constant. There may, however, be cases where the mereology of a part changes. In order to update mereology values the description language offers the “built-in” operator:

Mereology Update Function: <u>upd_mereology</u>	
• upd_mereology : $P \rightarrow M \rightarrow P$	

for all relevant M and P . The meaning of **upd_mereology** is, informally:

121

type

P, M

value

upd_mereology: $P \rightarrow M \rightarrow P$

upd_mereology(p)(m) as p'

post: **mereo**_ $(p') = m$

122

The above is a simplification. It lacks explaining that all other aspects of the part $p:P$ are left unchanged. It also omits mentioning some proof obligations. The updated mereology must, for example, only specify such unique identifiers of parts that are indeed existing parts. A proper formal explication requires that we set up a formal model of the domain/method/analyser/description quadrangle.

123

Example 23 . Mereology Update *The example is that of updating the mereology of a hub. Cf. Example 21 on page 17.*

30 Inserting a link, $l:L$, between two hubs, $ha:H, hb:H$ require the update of the mereologies of these two existing hubs.

31 The unique identifier of the inserted link, $l:L$, is li , $li = \text{uid_L}(l)$ and h is either ha or hb ;

32 li is joined to the mereology of either ha or hb ; and respective hubs are updated accordingly.

value

30. $\text{update_hub_mereology}$: $H \rightarrow LI \rightarrow H$

31. $\text{update_hub_mereology}(h)(li) \equiv$

32. **let** $m = \{li\} \cup \text{mereo_}(h)$ **in** **upd_mereology**(h)(m) **end** ■

3.10 Discrete Endurant Attributes

124

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

3.10.1 Inseparability of Attributes from Endurants

Endurants are typically recognised because of their spatial form (parts or materials) and are otherwise characterised by their intangible, but measurable attributes. That is, whereas endurants, whether discrete (as are parts) 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 touched²¹, or seen²², but can be objectively measured²³. Thus,

²¹One can see the red colour of a wall, but one touches the wall.

²²One cannot see electric current, and one may touch an electric wire, but only if it conducts reasonably high voltage can one feel it.

²³PHILOSOPHICAL NOTE: That is, we restrict our domain analysis with respect to attributes to such quantities which are observable, say by mechanical, electrical or chemical instruments.

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 learned from Sect. 2 that a formal concept, that is, a type, consists of all the entities which all have the same qualities. Thus removing a quality from an entity makes no sense: the entity of that type either becomes an entity of another type or ceases to exist (i.e., becomes a non-entity)!

3.10.2 Attribute Quality and Attribute Value

125

We distinguish between an attribute, as a logical proposition, and an attribute value, as a value in some not necessarily Boolean value space.

Example 24 . Attribute Propositions and Other Values *A particular street segment (i.e., a link), say ℓ , satisfies the proposition (attribute) `has_length`, and may then have value `length 90 meter` for that attribute. Another link satisfies the same proposition but has another value; And yet another link satisfies the same proposition and may have the same value. That is: all links satisfies `has_length` and has some value for that attribute. A particular road transport domain, δ , has three immediate sub-parts: `net`, n , `fleet`, f , and `monitor` m ; typically `nets` `has_net_name` and `has_net_owner` proposition attributes with, for example, `US Interstate Highway System` respectively `US Department of Transportation` as values for those attributes. There may be other components of the net value n ■*

3.10.3 Endurant Attributes: Types and Functions

126

Let us recall that attributes cover qualities other than unique identifiers and mereology. Let us then consider that parts have one or more attributes. These attributes are qualities which help characterise “what it means” to be a part, that is, a discrete endurant.

Example 25 . Atomic Part Attributes *Examples of attributes of atomic parts such as a human are: `name`, `gender`, `birth-date`, `birth-place`, `nationality`, `height`, `weight`, `eye colour`, `hair colour`, etc. Examples of attributes of transport net links are: `length`, `location`, `1 or 2-way link`, `link condition`, etc. ■*

Example 26 . Composite Part Attributes *Examples of attributes of composite parts such as a road net are: `owner`, `public or private net`, `free-way or toll road`, `a map of the net`, etc. Examples of attributes of a group of people could be: `statistic distributions of gender`, `age`, `income`, `education`, `nationality`, `religion`, etc. ■*

We now assume that all parts (and materials, see Sect. 3.11 on page 26), have attributes. The question is now, in general, how many and, particularly, which.

Endurant Analysis Prompt 13 . `attribute_names`: *The ‘domain analysis prompt’ `attribute_names` when applied to a part p yields the set of names of its attribute types:*

- $\text{attribute_names}(p): \{\eta A_1, \eta A_2, \dots, \eta A_n\}$.

η is a type operator. Applied to a type A it yields is name²⁴ ■

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

The Attribute Value Observer The “built-in” description language operator

- `attr_A`

applies to parts, $p:P$, where $\eta A \in \text{attribute_names}(p)$. It yields the value of attribute A of p .

²⁴Normally, in non-formula texts, type A is referred to by ηA . In formulas A denote a type, that is, a set of entities. Hence, when we wish to emphasize that we speak of the name of that type we use ηA . But often we omit the distinction

A Comprehensive Attributes Observer The “built-in”, part sort independent description language operator

- **obs_attribs**

when applied, **obs_attribs**(e), to an enduring entity (part or material) e of type P (or M) yields the set, **attrs**: P_ATTRS , of attributes of that part such that if $A_1, A_2, \dots, A_n$ are the types of the n attributes of e then for all i ($1 \leq i \leq n$) **attr** $_A_i(e) = \text{attr_}A_i(\text{obs_attribs}(e))$.

133

Domain Description Prompt 5 . **observe_attributes**: The domain analyser experiments, thinks and reflects about part (and material, see later) attributes. That process is initiated by the ‘domain description prompt’:

- **observe_attributes**.

[e.]

The result of that ‘domain description prompt’ writes down the attribute types and observers domain description text according to the following schema:

134

e. observe_attributes schema

Narration:

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

Formalisation:

type

- [t] A_i [$1 \leq i \leq n$]
- [t] P_ATTRS

value

- [o] **obs_attribs**: $P \rightarrow P_ATTRS$
- [o] **attr** $_A_i$:($P|P_ATTRS$) $\rightarrow A_i$ [$1 \leq i \leq n$]
- [i] **is** $_A_i$: $A_i \rightarrow \text{Bool}$ [$1 \leq i \leq n$]

axiom $\forall i:\text{Nat} \cdot 1 \leq i \leq n \Rightarrow \text{attr_}A_i(p) = \text{attr_}A_i(\text{obs_attribs}(p))$

proof obligation [*Disjointness of Attribute Types*]

- [p] $\forall \delta:\Delta$
- [p] let P be any part sort in [*the Δ domain description*]
- [p] let $a:(A_1|A_2|\dots|A_n)$ in **is** $_A_i(a) \neq \text{is_}A_j(a)$ end end [$i \neq j, 1 \leq i, j \leq n$]

The **type** (or rather sort) definitions: $A_1, A_2, \dots, A_n$ inform us that the domain analyser has decided to focus on the distinctly named $A_1, A_2, \dots, A_n$ attributes.²⁵ And the **value** clauses **attr** $_A_1$:($P|P_ATTRS$) $\rightarrow A_1$, **attr** $_A_2$:($P|P_ATTRS$) $\rightarrow A_2$, ..., **attr** $_A_n$:($P|P_ATTRS$) $\rightarrow A_n$ are then “automatically” given: if a part (type P) has an attribute A_i then there is postulated, “by definition” [*eureka*] an attribute observer function **attr** $_A_i$:($P|P_ATTRS$) $\rightarrow A_i$ etcetera ■

135

The fact that, for example, $A_1, A_2, \dots, A_n$ are attributes of $p:P$, means that the propositions

- **has_attribute** $_A_1(p)$, **has_attribute** $_A_2(p)$, ..., and **has_attribute** $_A_n(p)$

holds. Thus the observer functions **attr** $_A_1$, **attr** $_A_2$, ..., **attr** $_A_n$ can be applied to ps in P and yield attribute values $a_1:A_1, a_2:A_2, \dots, a_n:A_n$ respectively.

137

Example 27 . **Road Hub Attributes** After some analysis a domain analyser may arrive at some interesting hub attributes:

²⁵The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena. Cf. Example 25 on the preceding page and Example 26.

33 *hub state: from which links (by reference) can one reach which links (by reference),*

34 *hub state space: the set of all potential hub states that a hub may attain,*

35 *such that*

a the links referred to in the state are links of the hub mereology

b and the state is in the state space.

36 *Etcetera — i.e., there are other attributes not mentioned here.*

type

33. $H\Sigma = (LI \times LI)_{set}$

34. $H\Omega = H\Sigma\text{-set}$

value

33. $\mathbf{attr_H\Sigma} : H \rightarrow H\Sigma$

34. $\mathbf{attr_H\Omega} : H \rightarrow H\Omega$

axiom [*Well-formedness of Hub States, $H\Sigma$*]

35. $\forall h : H \bullet \mathbf{let} \text{ } lis = \mathbf{mereo_H}(h) \mathbf{in}$

35. $\quad \mathbf{let} \text{ } h\sigma = \mathbf{attr_H\Sigma}(h) \mathbf{in}$

35a. $\quad \{li, li' \mid li, li' : LI \bullet (li, li') \in h\sigma\} \subseteq lis$

35b. $\quad \wedge h\sigma \in \mathbf{attr_H\Omega}(h)$

35. $\quad \mathbf{end} \mathbf{end}$

type

36. $\dots, \dots$

value

36. $\mathbf{attr_}\dots, \dots \blacksquare$

3.10.4 Attribute Categories

139

One can suggest a hierarchy of part attribute categories: discrete or continuous (including chaotic) values, 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 inspired by [55, M.A. Jackson].)

Part attributes are either **discrete** or **continuous** or **chaotic** attributes. A part attribute is said to be a '**discrete attribute**', `is_discrete_attribute`, if it takes on a finite or countably infinite number of distinct or unconnected elements. A part attribute is said to be a '**continuous attribute**', `is_continuous_attribute`, if a suitable abstract model describes it as a continuous function from some point set value type A (A could be time) to some not necessarily point set value type (B): $A \rightarrow B$. We shall not explain concepts of chaotic attributes.

Discrete or continuous part attributes are either constant or a variable, i.e., **static** or **dynamic** attributes. By a '**static attribute**', `is_static_attribute`, we shall understand an attribute whose values are constants, i.e., cannot change. By a '**dynamic attribute**', `is_dynamic_attribute`, we shall understand an attribute whose values are variable, i.e., can change.

Dynamic attributes are either inert, reactive or active attributes. By an '**inert attribute**', `is_inert_attribute`, we shall understand a dynamic attribute whose values only change as the result of external stimuli where these stimuli prescribe exactly these new values. By a '**reactive attribute**', `is_reactive_attribute`, we shall understand a dynamic attribute whose values, if they vary, change value in response to the change of other attribute values. By an '**active attribute**', `is_active_attribute`, we shall understand a dynamic attribute whose values change (also) of its own volition.

Example 28 . Inert and Reactive Attributes *Busses (i.e., vehicles) have a timetable attribute which is dynamic, i.e., can change, namely when the operator of the bus decides so, thus the bus timetable attribute is inert. Pipeline valve units include the two attributes of valve opening (open, close) and internal flow (measured, say gallons per second). The valve opening attribute is of the programmable attribute category. The flow attribute is reactive (flow changes with valve opening/closing) ■*

Active attributes are either autonomous, biddable or programmable attributes. By an '**autonomous attribute**', `is_autonomous_attribute`, 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". By a '**biddable attribute**', `is_biddable_attribute` (of a part) we shall understand a dynamic active attribute whose values may be subject to a contract as to which values it is expected to exhibit. By a '**programmable attribute**', `is_programmable_attribute`, we shall understand a dynamic active attribute whose values can be accurately prescribed. ■

Example 29 . Static and Dynamic Link Attributes

37 Some link attributes

- a (link) length,
- b (link) name, e.g., Fifth Ave. between 50th and 51st Streets),

can be considered static,

38 whereas other link attributes

- a (link) state (one-way: say from 51st to 50th),
- b (link) state space (one single state, one-way: say from 51st to 50th)

can be considered programmable,

39 Finally link attributes

- a link state-of-repair,
- b date last maintained,

can be considered inert.

type
 37a. LEN
value
 37a. $\mathbf{obs_LEN}: L \rightarrow LEN$
type
 37b. $Name$
value
 37b. $\mathbf{obs_Name}: L \rightarrow Name$
type
 38a. $L\Sigma' = (HI \times HI)\text{-set}$
 38a. $L\Sigma = \{\mid \omega : L\Sigma \bullet \mathbf{card} \mid \omega \leq 2\}$
value

38a. $\mathbf{obs_L\Sigma}: L \rightarrow L\Sigma$
type
 38b. $L\Sigma' = L\Sigma\text{-set}$
 38b. $L\Sigma = \{\mid \omega : L\Sigma \bullet \mathbf{card} \mid \omega = 1\}$
value
 38b. $\mathbf{obs_L\Sigma}: L \rightarrow L\Sigma$
type
 39a. $LSoR$
 39b. DLM
value
 39a. $\mathbf{obs_LSoR}: L \rightarrow LSoR$
 39b. $\mathbf{obs_DLM}: L \rightarrow DLM$ ■

Example 30 . Autonomous and Programmable Hub Attributes *We continue Example 29. Time progresses autonomously, Hub states are programmed (traffic signals): changing from red to green via yellow, in one pair of (co-linear) directions, while changing, in the same time interval, from green via yellow to red in the "perpendicular" directions ■*

3.10.5 Shared Attributes

148

Normally part attributes of different part sorts are distinctly named. If, however, `observe_attributes(pik:Pi)` and `observe_attributes(pjℓ:Pj)`, for any two distinct part sorts, P_i and P_j, of a domain, “discovers” identically named attributes, say A, then we say that parts p_i:P_i and p_j:P_j **‘share’** attribute A. that is, that `a:attr_A(pi)` (and `a':attr_A(pj)`) is a **‘shared attribute’** (with `a=a'` always ($\square$) holding).

Example 31. Shared Attributes. Examples of shared attributes: (i) Bus timetable attributes have the same value as the regional transport system timetable attribute. (ii) Bus clock attributes have the same value as the regional transport system clock attribute. (iii) Bus owner attributes have the same value as the regional transport system owner attribute. (iv) Bank customer passbooks record bank transactions on, for example, demand/deposit accounts share values with the bank general ledger passbook entries. (v) A link incident upon or emanating from a hub shares the connection between that link and the hub as an attribute. (vi) Two pipeline units²⁶, p_i, p_j, that are connected, such that an outlet π_j of p_i “feeds into” an inlet π_i of p_j, are said to share the connection (modelled by, e.g., {(π_i, π_j)}). ■

Example 32 . Shared Timetables *The fleet and vehicles of Example 9 on page 10 and Example 10 on page 12 is that of a bus company.*

40 *From the fleet and from the vehicles we observe unique identifiers.*

41 *Every bus mereology records the same one unique fleet identifier.*

42 *The fleet mereology records the set of all unique bus identifiers.*

43 *A bus timetable is a share fleet and bus attribute.*

type

40. *FI, VI, BT*

value

40. **uid_F**: $F \rightarrow FI$

40. **uid_V**: $V \rightarrow VI$

41. **mereo_F**: $F \rightarrow VI\text{-set}$ [cf. Sect. 3.9.2 on page 16]

42. **mereo_V**: $V \rightarrow FI$

43. **attr_BT**: $(F|V) \rightarrow BT$

axiom

$\square \forall f:F \Rightarrow$

$\forall v:V \bullet v \in \mathbf{obs_Vs}(\mathbf{obs_VC}(f)) \bullet \mathbf{attr_BT}(f) = \mathbf{attr_BT}(v)$

[which is the same as]

$\square \forall f:F \Rightarrow$

$\{\mathbf{attr_BT}(f)\} = \{\mathbf{attr_BT}(v):v:V \bullet v \in \mathbf{obs_Vs}(\mathbf{obs_VC}(f))\}$ ■

Part attributes of one sort, P_i, may be simple type expressions such as **A-set**, where A may be an attribute of some other part sort, P_j, in which case we say that part attributes **A-set** and A are shared.

Example 33 . Shared Passbooks

44 *A banking system contains*

- *an administration and*
- *a set of customers.*

²⁶See upcoming Example 22 on page 18

- 45 *The administration contains a general ledger.*
- 46 *An attribute of a general ledger is a set of passbooks.*
- 47 *An attribute of a customer is that of a passbook.*
- 48 *Passbooks are uniquely identified by unique customer identifiers.*

154

type44. $[parts] \quad BS, AD, GL, CS, Cs = C\text{-set}$ 47. $[attributes] \quad PB$ **value**44. $\mathbf{obs_AD}: BS \rightarrow AD$ 45. $\mathbf{obs_GL}: AD \rightarrow GL$ 46. $\mathbf{attr_PBs}: GL \rightarrow PB\text{-set}$ 44. $\mathbf{obs_CS}: BS \rightarrow CS$ 44. $\mathbf{obs_Cs}: BS \rightarrow Cs$ 47. $\mathbf{attr_PB}: C \rightarrow PB$ 48. $\mathbf{uid_PB}: PB \rightarrow PBI$ **axiom**

$$\square \forall bs:BS \bullet \\ \mathbf{attr_PBs}(\mathbf{attr_GL}(\mathbf{obs_AD}(bs))) \\ = \{\mathbf{attr_PB}(c) \mid c:C \bullet c \in \mathbf{obs_Cs}(\mathbf{obs_CS}(bs))\} \quad \blacksquare$$

3.10.6 Update of Dynamic Attributes

155

In order to update values of dynamic attributes the description language offers the “built-in” operator:

Attribute Update Function: **upd_attr**

$$\bullet \mathbf{upd_attr}: P \times A \rightarrow P$$

for all relevant A and P. The meaning of **upd_attr** is, informally:

156

type

P, A

value $\mathbf{upd_attr}: P \times A \rightarrow P$ $\mathbf{upd_attr}(p)(a) \text{ as } p'$ **pre:** $\eta A \in \mathbf{attribute_names}(p)$ **post:** $\mathbf{attr_A}(p') = a \wedge \text{no other qualities of } P \text{ other than } A \text{ are affected}$

As for the mereology update operation (Page 19) the above is a simplification. We shall presently leave it with the above! Remarks similar to those given for **upd_mereology** apply (Page 19). We shall presently leave it with the above!

157

158

Example 34 . Hub State Update We continue Example 27 on page 21. Hub states can be considered abstractions of road intersection signal values **red/yellow/green**.

49 We consider **set_hub_state** a primitive operation

a which takes a hub, h , and a hub state, $h\sigma$, as arguments and “delivers the same” hub, h' ,

b with unchanged hub state space (etcetera)

c but with $h\sigma$ being the new value of the $H\Sigma$ attribute of h .

159

value

```

49.  set_hub_state:
49a.   $H \rightarrow H\Sigma \rightarrow H$ 
49.  set_hub_state( $h$ )( $h\sigma$ ) as  $h'$ 
49a.  pre:  $h\sigma \in \mathbf{attr\_H}\Omega(h)$ 
49b.  post:  $h\sigma \in \mathbf{attr\_H}\Omega(h')$ 
49c.   $\wedge h' = \mathbf{upd\_H}\Sigma\text{-of-}H(h)(h\sigma)$  ■

```

3.11 Materials: Continuous Endurants

160

We refer to Page8 for a first coverage of the concept of materials.

Continuous endurants (i.e., '**material**'s) are entities, m , which satisfy:

- $\mathbf{is_material}(m) \equiv \mathbf{is_endurant}(m) \wedge \mathbf{is_continuous}(m)$

Example 35 . Parts and Materials We observe materials as components of parts: Thus liquid or gaseous materials are observed in pipeline units. We can also observe parts immersed in materials: container vessels “floats” on the oceans; aircrafts flies in the air! ■

We shall in this paper not cover the case of parts being immersed in materials²⁷.

We now complement the **observe_part_sorts** (of Sect.3.5.3). We assume, without loss of generality, that only atomic parts may contain materials. Let $p:P$ be some atomic part.

Endurant Analysis Prompt 14 . has_materials: The '**domain analysis prompt**':

- $\mathbf{has_materials}(p)$

yields **true** if a component m of the atomic $p:P$ satisfies $\mathbf{is_material}(m)$, otherwise false ■

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

Domain Description Prompt 6 . observe_material_sorts: The '**domain description prompt**':

- $\mathbf{observe_material_sorts}(e)$

yields the material sorts and material sort observers domain description text according to the followig schema:

6. observe_material_sorts 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] M_i [$1 \leq i \leq n$]

value

- [o] $\mathbf{obs_}M_i: P \rightarrow M_i$ [$1 \leq i \leq n$]
- [i] $\mathbf{is_}M_i: M \rightarrow \mathbf{Bool}$ [$1 \leq i \leq n$]

²⁷Most such cases have the material play a minor, an abstract rôle with respect to the immersed parts. That is, we presently leave it to hydro- and aerodynamics to domain analyse those cases.

proof obligation [*Disjointness of Material Sorts*]

$$[p] \quad \forall m_i:(M_1|M_2|\dots|M_n) \bullet$$

$$[p] \quad \bigwedge \{ \mathbf{is_}M_i(m_i) \equiv \bigvee \sim \{ \mathbf{is_}M_j(m_i) | j \in \{1..m\} \setminus \{i\} \} | i \in \{1..m\} \}$$

The M_i are all distinct ■

164

Example 36 . Pipeline Material We continue Example 16 on page 14 and Example 22 on page 18. Previously we may have left the impression that pipeline units, $u:U$, were atomic. Now we consider them composite all satisfying $\mathbf{has_material}(u)$.

50 When we apply $\mathbf{obs_material_sorts_U}$ to any unit $u:U$ we obtain

- a a type clause stating the material sort $\mathbf{LoG}$ for some further undefined liquid or gaseous material, and
- b a material observer function signature.

type

50a $\mathbf{LoG}$

value

50b $\mathbf{obs_LoG}: U \rightarrow \mathbf{LoG}$ ■

3.11.1 Material Qualities

165

It seems that we do not need to model unique identifier nor mereology qualities of materials²⁸. But materials do have attributes. We extend the usual attribute-related analysis and synthesis prompts ($\mathbf{observe_attributes}$) to apply also to materials.

166

Example 37 . Pipeline Material Attributes We continue Examples 16, 22 and 36. One possible attribute of the liquid material of pipelines are liquid type: *organic oil* or *mineral oil*. For, for example, mineral oil there are many, many petrochemical attributes. We refer to standard work on petrochemistry for details ■

3.11.2 Materials-related Part Attributes

167

It seems that the “interplay” between parts and materials is an area where domain analysis in the sense of this paper is relevant.

168

Example 38 . Pipeline Material Flow We continue Examples 16, 22, 36 and 37. Let us postulate a $[n \text{ attribute}]$ sort $\mathbf{Flow}$. We now wish to examine the flow of liquid (or gaseous) material in pipeline units. We use two types

51 F for “productive” flow, and L for wasteful leak.

Flow and leak is measured, for example, in terms of volume of material per second. We then postulate the following unit attributes “measured” at the point of in- or out-flow or in the interior of a unit.

169

52 current flow of material into a unit input connector,

53 maximum flow of material into a unit input connector while maintaining laminar flow,

54 current flow of material out of a unit output connector,

55 maximum flow of material out of a unit output connector while maintaining laminar flow,

²⁸We might be persuaded to call the isotope marking of a liquid (for the purposes of tracing the sources or sinks of that liquid) a unique identifier.

- 56 *current leak of material at a unit input connector,*
 57 *maximum guaranteed leak of material at a unit input connector,*
 58 *current leak of material at a unit input connector,*
 59 *maximum guaranteed leak of material at a unit input connector,*
 60 *current leak of material from “within” a unit, and*
 61 *maximum guaranteed leak of material from “within” a unit.*

type

51. F, L

value

52. **attr_cur_iF**: $U \rightarrow UI \rightarrow F$
 53. **attr_max_iF**: $U \rightarrow UI \rightarrow F$
 54. **attr_cur_oF**: $U \rightarrow UI \rightarrow F$
 55. **attr_max_oF**: $U \rightarrow UI \rightarrow F$
 56. **attr_cur_iL**: $U \rightarrow UI \rightarrow L$
 57. **attr_max_iL**: $U \rightarrow UI \rightarrow L$
 58. **attr_cur_oL**: $U \rightarrow UI \rightarrow L$
 59. **attr_max_oL**: $U \rightarrow UI \rightarrow L$
 60. **attr_cur_L**: $U \rightarrow L$
 61. **attr_max_L**: $U \rightarrow L$

The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows below which laminar flow can be expected. The current flow attributes are dynamic attributes ■

3.11.3 Laws of Material Flows and Leaks

171

It may be difficult or costly, or both, to ascertain flows and leaks in materials-based domains. But one can certainly speak of these concepts. This casts new light on domain modelling. That is in contrast to incorporating such notions of flows and leaks in requirements modelling where one has to show implementability.

Modelling flows and leaks is important to the modelling of materials-based domains.

Example 39 . Pipelines: Intra Unit Flow and Leak Law

62 *For every unit of a pipeline system, except the well and the sink units, the following law apply.*

63 *The flows into a unit equal*

- a the leak at the inputs*
- b plus the leak within the unit*
- c plus the flows out of the unit*
- d plus the leaks at the outputs.*

axiom [*Well-formedness of Pipeline Systems, PLS (1)*]

62. $\forall pls:PLS, b:B \setminus We \setminus Si, u:U \bullet$
 62. $b \in \mathbf{obs_Bs}(pls) \wedge u = \mathbf{obs_U}(b) \Rightarrow$
 62. **let** $(iuis, ovis) = \mathbf{mereo_U}(u)$ **in**
 63. $\mathbf{sum_cur_iF}(iuis)(u) =$
 63a. $\mathbf{sum_cur_iL}(iuis)(u)$

63b. $\oplus \text{attr_cur_L}(u)$
 63c. $\oplus \text{sum_cur_oF}(ouis)(u)$
 63d. $\oplus \text{sum_cur_oL}(ouis)(u)$
 62. **end**

174

- 64 The `sum_cur_iF` (cf. Item 63) sums current input flows over all input connectors.
 65 The `sum_cur_iL` (cf. Item 63a) sums current input leaks over all input connectors.
 66 The `sum_cur_oF` (cf. Item 63c) sums current output flows over all output connectors.
 67 The `sum_cur_oL` (cf. Item 63d) sums current output leaks over all output connectors.

64. `sum_cur_iF`: $\text{UI-set} \rightarrow U \rightarrow F$
 64. `sum_cur_iF`(*iuis*)(*u*) $\equiv \oplus \{ \text{attr_cur_iF}(\text{ui})(u) \mid \text{ui}:\text{UI} \bullet \text{ui} \in \text{iuis} \}$
 65. `sum_cur_iL`: $\text{UI-set} \rightarrow U \rightarrow L$
 65. `sum_cur_iL`(*iuis*)(*u*) $\equiv \oplus \{ \text{attr_cur_iL}(\text{ui})(u) \mid \text{ui}:\text{UI} \bullet \text{ui} \in \text{iuis} \}$
 66. `sum_cur_oF`: $\text{UI-set} \rightarrow U \rightarrow F$
 66. `sum_cur_oF`(*ouis*)(*u*) $\equiv \oplus \{ \text{attr_cur_oF}(\text{ui})(u) \mid \text{ui}:\text{UI} \bullet \text{ui} \in \text{ouis} \}$
 67. `sum_cur_oL`: $\text{UI-set} \rightarrow U \rightarrow L$
 67. `sum_cur_oL`(*ouis*)(*u*) $\equiv \oplus \{ \text{attr_cur_oL}(\text{ui})(u) \mid \text{ui}:\text{UI} \bullet \text{ui} \in \text{ouis} \}$
 $\oplus: (F|L) \times (F|L) \rightarrow F$

where $\oplus$ is both an infix and a distributed-fix function which adds flows and or leaks ■

175

Example 40 . Pipelines: Inter Unit Flow and Leak Law

68 For every pair of connected units of a pipeline system the following law apply:

- a the flow out of a unit directed at another unit minus the leak at that output connector
 b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

axiom [Well-formedness of Pipeline Systems, PLS (2)]

68. $\forall \text{pls:PLS}, b, b' : B, u, u' : U \bullet$
 68. $\{b, b'\} \subseteq \text{obs_Bs}(\text{pls}) \wedge b \neq b' \wedge u' = \text{obs_U}(b')$
 68. $\wedge \text{let } (iuis, ouis) = \text{mereo_U}(u), (iuis', ouis') = \text{mereo_U}(u'),$
 68. $ui = \text{uid_U}(u), ui' = \text{uid_U}(u') \text{ in}$
 68. $ui \in iuis \wedge ui' \in ouis' \Rightarrow$
 68a. $\text{attr_cur_oF}(u')(ui') - \text{attr_leak_oF}(u')(ui')$
 68b. $= \text{attr_cur_iF}(u)(ui) + \text{attr_leak_iF}(u)(ui)$
 68. **end**
 68. **comment:** *b' precedes b* ■

176

From the above two laws one can prove the **theorem**: what is pumped from the wells equals what is leaked from the systems plus what is output to the sinks. We need formalising the flow and leak summation functions.

3.12 “No Junk, No Confusion”

177

Domain descriptions are, as we have already shown, formulated, both informally and formally, by means of abstract types, that is, by sorts for which no concrete models are usually given. Sorts are made to denote possibly empty, possibly infinite, rarely singleton, sets of entities on the basis of the qualities defined for these sorts, whether external or internal. By ‘**junk**’ we shall understand that the domain description unintentionally denotes undesired entities. By ‘**confusion**’ we shall

178

understand that the domain description unintentionally have two or more identifications of the same entity or type. The question is *can we formulate a [formal] domain description such that it does not denote junk or confusion?* The short answer to this is no! So, since one naturally wishes “no junk, no confusion” what does one do? The answer to that is *one proceeds with great care!* To avoid junk we have stated a number of sort well-formedness axioms, for example:

- Page 16 for *Well-formedness of Links, L, and Hubs, H*,
- Page 17 for *Well-formedness of Domain Mereologies*,
- Page 18 for *Well-formedness of Road Nets, N*,
- Page 18 for *Well-formedness of Pipeline Systems, PLS (0)*,
- Page 22 for *Well-formedness of Hub States, HΣ*,
- Page 28 for *Well-formedness of Pipeline Systems, PLS (1)*,
- Page 29 for *Well-formedness of Pipeline Systems, PLS (2)*,
- Page 30 for *Well-formedness of Pipeline Route Descriptors and*
- Page 31 for *Well-formedness of Pipeline Systems, PLS (3)*.

To avoid confusion we have stated a number of proof obligations:

- Page 10 for *Disjointness of Part Sorts*,
- Page 21 for *Disjointness of Attribute Types and*
- Page 27 for *Disjointness of Material Sorts*.

Example 41 . No Pipeline Junk We continue Example 16 on page 14 and Example 22 on page 18.

69 We define a proper pipeline route to be a sequence of pipeline units.

a such that the i^{th} and $i+1^{st}$ units in sequences longer than 1 are (forward) adjacent, in the sense defined below, and

b such that the route is acyclic, in the sense also defined below.

To formalise the above we describe some auxiliary notions.

Pipe Routes

70 A route descriptor is the sequence of unit identifiers of the units of a route (of a pipeline system).

type

69. $R' = U^\omega$

69. $R = \{ \mid r:Route' \bullet wf_Route(r) \mid \}$

70. $RD = U^*$

axiom [*Well-formedness of Pipeline Route Descriptors, RD*]

70. $\forall rd:RD \bullet \exists r:R \bullet rd = descriptor(r)$

value

70. $descriptor: R \rightarrow RD$

70. $descriptor(r) \equiv \langle uid_UI(r[i]) \mid i: \mathbf{Nat} \bullet 1 \leq i \leq \mathbf{len} \ r \rangle$

- 71 Two units are (forward) adjacent if the output unit identifiers of one shares a unique unit identifier with the input identifiers of the other.

value

```

71.   adjacent:  $U \times U \rightarrow \mathbf{Bool}$ 
71.   adjacent( $u, u'$ )  $\equiv$ 
71.     let ( $ouis$ )= $\mathbf{mereo\_U}(u)$ ,
71.       ( $iuis$ )= $\mathbf{mereo\_U}(u')$  in
71.      $ouis \cap iuis \neq \{\}$  end

```

184

- 72 Given a pipeline system, pls , one can identify the (possibly infinite) set of (possibly infinite) routes of that pipeline system.

- a The empty sequence, $\langle \rangle$, is a route of pls .
- b Let u be a unit of pls , then $\langle u \rangle$ is a route of pls .
- c Let u, u' be adjacent units of pls then $\langle u, u' \rangle$ is a route of pls .
- d If r and r' are routes of pls such that the last element of r is the same as the first element of r' , then $r \hat{\mathbf{tl}} r'$ is a route of pls .
- e No sequence of units is a route unless it follows from a finite number of applications of the basis and induction clauses of Items 72a–72d.

value

```

72.   Routes:  $PLS \rightarrow \mathbf{R-infset}$ 
72.   Routes( $pls$ )  $\equiv$ 
72a.   let  $rs = \langle \rangle$ 
72b.      $\cup \{ \langle u \rangle \mid u:U \bullet u \in \mathbf{obs\_Us}(pls) \}$ 
72c.      $\cup \{ \langle u, u' \rangle \mid u, u':U \bullet \{u, u'\} \subseteq \mathbf{obs\_Us}(pls) \wedge \mathbf{adjacent}(u, u') \}$ 
72d.      $\cup \{ r \hat{\mathbf{tl}} r' \mid r, r':\mathbf{R} \bullet \{r, r'\} \subseteq rs \wedge r[\mathbf{len} \ r] = \mathbf{hd} \ r' \}$ 
72e.   in  $rs$  end

```

185

Well-formed Routes

- 73 A route is acyclic if no two route positions reveal the same unique unit identifier.

value

```

73.   acyclic_Route:  $\mathbf{R} \rightarrow \mathbf{Bool}$ 
73.   acyclic_Route( $r$ )  $\equiv \sim \exists i, j: \mathbf{Nat} \bullet \{i, j\} \subseteq \mathbf{inds} \ r \wedge i \neq j \wedge r[i] = r[j]$ 

```

186

Well-formed Pipeline Systems

- 74 A pipeline system is well-formed if

- a none of its routes are circular and
- b all of its routes embedded in well-to-sink routes.

axiom [Well-formedness of Pipeline Systems, PLS (3)]

```

74.    $\forall pls: PLS \bullet$ 
74a.   non_circular( $pls$ )
74b.    $\wedge$  are_embedded_in_well_to_sink_Routes( $pls$ )

```

value

```

74.   non_circular_PLS:  $PLS \rightarrow \mathbf{Bool}$ 
74.   non_circular_PLS( $pls$ )  $\equiv$ 
74.    $\forall r: \mathbf{R} \bullet r \in \mathbf{routes}(pls) \wedge \mathbf{acyclic\_Route}(r)$ 

```

75 We define well-formedness in terms of well-to-sink routes, i.e., routes which start with a well unit and end with a sink unit.

value

```
75. well_to_sink_Routes: PLS → R-set
75. well_to_sink_Routes(pls) ≡
75.   let rs = Routes(pls) in
75.   {r|R•r ∈ rs ∧ is_We(r[1]) ∧ is_Si(r[len r])} end
```

76 A pipeline system is well-formed if all of its routes are embedded in well-to-sink routes.

```
76. are_embedded_in_well_to_sink_Routes: PLS → Bool
76. are_embedded_in_well_to_sink_Routes(pls) ≡
76.   let wsrs = well_to_sink_Routes(pls) in
76.   ∀ r:R • r ∈ Routes(pls) ⇒
76.     ∃ r':R, i, j: Nat •
76.       r' ∈ wsrs
76.       ∧ {i, j} ⊆ inds r' ∧ i ≤ j
76.       ∧ r = ⟨r'[k] | k: Nat • i ≤ k ≤ j⟩ end
```

Embedded Routes

77 For every route we can define the set of all its embedded routes.

value

```
77. embedded_Routes: R → R-set
77. embedded_Routes(r) ≡
77.   {⟨r[k] | k: Nat • i ≤ k ≤ j⟩ | i, j: Nat • i {i, j} ⊆ inds(r) ∧ i ≤ j}
```

A Theorem

78 The following theorem is conjectured:

- a the set of all routes (of the pipeline system)
- b is the set of all well-to-sink routes (of a pipeline system) and
- c all their embedded routes

theorem:

```
78. ∀ pls:PLS •
78.   let rs = Routes(pls),
78.   wsrs = well_to_sink_Routes(pls) in
78a. rs =
78b.   wsrs ∪
78c.   ∪ {⟨r'|r':R • r' ∈ embedded_Routes(r'')⟩ | r'':R • r'' ∈ wsrs}
77. end ■
```

The above example, besides illustrating one way of coping with “junk”, also illustrated the need for introducing a number of auxiliary notions: types, functions, axioms and theorems.

3.13 Discussion of Endurants

192

In Sect. 3.5.2 on page 9 a “depth-first” search for part sorts was hinted at. It essentially expressed that we discover domains epistemologically but understand them ontologically. The Danish philosopher Søren Kirkegaard (1813–1855) expressed it this way: *Life is lived forwards, but is understood backwards*. The presentation of the of the ‘domain analysis prompt’s and the ‘domain description prompt’s is based on resulting in a domain description which is ontological. The “depth-first” search recognizes the epistemological nature of bringing about understanding. This “depth-first” search that ends with the analysis of atomic part sorts can be guided, i.e., hastened (shortened), by postulating composite sorts that “correspond” to vernacular nouns: everyday nouns that stand for classes of endurants.

193

194

We could have chosen our ‘domain analysis prompt’s and ‘domain description prompt’s to reflect a “bottom-up” epistemology, one that reflected how we composed composite understandings from initially atomic parts. We leave such a collection of ‘domain analysis prompt’s and ‘domain description prompt’s to the reader.

4 Perdurants

195

We shall give only a cursory overview of perdurants. That is, we shall not systematically present a set of ‘domain analysis prompt’s and a set of ‘domain description prompt’s leading to description language, i.e., RSL texts describing perdurant entities.

196

The reason for giving this albeit cursory overview of perdurants is that, through this cursory overview, we can justify our detailed study of endurants, their part and subparts, their unique identifiers, attributes and mereology. This justification is manifested in expressing the types of signatures, in basing behaviours on parts, in basing the for need CSP-oriented inter-behaviour communications on shared part attributes, in indexing behaviours as are parts, i.e., on unique identifiers, and in directing inter-behaviour communications across channel arrays indexed as per the mereology of the part behaviours. These are all notions related to endurants and are now justified by their use in describing perdurants.

197

Perdurants can perhaps best be explained in terms of a notion of state and a notion of time. We shall, in this paper, not detail notions of time, but refer to [50, 37, 26, 96].

4.1 States

198

Definition 9 . State: By a ‘state’ we shall understand any collection of endurants each of which has at least one dynamic attribute.

Example 42 . States Some examples of states are: A road hub can be a state, cf. Hub State, $L\S$, Example 27 on page 21. A road net can be a state – since its hubs can be. Container stowage areas. CSA, Example 11 on page 13, of container vessels and container terminal ports can be states as containers can be removed from and put on top of container stacks ■

4.2 Actions, Events and Behaviours

199

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

4.2.1 Time Considerations

200

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may “occur” during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can

be analysed into “constituent” actions, events and “sub-behaviours”. We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal monographs: Specifying Systems [60, Leslie Lamport] and Duration Calculus: A Formal Approach to Real-Time Systems [102, Zhou ChaoChen and Michael Reichhardt Hansen].

4.2.2 Actors

202

Definition 10 . Actor: *By an ‘actor’ we shall understand something that is capable of initiating and/or carrying out actions, events or behaviours.*

We shall, in principle, associate an actor with each part. These actors will be described as behaviours. These behaviours evolve around a state. The state is the set of qualities, in particular the dynamic attributes, of the associated part.

Example 43 . Actors *We refer to the road transport and the pipeline systems examples of earlier. The fleet, each vehicle and the road management of the Transportation system of Examples 9 on page 10 and 32 on page 24 can be considered actors; so can the net and its links and hubs. The pipeline monitor and each pipeline unit of the Pipeline System, Example 16 on page 14 and Examples 16 on page 14 and 22 on page 18 will be considered actors. The bank general ledger and each bank customer of the Shared Passbooks example, Example 33 on page 24, will be considered actors ■*

4.2.3 Parts, Attributes and Behaviours

204

Example 43 focused on what shall soon become a major relation within domains: that of parts being also considered actors, or more specifically, being also considered to be behaviours.

Example 44 . Parts, Attributes and Behaviours *Consider the term ‘train’²⁹. It has several possible “meanings”. (i) the train as a part, viz., as standing on a train station platform; (ii) the train as listed in a timetable (an attribute of a transport system part), (iii) the train as a behaviour: speeding down the rail track ■*

4.3 Discrete Actions

205

Definition 11 . Discrete Action: *By a ‘discrete action’ [100] we shall understand some foreseeable thing which deliberately, that is, on purpose, 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 ■*

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

Example 45 . Road Net Actions *Examples of road net actions initiated by the net actor are: insertion of hubs, insertion of links, removal of hubs, removal of links, setting of hub states. Examples of traffic system actions initiated by vehicle actors are: moving a vehicle along a link, stopping a vehicle, starting a vehicle, moving a vehicle from a link to a hub and moving a vehicle from a hub to a link ■*

4.4 Discrete Events

207

Definition 12 . Event: *By an ‘event’ we shall understand some unforeseen thing, that is, some ‘not-planned-for’ “action”, one which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular (domain) actor can be made responsible ■*

Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a time or time interval. The notion of event continues to puzzle philosophers [36, 80, 66, 34, 47, 5, 59, 30, 75, 29]. We note, in particular, [34, 5, 59].

²⁹This example is due to Paul Lindgreen, a Danish computer scientist. It dates from the late 1970s.

Example 46 . Road Net and Road Traffic Events Some road net events are: “disappearance” of a hub or a link, failure of a hub state to change properly when so requested, and occurrence of a hub state leading traffic into “wrong-way” links. Some road traffic events are: the crashing of one or more vehicles (whatever ‘crashing’ means), a car moving in the wrong direction of a one-way link, and the clogging of a hub with too many vehicles ■

4.5 Discrete Behaviours

210

Definition 13 . Discrete Behaviour: By a ‘discrete behaviour’ we shall understand a set of sequences of potentially interacting sets of discrete actions, events and behaviours ■

211

Example 47 . Behaviours Examples of behaviours: *Road Nets:* A sequence of hub and link insertions and removals, link disappearances, etc. *Road Traffic:* A sequence of movements of vehicles along links, entering, circling and leaving hubs, crashing of vehicles, etc. *Pipelines:* A sequence of pipeline pump and valve openings and closings, and failures to do so (events), etc. *Container Vessels and Ports:* Concurrent sequences of movements (by cranes) of containers from vessel to port (unloading), with sequences of movements (by cranes) from port to vessel (loading), with dropping of containers by cranes, etcetera ■

4.5.1 Channels and Communication

212

Behaviours usually communicate. Using the CSP [51] notation (adopted by RSL) we introduce and model behaviour communication. Communication is abstracted as the sending ($\text{ch}!m$) and receipt ($\text{ch}?$) of messages, $m:M$, over channels, ch .

type M
channel $\text{ch} M$

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

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

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

4.5.2 Relations Between Attribute Sharing and Channels

213

We shall now interpret the syntactic notion of attribute sharing with the semantic notion of channels. This is in line with the above hinted interpretation of parts with behaviours, and, as we shall soon see part attributes with behaviour states.

214

Thus, for every pair of parts, $p_{ik}:P_i$ and $p_{jl}:P_j$, of distinct sorts, P_i and P_j which share attribute values in A we are going to associate a channel. If there is only one pair of parts, $p_{ik}:P_i$ and $p_{jl}:P_j$, of these sorts, then just a simple channel, say ch_{P_i,P_j} .

channel $\text{ch}_{P_i,P_j}:A$.

If there is only one part, $p_i:P_i$, but a definite set of parts $p_{jk}:P_j$, with shared attributes, then a vector of channels. Let $\{p_{j1}, p_{j2}, \dots, p_{jn}\}$ be all the part of the domain of sort P_j . Then $\text{uids} : \{\text{uid}_{p_{j1}}, \text{uid}_{p_{j2}}, \dots, \text{uid}_{p_{jn}}\}$ is the set of their unique identifiers. Now a schematic channel array declaration can be suggested:

channel $\{\text{ch}[\{\pi_i, \pi_j\}]|\pi_i=\text{uid}_{P_i}(p_i) \wedge \pi_j \in \text{uids}\}:A$.

The above can be extended from channel matrices to channel tensors, etc., hence the term channel ‘array’.

215

Example 48 . Bus System Channels We extend Examples 9 on page 10 and 32 on page 24. We consider the fleet and the vehicles to be behaviours.

79 We assume some transportation system, δ . From that system we observe

80 the fleet and

81 the vehicles.

82 The fleet to vehicle channel array is indexed by the 2-element sets of the unique fleet identifier and the unique vehicle identifiers. We consider *bus timetables* to be the only message communicated between the fleet and the vehicle behaviours.

value

79. $\delta:\Delta,$

80. $f:F = \mathbf{obs_F}(\delta),$

81. $vs:V\text{-set} = \mathbf{obs_Vs}(\mathbf{obs_VC}((\mathbf{obs_F}(\delta))))$

channel

82. $\{fch[\{\mathbf{uid_F}(f), \mathbf{uid_V}(v)\}] | v:V \bullet v \in vs\}:BT \blacksquare$

Example 49 . Bank System Channels We extend Example 33 on page 24. We consider the general ledger and the customers to be behaviours.

83 We assume some bank system. From the bank system

84 we observe the general ledger.

85 and the set of customers.

86 We consider *passbooks* to be the only message communicated between the general ledger and the customer behaviours.

value

83. $bs:BS$

84. $gl=\mathbf{obs_GL}(\mathbf{obs_AD}(bs)):GL$

85. $cs=\mathbf{obs_Cs}(\mathbf{obs_CS}(bs)):C\text{-set}$

channel

86. $\{bsch[\{\mathbf{uid_GL}(gl), \mathbf{uid_C}(c)\}] | c:C \bullet c \in cs\}:PB \blacksquare$

4.6 Continuous Behaviours

218

By a '**continuous behaviour**' we shall understand a continuous time sequence of state changes. We shall not go into what may cause these state changes.

Example 50 . Flow in Pipelines We refer to Examples 22, 36, 38, 39 and 40. Let us assume that oil is the (only) material of the pipeline units. Let us assume that there is a sufficient volume of oil in the pipeline units leading up to a pump. Let us assume that the pipeline units leading from the pump (especially valves and pumps) are all open for oil flow. Whether or not that oil is flowing, if the pump is pumping (with a sufficient *head*³⁰) then there will be oil flowing from the pump outlet into adjacent pipeline units. $\blacksquare$

To describe the flow of material (say in pipelines) requires knowledge about a number of material attributes — not all of which have been covered in the above-mentioned examples. To express flows one resorts to the mathematics of fluid-dynamics using such second order differential equations as first derived by Bernoulli (1700–1782) and Navier–Stokes (1785–1836 and 1819–1903).

³⁰The '**pump head**' is the linear vertical measurement of the maximum height a specific pump can deliver a liquid to the pump outlet.

4.7 Perdurant Signatures and Definitions

221

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

4.7.1 Function Signatures

222

Definition 14 . Function Signature: By a ‘function signature’ we shall understand a function name and a function type expression.

Definition 15 . Function Type Expression: By a ‘function type expression’ we shall understand a pair of type expressions. separated by a function type constructor either $\rightarrow$ (total function) or $\rightarrow$ (partial function).

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

4.7.2 Action Signatures and Definitions

223

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

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

expresses that a selection of the domain as provided by the Σ type expression is acted upon and possibly changed. The partial function type operator $\xrightarrow{\sim}$ shall indicate that $\text{action(v)(}\sigma\text{)}$ may not be defined for the argument, i.e., initial state σ and/or the argument $v:\text{VAL}$, hence the precondition $\mathcal{P}(v, \sigma)$. The postcondition $\mathcal{Q}(v, \sigma, \sigma')$ characterises the “after” state, $\sigma':\Sigma$, with respect to the “before” state, $\sigma:\Sigma$, and possible arguments ($v:\text{VAL}$).

Example 51 . Insert Hub Action Formalisation We formalise aspects of the above-mentioned hub and link actions:

87 Insertion of a hub requires

88 that no hub exists in the net with the unique identifier of the inserted hub,

89 and then results in an updated net with that hub.

value

87. $\text{insert_H: } H \rightarrow N \xrightarrow{\sim} N$

87. $\text{insert_H}(h)(n) \text{ as } n'$

88. **pre:** $\sim \exists h': H \bullet h' \in \text{obs_HS}(\text{obs_HS}(n)) \bullet \text{uid_H}(h) = \text{uid_H}(h')$

89. **post:** $\text{obs_HS}(\text{obs_HS}(n')) = \text{obs_HS}(\text{obs_HS}(n)) \cup \{h\}$ ■

Which could be the argument values, $v:\text{VAL}$, of actions? Well, there can basically be only two kinds of argument values: parts and materials, respectively unique part identifiers, mereologies and attribute values. It basically has to be so since there are no other kinds of values in domains. There can be exceptions to the above (Booleans, natural numbers), but they are rare!

Perdurant (action) analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the “smallest” relevant state³¹, ascribing signatures to action functions, and determining action pre-conditions and action post-conditions. Of these, ascribing signatures

is, perhaps, the most crucial: In the process of determining the action signature one oftentimes discovers that part or material attributes have been left “undiscovered”.

228

Example 52 shows examples of signatures whose arguments are either parts, or parts and unique identifiers, or parts and unique identifiers and attributes.

Example 52 . Some Function Signatures *Inserting a link between two identified hubs in a net:*

value *insert_L*: $L \times (HI \times HI) \rightarrow N \xrightarrow{\sim} N$

Removing a hub and removing a link:

value *remove_H*: $HI \rightarrow N \xrightarrow{\sim} N$
remove_L: $LI \rightarrow N \xrightarrow{\sim} N$

Changing a hub state.

value *change_HΣ*: $HI \times HΣ \rightarrow N \xrightarrow{\sim} N$ ■

4.7.3 Event Signatures and Definitions

229

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

value
event: $\Sigma \times \Sigma \rightarrow \mathbf{Bool}$
event(σ, σ') **as** **true** || **false**
pre: $P(\sigma)$
post: $Q(\sigma, \sigma')$

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

Example 53 . Link Disappearance Formalisation *We formalise aspects of the above-mentioned link disappearance event:*

- 90 *The result net is not well-formed.*
- 91 *For a link to disappear there must be at least one link in the net;*
- 92 *and such a link may disappear such that*
- 93 *it together with the resulting net makes up for the “original” net.*

value
90. *link_diss_event*: $N \times N' \times \mathbf{Bool}$
90. *link_diss_event*(n, n') **as** *tf*
91. **pre**: $\mathbf{obs_Ls}(\mathbf{obs_LS}(n)) \neq \{\}$
92. **post**: $\exists l: L \cdot l \in \mathbf{obs_Ls}(\mathbf{obs_LS}(n)) \Rightarrow$
93. $l \notin \mathbf{obs_Ls}(\mathbf{obs_LS}(n')) \wedge n' \cup \{l\} = \mathbf{obs_Ls}(\mathbf{obs_LS}(n))$ ■

4.7.4 Discrete Behaviour Signatures and Definitions

233

We shall only cover behaviour signatures when expressed in RSL/CSP [44]. The behaviour functions are now called processes. As shown in [17] we can, without loss of generality, associate with each part and sub-part a behaviour; parts which share attributes and are therefore referred to in some parts’ mereology, can communicate (their “sharing”) via channels. A behaviour signature is

³¹Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

therefore:

behaviour: $\pi:\Pi \times p:P \times \text{VAL} \rightarrow \text{out ochs in ichns} \rightarrow \text{process}$

where $\pi:\Pi$ is the unique identifier of part p , i.e., $\pi = \text{uid_P}(p)$, and ochs, ichs are channel expressions, generally of the form

$\text{ochs,ichs: } \{\text{ch}[i] | i:\text{IDE} \bullet \mathcal{P}(i)\}$

where $\mathcal{P}(i)$ is some predicate expression. Let P be a composite sort. Let P be defined in terms of sub-sorts PA, PB, ..., PC. Proces p “derived” from $p:P$, is composed from a process, $\mathcal{M}_P$, relying on and handling the attributes of process p as defined by P operating in parallel with processes $p_a, p_b, \dots, p_c$ where p_a is “derived” from PA, p_b is “derived” from PB, ..., and p_c is “derived” from PC. The domain description “compilation” schematic below “formalises” the above.

Process Schema I

```

value
  p_a:PA=obs_PA(p),
  p_b:PB=obs_PB(p),
  ...,
  p_c:PC=obs_PC(p),
  comp_process(p)  $\equiv$ 
     $p: \pi:\Pi \times p:P \times \text{attrs:P\_ATTRS} \rightarrow$ 
      in,out  $\{\text{ch}[\{\pi, \text{pi}\}] \bullet \text{pi} \in \text{mereo\_P}(p)\}$  process
     $p(\pi:\text{uid\_P}(p), p, \text{attrs}:\text{obs\_attribs}(p)) \equiv$ 
       $\mathcal{M}_P(\pi, p, \text{attrs})$ 
      || comp_process(p_a)
      || comp_process(p_b)
      || ...
      || comp_process(p_c)

```

Let P be a composite sort. Let P be defined in terms of the concrete type **Q-set**. Proces p “derived” from $p:P$, is composed from a process, $\mathcal{M}_P$, relying on and handling the attributes of process p as defined by P operating in parallel with processes $q:\text{obs_Qs}(p)$. The domain description “compilation” schematic below “formalises” the above.

Process Schema II

```

type
  Qs = Q-set
value
  qs:Q-set = obs_Qs(p) in
  comp_process(p)  $\equiv$ 
     $p: \pi:\Pi \times p:P \times \text{attrs:P\_ATTRS} \rightarrow$ 
      in,out  $\{\text{ch}[\{\pi, \text{pi}\}] \bullet \text{pi} \in \text{mereo\_P}(p)\}$  process
     $p(\pi:\text{uid\_P}(p), p, \text{attrs}:\text{obs\_attribs}(p)) \equiv$ 
       $\mathcal{M}_P(\pi, p, \text{attrs})$ 
      || ||  $\{\text{comp\_process}(q) | q:\text{Q} \bullet q \in \text{qs}\}$ 

```

Example 54 . Bus Timetable Coordination We refer to Examples 9 on page 10, 10 on page 12, 32 on page 24 and 48 on page 36.

9_4 δ is the transportation system; f is the fleet part of that system; vs is the set of vehicles of the fleet; bt is the shared bus timetable of the fleet and the vehicles.

95 The fleet process is compiled as per Process Schema II (Page 39)

240

```

type
  Δ, F, VC      [Example 9 on page 10]
  V, Vs = V-set [Example 10 on page 12]
  FI, VI, BT    [Example 32 on page 24]
channel
  {fch...}      [Example 48 on page 36]
value
94.   δ:Δ,
94.   f:F = obs_F(δ),
94.   vs:V-set = obs_Vs(obs_VC(f)),
94.   bt:BT = attr_BT(f)
axiom
94.   ∀ v:V.v ∈ vs ⇒ bt = attr_BT(v) [Example 32 on page 24]
value
95.   fleet: fi:FI×f:F×BT → in,out {fch[ {fi,uid_V(v)} | v:V.v ∈ vs] } process
95.   fleet(fi,f,bt) ≡ M_F(fi,f,bt) || || {vehicle(uid_V(v),v,bt) | v:V.v ∈ vs}
95.   vehicle: vi:VI×v:V×bt:BT → in,out fch[ {fi,vi} ] process
95.   vehicle(vi,v,bt) ≡ M_V(vi,v,bt)

```

241 Fleet and vehicle processes $\mathcal{M}_F$ and $\mathcal{M}_V$ are both “never-ending” processes:

```

value
  M_F: FI×F×BT → in,out {fch[ {fi,uid_V(v)} | v:V.v ∈ vs] } process
  M_F(fi,f,bt) ≡ let bt' = F(fi,f,bt) in M_F(fi,f,bt') end

  M_V: VI×V×BT → in,out fch[ {fi,vi} ] process
  M_V(vi,v,bt) ≡ let bt' = V(vi,v,bt) in M_V(vi,v,bt') end

```

242 The “core” processes, $\mathcal{F}$ and $\mathcal{V}$, are simple actions. In this example we simplify them to change only bus timetables. The expression of actual synchronisation and communication between the fleet and the vehicle processes are contained in $\mathcal{F}$ and $\mathcal{V}$.

```

value
  F: FI×F×BT → in,out {fch[ {fi,uid_V(v)} | v:V.v ∈ vs] } BT
  F(fi,f,bt) ≡ ...

  V: VI×V×BT → in,out fch[ {fi,vi} ] BT
  V(vi,v,bt) ≡ ...

```

243 What the synchronisation and communication between the fleet and the vehicle processes consists of we leave to the reader! ■

Example 55 . Client Bank Transactions We refer to Example 33 on page 24.

```

96 bs is the bank system,
97 gl is the general ledger of the bank administration,
98 pbs is the set of passbooks attribute of the general ledger and
99 cs is the set of bank customers.
100 bank is the overall bank system behaviour.

```

101 *gen_ldgr* is the behaviour of the general ledger, that is, the demand/deposit activities of the bank.

102 *clients* is the overall behaviour of the ensemble of bank demand/deposit [account] customers.

103 *customer* is the behaviour of the individual bank customer. It is here simplified to just the customer behaviour with respect to the demand/deposit account as manifested by the passbook attribute.

The processes are compiled as per Process Schema I (Page 39) – two “compilations” !

244

type

[parts] BS, AD, GL, CS, Cs, C [Example 33 on page 24]
 [attribute] PB [Example 33 on page 24]

value

96. $bs:BS$
 97. $gl:GL = \mathbf{obs_GL}(\mathbf{obs_AD}(bs)), glid:GLI = \mathbf{uid_GL}(gl)$
 98. $pbs:PB\text{-set} = \mathbf{attr_PSs}(gl)$
 99. $cs:C\text{-set} = \mathbf{obs_Cs}(\mathbf{obs_CS}(bs))$

axiom

98. $pbs = \{\mathbf{attr_PS}(c) \mid c:C:0c \in cs\}$ [Example 33 on page 24]

245

value

98. $\mathbf{bank}: bs:BS \rightarrow \mathbf{process}$
 98. $\mathbf{bank}(bs) \equiv \mathbf{gen_ldgr}(glid)(gl)(pbs) \parallel \mathbf{clients}(cs)$
 99. $\mathbf{gen_ldgr}: \pi:GLI \times GL \times PB\text{-set} \rightarrow \mathbf{in,out} \{\mathbf{bch}[\pi, \mathbf{uid_C}(c)] \mid c:C \bullet c \in cs\} \mathbf{process}$
 99. $\mathbf{gen_ldgr}(\pi, gl, pbs) \equiv \mathcal{M}_{GL}(\pi, gl, pbs)$
 100. $\mathbf{clients}: C\text{-set} \rightarrow \mathbf{in,out} \{\mathbf{bch}[\pi, \mathbf{uid_C}(c)] \mid c:C \bullet c \in cs\} \mathbf{process}$
 100. $\mathbf{clients}(cs) \equiv \parallel \{\mathbf{customer}(\mathbf{uid_C}(c), c, \mathbf{attr_PB}(c)) \mid c:C \bullet c \in cs\}$
 101. $\mathbf{customer}: \pi:CI \times C \times PB \rightarrow \mathbf{in,out} \{\mathbf{bsch}[glid, \pi]\} \mathbf{process}$
 101. $\mathbf{customer}(\mathbf{uid_C}(c), c, \mathbf{attr_PB}(c)) \equiv \mathcal{M}_C(\mathbf{uid_C}(c), c, \mathbf{attr_PB}(c))$

246

The $\mathcal{M}_{GL}$ and $\mathcal{M}_C$ behaviours are seen as “never-ending”. We leave their definition to the reader who is expected to model simple deposit, withdraw and inter-account transfer transactions. We have here assumed that each such transactions all lead to update of both the client and the general ledger passbooks ■

4.8 Summary and Discussion of Perdurants

247

4.8.1 Summary

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

4.8.2 Discussion

248

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

5 Conclusion

249

Our last characterisation of the concept of domain can now be formulated: By a '**domain**'³² we shall understand a spatially organised collection of manifest endurants and a temporally understood collection of interacting observable perdurants, including humans. Mathematically we can characterise a '**domain description**' as denoting a heterogeneous algebra of sorts and operations. The sorts model endurants and the operations model actions, events and behaviours.

The (i) construction, (ii) existence, and (iii) propagation & acceptance of domain descriptions serve a number of rôles: (i) a construction (of a domain analysis & description) process rôle is to explore, inquire and theorize about a domain; (ii) an existence rôle is, for the domain description, to be read and discussed by colleagues and domain stake-holders in order to reach agreements, through this social process, that the description is (hopefully) an appropriate model of the domain; (ii) another existence rôle is for a domain description to serve as a basis for the development of various kinds of software: demos, simulators and actual production software [12, 16]; (iii) a propagation & acceptance rôle is for a domain description to become a de facto standard for further propagation of the domain description, including its use in teaching and training — also in the ordinary (say, secondary) school system!

This paper started with a quote from Dostoyevsky's *The Idiot*.

*It's life that matters, nothing but life –
the process of discovering, the everlasting and perpetual process,
not the discovery itself, at all.*³³

I find that quote appropriate in the following, albeit rather mundane, sense: It is the process of analysing and describing a domain that exhilarates me: that causes me to feel very happy and excited. There is beauty [39, E.W. Dijkstra] not only in the result but also in the process.

I shall elaborate on the above a bit further.

MORE TO COME

5.1 On Domain Description Languages

253

We have in this paper expressed the domain descriptions in the RAISE [45] specification language RSL [44]. With what is thought of as basically inessential, editorial changes, one can reformulate these domain description texts in either of Alloy [53] or The B-Method [1] or VDM [23, 24, 40] or Z [101]. We did not go into much detail with respect to perdurants, let alone behaviours. For all the very many domain descriptions, covered elsewhere, RSL (with its CSP sub-language) suffices. But there are cases where we have conjoined our RSL domain descriptions with descriptions in Petri Nets [81] or MSC [52] (Message Sequence Charts) or StateCharts [48]. Since this paper only focused on endurants there was no need, it appears, to get involved in temporal issues. When that becomes necessary, in a study or description of perdurants, then we either deploy DC: The Duration Calculus [102] or TLA+: Temporal Logic of Actions [60].

5.2 A Review of Our Claims: Interpretation and Evaluation

256

We structure this review according to that of Sect.1.2 Pages 4–5.

5.2.1 Domain Description Components

257

Section 3 reveal that we have in mind to describe domains model-theoretically. Usually, in a model-theoretic specification, one specifies “data”, i.e., the type of data, before specifying the operations on these. (In a property-oriented specification, notably in algebraic specifications [86], one postulates the data (e.g., “stacks”) and then characterises these data through the interaction of operations on these data (e.g., “empty”, “push”, “pop”, “top”).)

³² This is our last characterisation of the concept of ‘domain’.

³³ Fyodor Dostoyevsky, *The Idiot*, 1868, Part 3, Sect. V

In order to analyse a domain we must make some simplifying assumptions. A first significant “reduction of domain complexity” is the decision to “divide the world” into two kinds of endurants: manifest discrete and manifest continuous, that is, into parts and materials. The second ‘reduction’ is to allow that parts may contain material(s), **part_has_material**, that is, that these forms of endurants can be clearly separated, or, vice-versa, that **material_has_parts**. 259

The choice as to whether a part contains material(s) or whether a material contains parts is a matter of “style”: whichever way best expresses a “being”; the described domain is the same! 260

The separation of external and internal qualities is justified purely on pragmatic grounds. The issue here is that of separating concerns. Those concerns which are “more-or-less” manifest are “relegated” to ‘external qualities’. Those that are not manifest but can be measured, “more-or-less” by gadgets of physics are “relegated” to ‘internal qualities’. 261

The choice for modelling internal qualities: that is, **unique identifiers**, **mereology** and the **attributes**, is “conventional abstraction”! Unique identification is a postulated “figment of imagination”! They are best modelled as sorts. Mereology can be modelled many ways. Where mereology reflects spatial “adjacency” or “embeddedness” relations, they could be modelled “geometrically” but that would sometimes lead to unnecessarily intricate “models”. So we choose simple forms of expressions over unique identifiers. Attributes are usually simple, “measurable” phenomena. As such they are modelled as we would model simple data types. In modelling domain attributes one usually exerts great restraint. Usually one can associate “zillions” of attributes even with atomic parts. We advocate selecting as few attributes as needed. 263 264

The chosen domain description language, RSL [44], allows us to distinguish between sorts and types. Characteristics of sort values are then given through domain axioms, as expressed in RSL. This is the case in general, it is the very basis for the “built-in” domain observer functions, **obs_P**, **obs_M**, **uid_P**, **mereo_P** and **attr_A**, briefly reviewed below. In this RSL “borrows” more from algebraic specifications, [86], than from model-oriented ones.

5.2.2 Domain Observer Functions

265

The **obs_P**, **obs_M**, **uid_P**, **mereo_P** and **attr_A**, etcetera, literals are emphasized in bold face. They need not have been written in this way; **obs_P**, **obs_M**, **uid_P**, **mereo_P** and **attr_A**, etcetera, would be just as fine. They are postulated functions that “become defined” through the axioms that usually accompany their first domain description text presentation. (Many other functions are postulated and defined through axioms.) We have chosen the **obs_P**, ..., **attr_A** forms in order to emphasize that they are “standard” sort observer functions, ‘standard’ in the sense of ‘common’ to all domain descriptions. We consider this repertoire of ‘standard’ observer functions to be novel.

5.3 Comparison to Other Work

266

Section 3 outlined the **TriPTych** modelling approach to domain endurants. We shall now compare that approach to a number of techniques and tools that are somehow related — if only by the term ‘domain’! The next two subsections, Sects. 5.3.1–5.3.2 overlap.

5.3.1 Ontological Engineering

267

Ontological engineering [6] build ontologies. Ontologies are “*formal representations of a set of concepts within a domain and the relationships between those concepts*” — expressed usually in some logic. Published ontologies usually consists of thousands of logical expressions. These are represented in some, for example, low-level mechanisable form so that they can be interchanged between ontology research groups and processed by various tools. There does not seem to be a concern for “deriving” such ontologies into requirements for software. Usually ontology presentations either start with the presentation of, or makes reference to its reliance on, an **upper ontology**. Ontology databases appear to be used for the computerised discovery and analysis of relations between ontologies. 268 269

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

5.3.2 Knowledge and Knowledge Engineering

271

The concept of knowledge has occupied philosophers since Plato. No common agreement on what ‘knowledge’ is has been reached. From [63, 3, 69, 93] we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.* The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [38]: knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. Knowledge engineering focus on continually building up (acquire) large, shared data bases (i.e., knowledge bases), their continued maintenance, testing the validity of the stored ‘knowledge’, continued experiments with respect to knowledge representation, etcetera. Knowledge engineering can, perhaps, best be understood in contrast to algorithmic engineering: In the latter we seek more-or-less conventional, usually imperative programming language expressions of algorithms whose algorithmic structure embodies the knowledge required to solve the problem being solved by the algorithm. The former seeks to solve problems based on an interpreter inferring possible solutions from logical data. This logical data has three parts: a collection that “mimics” the semantics of, say, the imperative programming language, a collection that formulates the problem, and a collection that constitutes the knowledge particular to the problem. We refer to [25].

• • •

Domain science & engineering is not aimed at letting the computer solve problems based on the knowledge it may have stored. Instead it builds models based on knowledge of the domain.

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

5.3.3 Database Analysis

278

There are different, however related “schools of database analysis”. DSD: the Bachman (or data structure) diagram model [4], RDM: the relational data model [32] and ER: entity set relationship model [31] “schools”. DSD and ER aim at graphically specifying database structures. Codd’s RDM simplifies the data models of DSD and ER while offering two kinds of languages with which to operate on RDM databases: SQL and Relational Algebra. All three “schools” are focused more on data modelling for databases than on domain modelling both enduring and perduring entities.

5.3.4 Domain Analysis

279

Domain analysis, or product line analysis (see below), as it was then conceived in the early 1980s by James Neighbors [71, 72], is the analysis of related software systems in a domain to find their common and variable parts. It is a model of wider business context for the system. This form of domain analysis turns matters “upside-down”: it is the set of software “systems” (or packages) that is subject to some form of inquiry, albeit having some domain in mind, in order to find common features of the software that can be said to represent a named domain.

280

In this section we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Díaz’s approach [77, 78, 79]. Firstly, our understanding of domain analysis basically coincides with Prieto-Díaz’s. Secondly, in, for example, [77], Prieto-Díaz’s domain analysis is focused on the very important stages that precede the kind of domain modelling that we have described: major concerns are selection of what appears to be similar, but specific entities, identification of common features, abstraction of entities and classification. Selection and identification is assumed in our approach, but we suggest to follow the ideas of Prieto-Díaz. Abstraction (from values to types and signatures) and classification into parts, materials, actions, events and behaviours is what we have focused on. All-in-all we find Prieto-Díaz’s work very relevant to our work: relating to it by providing guidance to pre-modelling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Díaz does mention a need for domain specific languages, he does not show examples of domain descriptions in such DSLs. We, of course, basically use mathematics as the DSL. In our approach we do not consider requirements, let alone software components, as do Prieto-Díaz, but we find that that is not an important issue.

281

5.3.5 Domain Specific Languages

282

Martin Fowler³⁴ defines a *Domain-specific language* (DSL) as a *computer programming language of limited expressiveness focused on a particular domain* [41]. Other references are [68, 91]. Common to [91, 68, 41] is that they define a domain in terms of classes of software packages; that they never really “derive” the DSL from a description of the domain; and that they certainly do not describe the domain in terms of that DSL, for example, by formalising the DSL.

5.3.6 Feature-oriented Domain Analysis (FODA)

283

Feature oriented domain analysis (FODA) is a domain analysis method which introduced feature modelling to domain engineering FODA was developed in 1990 following several U.S. Government research projects. Its concepts have been regarded as critically advancing software engineering and software reuse. The US Government supported report [58] states: “FODA is a necessary first step” for software reuse. To the extent that TripTych domain engineering with its subsequent requirements engineering indeed encourages reuse at all levels: domain descriptions and requirements prescription, we can only agree. Another source on FODA is [33]. Since FODA “leans” quite heavily on ‘Software Product Line Engineering’ our remarks in that section, next, apply equally well here.

284

5.3.7 Software Product Line Engineering

285

Software product line engineering, earlier known as domain engineering, is the entire process of reusing domain knowledge in the production of new software systems. Key concerns of software product line engineering are reuse, the building of repositories of reusable software components, and domain specific languages with which to more-or-less automatically build software based on reusable software components. These are not the primary concerns of TripTych domain science & engineering. But they do become concerns as we move from domain descriptions to requirements prescriptions. But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is TripTych domain engineering. It seems that

286

³⁴<http://martinfowler.com/dsl.h>

software product line engineering is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [16] puts the ideas of software product lines and model-oriented software development in the context of the TripTych approach.

5.3.8 Problem Frames

287

The concept of problem frames is covered in [56]. Jackson's prescription for software development focus on the "triple development" of descriptions of the problem world, the requirements and the machine (i.e., the hardware and software) to be built. Here domain analysis means, the same as for us, the problem world analysis. In the problem frame approach the software developer plays three, that is, all the TripTych rôles: domain engineer, requirements engineer and software engineer, "all at the same time", iterating between these rôles repeatedly. So, perhaps belabouring the point, domain engineering is done only to the extent needed by the prescription of requirements and the design of software. These, really are minor points. But in "restricting" oneself to consider only those aspects of the domain which are mandated by the requirements prescription and software design one is considering a potentially smaller fragment [54] of the domain than is suggested by the TripTych approach. At the same time one is, however, sure to consider aspects of the domain that might have been overlooked when pursuing domain description development in the "more general" TripTych approach.

5.3.9 Domain Specific Software Architectures (DSSA)

290

It seems that the concept of DSSA was formulated by a group of ARPA³⁵ project "seekers" who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [95]. The [95] definition of domain engineering is *"the process of creating a DSSA: domain analysis and domain modelling followed by creating a software architecture and populating it with software components."* This definition is basically followed also by [70, 87, 65]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the analysis of software components, "per domain", to identify commonalities within application software, and to then base the idea of software architecture on these findings. Thus DSSA turns matter "upside-down" with respect to TripTych requirements development by starting with software components, assuming that these satisfy some requirements, and then suggesting domain specific software built using these components. This is not what we are doing: we suggest that requirements can be "derived" systematically from, and related back, formally to domain descriptions without, in principle, considering software components, whether already existing, or being subsequently developed. Of course, given a domain description it is obvious that one can develop, from it, any number of requirements prescriptions and that these may strongly hint at shared, (to be) implemented software components; but it may also, as well, be the case two or more requirements prescriptions "derived" from the same domain description may share no software components whatsoever! It seems to this author that had the DSSA promoters based their studies and practice on also using formal specifications, at all levels of their study and practice, then some very interesting insights might have arisen.

5.3.10 Domain Driven Design (DDD)

295

Domain-driven design (DDD)³⁶ *"is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts; the premise of domain-driven design is the following: placing the project's primary focus on the core domain and domain logic; basing complex designs on a model; initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem."*³⁷ We have studied some of the DDD literature, mostly only accessible on the Internet, but

³⁵ARPA: The US DoD Advanced Research Projects Agency

³⁶Eric Evans: <http://www.domaindrivendesign.org/>

³⁷http://en.wikipedia.org/wiki/Domain-driven_design

see also [49], and find that it really does not contribute to new insight into domains such as we see them: it is just “plain, good old software engineering cooked up with a new jargon.”

5.3.11 Unified Modelling Language (UML)

297

Three books representative of UML are [27, 82, 57]. The term **domain analysis** appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the **domain**, stands for entities in the domain such as we understand it, or whether it is wrought up, as in several of the ‘approaches’ treated in this section, to wit, Sects. 5.3.3, 5.3.4, 5.3.6, 5.3.8, 5.3.9 and 5.3.10, with either **software design** (as it most often is), or **requirements prescription**. Certainly, in UML, in [27, 82, 57] as well as in most published papers claiming “adherence” to UML, that domain analysis usually is manifested in some UML text which “models” some **requirements** facet. Nothing is necessarily wrong with that, but it is therefore not really the **TripTych** form of **domain analysis** with its concepts of abstract representations of **endurant** and **perdurant**, with its distinctions between **domain** and **requirements**, and with its possibility of “deriving” **requirements** prescriptions from **domain** descriptions. The UML notion of **class diagrams** is worth relating to our structuring of the domain. **Class diagrams** appear to be inspired by [4, Bachman, 1969] and [31, Chen, 1976]. It seems that each part sort — as well as other than part (or material) sorts — deserves a **class diagram** (box), that (assignable) attributes — as well as other non-part (or material) types — are written into the diagram box — as are action signatures — as well as other function signatures. **Class diagram** boxes are line connected with annotations where some annotations are as per the mereology of the part type and the connected part types and others are not part related. The **class diagrams** are said to be object-oriented but it is not clear how objects relate to parts as many are rather implementation-oriented quantities. All this needs looking into a bit more, for those who care.

5.3.12 Requirements Engineering

301

There are in-numerous books and published papers on **requirements engineering**. A seminal one is [97]. I, myself, find [61] full of very useful, non-trivial insight. [35] is seminal in that it brings a number of early contributions and views on **requirements engineering**. Conventional text books, notably [74, 76, 89] all have their “mandatory”, yet conventional coverage of **requirements engineering**. None of them “derive” requirements from domain descriptions, yes, OK, from domains, but since their description is not mandated it is unclear what “the domain” is. Most of them repeatedly refer to **domain analysis** but since a written record of that **domain analysis** is not mandated it is unclear what “domain analysis” really amounts to. Axel van Laamsweerde’s book [97] is remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between **goals** and **requirements**, also formally. Most of the advices given in [61] can beneficially be followed also in **TripTych** requirements development. Neither [97] nor [61] preempts **TripTych** requirements development.

• • •

Summary of Comparisons: It should now be clear from the above that basically only Jackson’s *problem frames* really take the same view of domains and, in essence, basically maintain similar relations between **requirements prescription** and **domain description**. So potential sources of, we should claim, mutual inspiration ought be found in one-another’s work — with, for example, [46, 54], and the present document, being a good starting point.

But none of the referenced works make the distinction between discrete **endurants** (parts) and their qualities, with their further distinctions between **unique identifiers**, **mereology** and **attributes**. And none of them makes the distinction between **parts** and **materials**. Therefore our contribution can include the mapping of parts into behaviours interacting as per the part mereologies as highlighted in the process schemas of Sect. 4.7.4 Pages 39–39.

5.4 What Have We Not Covered ?

306

The concept of domain science & engineering, such as basically formulated in this paper, was covered, in some other form, in [10]. The domain analysis development stage, such as covered in this paper, is, however, new.

5.4.1 Domain Facets

307

A development stage that has not been covered in this paper is that of domain facet description. It was covered in [10] and then further developed in [13]. In [11] we list some research challenges based on that 2007–2008 understanding of domain science & engineering.

By a '**domain facet**' we understand one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain. We consider the following facets: **intrinsic**s, support technology, organisation & management, rules & regulations, script and behaviour.

The Intrinsic Facet: By '**intrinsic**s' of a domain we understand those phenomena and concepts of a domain which are basic to any of the other facets (listed above and cursorily described below), with such domain intrinsic's initially covering at least one specific, hence named, stake-holder view.

The Support Technology Facet: By the '**support technology facet**' of a domain we understand ways and means of implementing certain observed phenomena.

The Organisation & Management Facet: By the '**organisation & management facet**' we understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations below) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management, and to floor staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstop" complaints from lower management levels and from floor staff.

The Rules & Regulations Facet: By a '**rule**' of a domain we shall understand some text (in the domain) which prescribes how people or equipment is expected to behave when dispatching their duty, respectively when performing their function. By a '**regulation**' of a domain we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention.

The Script Facet: By the '**script facet**' we understand the structured, almost, if not outright, formally expressed, wording of somehow structured collection of rules or regulations that has legally binding power, that is, which may be contested in a court of law.

The Behaviour Facet: By the '**human behaviour facet**' we understand any of a quality spectrum of carrying out assigned work: from *careful*, *diligent* and *accurate*, via *sloppy* dispatch, and *delinquent* work, to outright *criminal* pursuit. By the '**support technology behaviour facet**' we understand the ability of that technology to faithfully carry out the prescriptions laid down in scripts for that technology, or occasionally, repeatedly, or permanently failing to do so.

• • •

The concept of domain facets is one of pragmatics. It cannot be formalised. The borderlines between the entities described through the various facet views are not precise. The list of domain facets is a check-list. The domain analyser & describer may be well served in checking the domain "against" this list. The domain analyser & describer need not structure a domain description

according to the check-list. But doing so may sometimes help the reader of the domain description.

We shall not cover the concept of domain facets further than saying that the presentation of this concept in [13] and [11] need be reviewed in the context of a much sharper understanding now of the concept of domain analysis.

5.4.2 Laws of Domain Description Prompts

318

Domain description prompts result in texts. Typically `observe_part_sorts` applies to a composite part, $p:P$, and yield descriptions of one or more part sorts: $p-1:P_1, p_2:P_2, \dots, p_m:P_m$. Let $p-i:P_i, p_j:P_j, \dots, p_k:P_k$ (of these) be composite. Now `observe_part_sorts(pi)` and `observe_part_sorts(pj)`, etc., can be applied and yield texts *text_i*, respectively *text_j*. A law of domain description prompts now expresses that the order in which the two or more observers is applied is immaterial, that is, they commute. In [15] we made an early exploration of such laws of domain description prompts. More work, see also below, need be done.

5.5 Future Work

319

The previous sections indicated a number of topics that need further study. For example: *Order of Analysis & Description*, *Laws of Domain Description Prompts* and *A Formal Understanding of Domain Facets*. Below we briefly mention some further topics.

5.5.1 Analysis of Perdurants

320

We plan to carry out a study of perdurants, as detailed as that of our study of endurants. The difficulty, as we see it, is the choice of formalisms: whereas the basic formalisms for the expression of endurants and their qualities was type theory and simple functions and predicates, there is no such simple set of formal constructs that can “carry” the expression of behaviours. Besides the textual CSP, [51], there is graphic notations of Petri Nets, [81], Message Sequence Charts, [52], State-charts, [48], and others.

5.5.2 Commensurate Discrete and Continuous Models

321

Section 4.6 on page 36 hinted at co-extensive descriptions of discrete and continuous behaviours, the former in, for example, RSL, the latter in, typically, the calculus mathematics of partial differential equations (PDEs). The problem that arises in this situation is the following: there will be, say variable identifiers, e.g., $x, y, \dots, z$ which in the RSL formalisation has one set of meanings, but which in the PDE “formalisation” has another set of meanings. Current formal specification languages³⁸ do not cope with continuity. Some research is going on. But to substantially cover, for example, the proper description of laminar and turbulent flows in networks (e.g., pipelines, Example 50 on page 36) requires more substantial results.

322

5.5.3 Interplay between Parts and Materials

323

Examples 37 on page 27, 38 on page 27, 39 on page 28, 40 on page 29 and 50 on page 36 revealed but a small fraction of the problems that may arise in connection with modelling the interplay between parts and materials. Subject to proper formal specification language and, for example PDE specification we may expect more interesting laws, as for example those of Examples 39 on page 28, 40 on page 29, and even proof of these as if they were theorems. Formal specifications have focused on verifying properties of requirements and software designs. With co-extensive (i.e., commensurate) formal specifications of both discrete and continuous behaviours we may expect formal specifications to also serve as bases for predictions.

³⁸Alloy [53], Event B [1], RSL [44], VDM-SL [23, 24, 40], Z, etc.

5.5.4 Towards a Mathematical Model of Domain Analysis & Description

324

There are two aspects to a precise description of the '**domain analysis prompt**'s and '**domain description prompt**'s. There is that of describing the individual prompts as if they were "machine instructions" for an albeit strange machine; and there is that of describing the interplay between prompts: the sequencing of '**domain description prompt**'s as determined by the outcome of the '**domain analysis prompt**'s. We have described and formalised the latter in [19, Processes]. And we are in the midst of describing and formalising the former in [18, Prompts].

5.5.5 Domains and Galois Connections

325

Section 2.4 on page 6 very briefly mentioned that formal concepts form Galois Connections. In the seminal [43] a careful study is made of this fact and beautiful examples show the implications for domains. It seems that our examples have all been too simple. They do not easily lead on to the "discovery" of "new" domain concepts from appropriate concept lattices. Further study need be done.

5.5.6 Domain Theories

326

An ultimate goal of domain science & engineering is to prove properties of domains. Well, maybe not properties of domains, but then at least properties of domain descriptions. If one can be convinced that a posited domain description indeed is a faithful description of a domain, then proofs of properties of the domain description are proofs of properties of that domain. Ultimately domain science & engineering must embrace such studies of *laws of domains*. Here is a fertile ground for zillions of Master and PhD theses!

Example 56 . A Law of Train Traffic: *Let a transport net, $n:N$, be that of a railroad system. Hubs are train stations. Links are rail lines between stations. Let a train timetable record train arrivals and train departures from stations. And let such a timetable be modulo some time interval, say typically 24 hours. Now let us (idealistically) assume that actual trains arrive at and depart from train stations according the train timetable and that the train traffic includes all and only such trains as are listed in the train timetable. Now a law of train traffic expresses Over the modulo time interval of a train timetable it is the case that the number of trains arriving at a station minus the number of trains ending their journey at that station plus the number of trains starting their journey at that station equals number of trains departing from that station. ■*

5.5.7 Precise Descriptions of Man-made Domains

329

The focus on the principles, techniques and tools of domain analysis & description has been such domains in which humans play an active rôle. Formal descriptions of domains may serve to prove properties of domains, in other words, to understand better these domains, and to validate requirements derived from such domain descriptions, and thereby to ensure that software derived from such requirements is not only correct, but also meet users expectations. Improved understanding of man-made domains — without necessarily leading to new software — may serve to improve the "business processes" of these domains, make them more palatable for the human actors, make them more efficient wrt. resource-usage.

Descriptions of domains are descriptions of the syntax and semantics of the technical languages used in speaking about and in the domain. The domain analysis required for the design of programming languages is based on computability: mathematical logic and recursive function theory. The domain analysis required for "real-world" domains is not based on computability: that "world" is not computable. Requirements engineering based on domain descriptions is based on deriving computable subsets of refined domain descriptions. The classical theory and practice of programming language semantics and compiler development [8] and [9, Part VII (Chapters 16–19)] can now be further developed into a theory and practice for deriving general software from formal domain descriptions [12].

Physicists study ‘Mother Nature’, the world without us. Domain scientists study man-made part and material based universes with which we interact — the world within and without us. Classical engineering builds on laws of physics to design and construct buildings, chemical compounds, machines and electrical and electronic products. So far software engineers have not expressed software requirements on any precise description of the basis domain. This paper strongly suggests such a possibility.

333

Regardless: it is interesting to also formally describe domains; and, as shown, it can be done.

5.6 Acknowledgements

334

The writing of this paper was begun in mid December 2012 after a PhD lecture tour of China where I presented a previous version of ‘Domain Analysis’. Versions prior to the China PhD lectures were presented in the years 2007–2012:³⁹ Nancy, Graz, Saarland, Edinburgh, St. Andrews, Glasgow, Tokyo, Vienna, Budapest, Uppsala and Paris. The specific impetus to do a complete rewrite derived from my ‘Domain Analysis’ lectures at the University of Bergen, Norway, May 2012 and from remarks and observations made there by Prof. Magne Haveraaen. I especially thank Magne Haveraaen and also my many PhD lecture hosts around Europe, Japan and China for their kind support.

5.7 References

- [1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings and Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, England, 1996 and 2009.
- [2] Aristotle. *Metaphysics, Books I-IX*. Loeb Classical Library No. 271. Harvard University Press, Cambridge, Mass., USA, 1933 (1996).
- [3] R. Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.
- [4] C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.
- [5] A. Badiou. *Being and Event*. Continuum, 2005. (L'être et l'événements, Edition du Seuil, 1988).
- [6] V. Benjamins and D. Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. <http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm>.
- [7] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 3 edition, 1967.
- [8] D. Bjørner. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77* (eds. E. Morlet and D. Ribbens), pages 1–21. European ACM, North-Holland Publ.Co., Amsterdam, 1977.
- [9] D. 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.
- [10] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
- [11] D. Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science* (eds. J.C.P. Woodcock et al.), pages 1–17, Heidelberg, September 2007. Springer.

³⁹You may find a list of places and references to these earlier versions at <http://www2.compute.dtu.dk/~dibj/node4.html>.

- [12] D. 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.
- [13] D. Bjørner. Domain Engineering. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.
- [14] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.
- [15] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.
- [16] D. 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.
- [17] D. Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, May 2013.
- [18] D. Bjørner. Domain Analysis: A Model of Prompts (paper⁴⁰, slides⁴¹). Research Report 2013-6, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Fall 2013.
- [19] D. Bjørner. Domain Analysis: Endurants – a Consolidated Model of Prompts. Research Report, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, May 2013. (A consolidation of [22] and [18].).
- [20] D. Bjørner. Pipelines – a Domain Description⁴². Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
- [21] D. Bjørner. Road Transportation – a Domain Description⁴³. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.
- [22] D. Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014. (paper⁴⁴, slides⁴⁵).
- [23] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
- [24] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
- [25] D. Bjørner and J. F. Nilsson. Algorithmic & Knowledge Based Methods — Do they “Unify” ? In *International Conference on Fifth Generation Computer Systems: FGCS’92*, pages 191–198. ICOT, June 1–5 1992.
- [26] W. D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
- [27] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

⁴⁰<http://www.imm.dtu.dk/~dibj/da-mod-p.pdf>

⁴¹<http://www.imm.dtu.dk/~dibj/da-mod-s.pdf>

⁴²<http://www.imm.dtu.dk/~dibj/pipe-p.pdf>

⁴³<http://www.imm.dtu.dk/~dibj/road-p.pdf>

⁴⁴<http://www.imm.dtu.dk/~dibj/jaist-da.pdf>

⁴⁵<http://www.imm.dtu.dk/~dibj/jaist-s.pdf>

- [28] R. Carnap. *Der Logische Aufbau der Welt*. Weltkreis, Berlin, 1928.
- [29] R. Casati and A. Varzi. Events. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.
- [30] R. Casati and A. C. Varzi, editors. *Events*. Ashgate Publishing Group – Dartmouth Publishing Co. Ltd., Wey Court East, Union Road, Farnham, Surrey, GU9 7PT, United Kingdom, 23 March 1996.
- [31] P. P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.
- [32] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
- [33] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
- [34] D. Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.
- [35] M. Dorfman and R. H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.
- [36] F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. reprinted in [30], pp. 415-428.
- [37] D. J. Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.
- [38] E. A. Feigenbaum and P. McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.
- [39] W. Feijen, A. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is Our Business*, Texts and Monographs in Computer Science, New York, NY, USA, 1990. Springer. A Birthday Salute to Edsger W. Dijkstra.
- [40] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
- [41] M. Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.
- [42] C. Fox. *The Ontology of Language: Properties, Individuals and Discourse*. CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.
- [43] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999. ISBN: 3540627715, 300 pages, Amazon price: US\$ 44.95.
- [44] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1992.
- [45] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1995.
- [46] C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.
- [47] P. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [30], pp. 429-447.

- [48] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [49] D. Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'). <http://pragprog.com/>, 2009.
- [50] M. Heidegger. *Sein und Zeit (Being and Time)*. Oxford University Press, 1927, 1962.
- [51] C. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: <http://www.usingcsp.com/csp-book.pdf> (2004).
- [52] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
- [53] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
- [54] M. Jackson. Program Verification and System Dependability. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78, London, UK, 2010. Springer.
- [55] M. A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.
- [56] M. A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.
- [57] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [58] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Foda: Feature-oriented domain analysis. Feasibility Study CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. <http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm>.
- [59] J. Kim. *Supervenience and Mind*. Cambridge University Press, 1993.
- [60] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., USA, 2002.
- [61] S. Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.
- [62] H. Laycock. Object. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.
- [63] W. Little, H. Fowler, J. Coulson, and C. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.
- [64] E. Luschei. *The Logical Systems of Leśniewski*. North Holland, Amsterdam, The Netherlands, 1962.
- [65] N. Medvidovic and E. Colbert. Domain-Specific Software Architectures (DSSA). Power Point Presentation, found on The Internet, Absolute Software Corp., Inc.: Abs[S/W], 5 March 2004.
- [66] D. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.
- [67] D. H. Mellor and A. Oliver, editors. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.
- [68] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

- [69] Merriam Webster Staff. Online Dictionary: <http://www.m-w.com/home.htm>, 2004. Merriam-Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.
- [70] E. Mettala and M. H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.
- [71] J. M. Neighbors. Software Construction Using Components. Technical Report 169, Dept. of Information and Computer Science, University of California, Irvine, 1980.
- [72] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transaction on Software Engineering*, 10(5), 1984.
- [73] C. S. Peirce. *Reasoning and the Logic of Things, Edited by Kenneth Laine Ketner*. Harvard University Press, 1 Feb 1993. Comprises a series of lectures given in Cambridge, Massachusetts in 1898.
- [74] S. L. Pfleeger. *Software Engineering, Theory and Practice*. Prentice-Hall, 2nd edition, 2001.
- [75] C.-Y. T. Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.
- [76] R. S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw-Hill, 5th edition, 1981–2001.
- [77] R. Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.
- [78] R. Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
- [79] R. Prieto-Díaz and G. Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
- [80] A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.
- [81] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
- [82] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [83] B. Russel. "Preface," *Our Knowledge of the External World*. G. Allen & Unwin, Ltd., London, 1952.
- [84] B. Russell. On Denoting. *Mind*, 14:479–493, 1905.
- [85] B. 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.
- [86] D. Sannella and A. Tarlecki. *Foundations of Algebraic Semantics and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.
- [87] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [88] B. Smith. Ontology and the Logistic Analysis of Reality. In G. Haeffliger and P. M. Simons, editors, *Analytic Phenomenology*. Dordrecht/Boston/London: Kluwer, Padua, Italy, 1993.
- [89] I. Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.
- [90] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pws Pub Co, August 17, 1999. ISBN: 0534949657, 512 pages, Amazon price: US \$ 70.95.

- [91] D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.
- [92] J. Szrednicki and Z. Stachniak, editors. *Leśniewski's Lecture Notes in Logic*. Dordrecht, 1988.
- [93] Staff of Encyclopædia Britannica. Encyclopædia Britannica. Merriam Webster/Brittanica: Access over the Web: <http://www.eb.com:180/>, 1999.
- [94] S. J. Surma, J. T. Szrednicki, D. I. Barnett, and V. F. Rickey, editors. *Stanisław Leśniewski: Collected works (2 Vols.)*. Dordrecht, Boston – New York, 1988.
- [95] W. Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.
- [96] J. van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methodology, and Philosophy of Science (Editor: Jaakko Hintikka)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
- [97] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
- [98] A. Whitehead. *The Concept of Nature*. Cambridge University Press, Cambridge, 1920.
- [99] A. Whitehead. *An Enquiry Concerning the Principles of Natural Knowledge*. Cambridge University Press, Cambridge, 2929.
- [100] G. Wilson and S. Shpall. Action. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.
- [101] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
- [102] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

6 Indexes

335

I have yet to check the indexing.

• Index of Domain Analysis Prompts	56
• Index of Domain Description Prompts	57
• Index of Description Language Observers and “Built-in” Functions	57
• Index of Description Language Text Schemas	57
• Index of Axioms	57
• Index of Proof Obligations	57
• Index of Definitions	58
• Index of Examples	60
• Index of Concepts	61

6.1 Index of Endurant Analysis Prompts

- | | |
|----------------------|----------------------------|
| a. is_ entity, 6 | h. is_ atomic, 8 |
| b. is_ enduring, 7 | i. is_ composite, 8 |
| c. is_ perdurant, 7 | j. observe_ parts, 9 |
| d. is_ discrete, 7 | k. has_ concrete_ type, 11 |
| e. is_ continuous, 8 | l. has_ mereology, 16 |
| f. is_ part, 8 | m. attribute_ names, 20 |
| g. is_ material, 8 | n. has_ materials, 26 |

6.2 Index of Attribute Analysis Prompts

- | | |
|----------------------------------|------------------------------------|
| A. is_ discrete_ attribute, 22 | F. is_ reactive_ attribute, 22 |
| B. is_ continuous_ attribute, 22 | G. is_ active_ attribute, 22 |
| C. is_ static_ attribute, 22 | H. is_ autonomous_ attribute, 22 |
| D. is_ dynamic_ attribute, 22 | I. is_ biddable_ attribute, 22 |
| E. is_ inert_ attribute, 22 | J. is_ programmable_ attribute, 22 |

6.3 Index of Domain Description Prompts

- | | |
|------------------------------------|---------------------------------|
| 1. observe_ part_ sorts, 9 | 4. observe_ mereology, 16 |
| 2. observe_ part_ type, 11 | 5. observe_ attributes, 20 |
| 3. observe_ unique_ identifier, 14 | 6. observe_ material_ sorts, 26 |

6.4 Index of Domain Description Schemas

6.5 Index of Description Language Observers and Functions

- | | |
|-----------------|-----------------------|
| a. obs_ P, 10 | f. upd_ mereology, 18 |
| b. is_ P, 10 | g. attr_ A, 20 |
| c. obs_ T, 11 | h. obs_ attribs, 20 |
| d. uid_ P, 15 | i. upd_ attr, 25 |
| e. mereo_ P, 17 | j. obs_ M, 26 |

6.6 Index of Axioms

- | | |
|---|--|
| Well-formedness of Domain Mereologies, 17 | Well-formedness of Pipeline Systems, PLS (1), 28 |
| Well-formedness of Hub States, $H\Sigma$, 21 | Well-formedness of Pipeline Systems, PLS (2), 29 |
| Well-formedness of Links, L, and Hubs, H, 15 | Well-formedness of Pipeline Systems, PLS (3), 31 |
| Well-formedness of Pipeline Route Descriptors, 30 | Well-formedness of Road Nets, N, 17 |
| Well-formedness of Pipeline Systems, PLS (0), 18 | |

6.7 Index of Proof Obligations

Disjointness of Attribute Types, 21
 Disjointness of Material Sorts, 26

Disjointness of Part Sorts, 10

6.8 Index of Definitions

- abstract
 - type, 9
- action
 - discrete, 34
- active
 - attribute, 22
- Actor, 34
- actor, 34
- analysis
 - domain
 - prompt, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - prompt
 - domain, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
- Atomic
 - part, 8
- Atomic Part, 8
- attribute
 - active, 22
 - autonomous, 22
 - biddable, 22
 - continuous, 22
 - discrete, 22
 - dynamic, 22
 - inert, 22
 - programmable, 22
 - reactive, 22
 - shared, 23
 - static, 22
- autonomous
 - attribute, 22
- behaviour
 - continuous, 36
 - discrete, 35
- biddable
 - attribute, 22
- Composite
 - part, 8
- Composite Part, 8
- concept
 - formal, 6
- concrete
 - type, 9
- confusion, 29
- context
 - formal, 5
- continuous
 - attribute, 22
 - behaviour, 36
 - endurant, 7
- derived, 12
- description
 - domain, 3
 - prompt, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - prompt
 - domain, 4, 11, 14, 16, 20, 26, 33, 48, 49
- discrete
 - action, 34
 - attribute, 22
 - behaviour, 35
 - endurant, 7
- Discrete Action, 34
- Discrete Behaviour, 35
- Discrete Endurant, 8
- domain, 1, 2
 - analysis
 - prompt, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description, 3
 - prompt, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - facet, 47
 - prompt
 - analysis, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description, 4, 11, 14, 16, 20, 26, 33, 48, 49
- dynamic
 - attribute, 22
- Endurant, 7
- endurant, 7
 - continuous, 7
 - discrete, 7
 - external
 - quality, 14
 - internal
 - quality, 14
 - quality
 - external, 14
 - internal, 14
- Entity, 6
- entity, 6
- Event, 34
- event, 34
- expression
 - function

- type, 37
 - type
 - function, 37
- extent, 6
- external
 - endurant
 - quality, 14
 - quality
 - endurant, 14
- facet
 - domain, 47
 - human behaviour, 47
 - intrinsic, 47
 - organisation & management, 47
 - script, 47
 - support technology, 47
 - support technology behaviour, 47
- formal
 - concept, 6
 - context, 5
- function
 - expression
 - type, 37
 - signature, 37
 - type
 - expression, 37
- Function Signature, 37
- Function Type Expression, 37
- has_ concrete_ type
 - prerequisite
 - prompt, 11
 - prompt
 - prerequisite, 11
- has_ mereology
 - prerequisite
 - prompt, 17
 - prompt
 - prerequisite, 17
- head
 - pump, 36
- human behaviour
 - facet, 47
- inert
 - attribute, 22
- intent, 6
- internal
 - endurant
 - quality, 14
 - quality
 - endurant, 14
- intrinsic, 47
- facet, 47
- is_ composite
 - prerequisite
 - prompt, 10
 - prompt
 - prerequisite, 10
- is_ discrete
 - prerequisite
 - prompt, 8
 - prompt
 - prerequisite, 8
- is_ entity
 - prerequisite
 - prompt, 6, 7
 - prompt
 - prerequisite, 6, 7
- junk, 29
- Material, 7
- material, 7, 8, 26
- mereology, 15
 - type, 16
- method, 2
- methodology, 2
- observe_ part_ type
 - prerequisite
 - prompt, 11
 - prompt
 - prerequisite, 11
- ontological engineering, 43
- organisation & management
 - facet, 47
- part, 7, 8
 - Atomic, 8
 - Composite, 8
 - qualities, 14
- Parts, 7
- Perdurant, 7
- perdurant, 7
- phenomenon, 6
- prerequisite
 - has_ concrete_ type
 - prompt, 11
 - has_ mereology
 - prompt, 17
 - is_ composite
 - prompt, 10
 - is_ discrete
 - prompt, 8
 - is_ entity
 - prompt, 6, 7

- observe_ part_ type
 - prompt, 11
- prompt
 - has_ concrete_ type, 11
 - has_ mereology, 17
 - is_ composite, 10
 - is_ discrete, 8
 - is_ entity, 6, 7
 - observe_ part_ type, 11
- programmable
 - attribute, 22
- prompt
 - analysis
 - domain, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description
 - domain, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - domain
 - analysis, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - has_ concrete_ type
 - prerequisite, 11
 - has_ mereology
 - prerequisite, 17
 - is_ composite
 - prerequisite, 10
 - is_ discrete
 - prerequisite, 8
 - is_ entity
 - prerequisite, 6, 7
 - observe_ part_ type
 - prerequisite, 11
 - prerequisite
 - has_ concrete_ type, 11
 - has_ mereology, 17
 - is_ composite, 10
 - is_ discrete, 8
 - is_ entity, 6, 7
 - observe_ part_ type, 11
- pump
 - head, 36
- qualities
 - part, 14
 - semantic, 14
- syntactic, 14
- quality
 - endurant
 - external, 14
 - internal, 14
 - external
 - endurant, 14
 - internal
 - endurant, 14
- reactive
 - attribute, 22
- regulation, 47
- rule, 47
- script
 - facet, 47
- semantic
 - qualities, 14
- share, 23
- shared
 - attribute, 23
- signature
 - function, 37
- sort, 9
- State, 33
- state, 33
- static
 - attribute, 22
- sub-part, 8
- support technology
 - facet, 47
- support technology behaviour
 - facet, 47
- syntactic
 - qualities, 14
- type, 9
 - abstract, 9
 - concrete, 9
 - expression
 - function, 37
 - function
 - expression, 37
 - mereology, 16

6.9 Index of Examples

- 56 A Law of Train Traffic, 49
- 30 Autonomous and Programmable Hub Attributes, 23
- 1 Domains, 1
- 34 Hub State Update, 25
- 28 Inert and Reactive Attributes, 22
- 23 Mereology Update, 18–19
- 41 No Pipeline Junk, 30–32
- 35 Parts and Materials, 26
- 4 Parts, 7

- 37 Pipeline Material Attributes, 27
- 38 Pipeline Material Flow, 27–28
- 36 Pipeline Material, 26–27
- 22 Pipeline Parts Mereology, 17–18
- 40 Pipelines: Inter Unit Flow and Leak Law, 29
- 39 Pipelines: Intra Unit Flow and Leak Law, 28–29
- 27 Road Hub Attributes, 21–22
- 21 Road Net Part Mereologies, 17
- 19 Shared Attribute Mereology, 15
- 31 Shared Attributes, 23
- 33 Shared Passbooks, 24–25
- 32 Shared Timetables, 24
- 29 Static and Dynamic Link Attributes, 22–23
- 20 Topological Connectedness Mereology, 16
- A Law of Train Traffic (# 56), 49
- Autonomous and Programmable Hub Attributes (# 30), 23
- Domains (# 1), 1
- Hub State Update (# 34), 25
- Inert and Reactive Attributes (# 28), 22
- Mereology Update (# 23), 18–19
- No Pipeline Junk (# 41), 30–32
- Parts (# 4), 7
- Parts and Materials (# 35), 26
- Pipeline Material (# 36), 26–27
- Pipeline Material Attributes (# 37), 27
- Pipeline Material Flow (# 38), 27–28
- Pipeline Parts Mereology (# 22), 17–18
- Pipelines: Inter Unit Flow and Leak Law (# 40), 29
- Pipelines: Intra Unit Flow and Leak Law (# 39), 28–29
- Road Hub Attributes (# 27), 21–22
- Road Net Part Mereologies (# 21), 17
- Shared Attribute Mereology (# 19), 15
- Shared Attributes (# 31), 23
- Shared Passbooks (# 33), 24–25
- Shared Timetables (# 32), 24
- Static and Dynamic Link Attributes (# 29), 22–23
- Topological Connectedness Mereology (# 20), 16

6.10 Index of Concepts

- abstract
 - type, 4
 - value, 14
- abstraction, 6
- action, 2, 3, 33
- algorithmic
 - engineering, 43
- analyser
 - domain, 2, 3
- analysis
 - development stage
 - domain, 46
 - domain, 2, 3, 6, 44–46
 - development stage, 46
 - prompt, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - problem
 - world, 45
 - product line, 44
 - prompt
 - domain, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - world
 - problem, 45
- architecture
 - software, 45
- atomic, 2, 3
- atomicity, 4
- attribute, 4
- axiom
 - sort
 - well-formedness, 29
 - well-formedness
 - sort, 29
- bases
 - knowledge, 43
- behaviour, 2, 3, 33
- change
 - state, 36
- class
 - diagram, 46
- component
 - reusable
 - software, 45
 - software, 45
 - reusable, 45
- composite, 2, 3
 - part, 4
- compositionality, 4
- conceive, 6
- concept

- formal, 6
- concrete
 - part
 - type, 5
 - type
 - part, 5
- confusion, 30
- constructor
 - function
 - type, 37
 - type
 - function, 37
- context, 6
- continuous, 2, 3
 - endurant, 7
 - time, 36
- criminal human behaviour, 47
- delinquent human behaviour, 47
- demo
 - domain, 2
- describer, 2
 - domain, 2, 3
- description
 - development
 - domain, 45
 - domain, 2, 3, 44–46
 - development, 45
 - facet, 47
 - prompt, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - facet
 - domain, 47
 - prompt
 - domain, 4, 11, 14, 16, 20, 26, 33, 48, 49
- descriptions
 - domain, 45, 46
- design
 - software, 2, 45, 46
- development
 - description
 - domain, 45
 - domain
 - description, 45
 - model-oriented
 - software, 45
 - requirements, 45
 - software, 2
 - model-oriented, 45
- development stage
 - analysis
 - domain, 46
 - domain
 - analysis, 46
- diagram
 - class, 46
- diligent human behaviour, 47
- discrete, 2, 3
 - endurant, 7, 8
- domain, 46
 - analyser, 2, 3
 - analysis, 2, 3, 6, 44–46
 - development stage, 46
 - prompt, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - demo, 2
 - describer, 2, 3
 - description, 2, 3, 44–46
 - development, 45
 - facet, 47
 - prompt, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - descriptions, 45, 46
 - development
 - description, 45
 - development stage
 - analysis, 46
 - engineer, 2, 3, 45
 - engineering, 2, 3, 44, 45
 - facet
 - description, 47
 - language
 - specific, 44, 45
 - modelling, 28, 44, 45
 - prompt
 - analysis, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - regulation, 47
 - rule, 47
 - science, 3
 - scientist, 3
 - simulator, 2
 - software
 - specific, 2, 45
 - specific
 - language, 44, 45
 - software, 2, 45
 - stake-holder, 42
- endurant, 2, 3
 - discrete, 8
 - entities, 4
 - external
 - quality, 14
 - internal
 - quality, 14
 - quality
 - external, 14
 - internal, 14
- engineer

- domain, 2, 3, 45
- requirements, 45
- software, 3, 45
- engineering
 - algorithmic, 43
 - domain, 2, 3, 44, 45
 - knowledge, 43
 - ontological, 44
 - product line
 - software, 45
 - requirements, 2, 44
 - software
 - product line, 45
- entities, 2, 3
 - endurant, 4
- entity, 2
- event, 2, 3, 33
- expression
 - function
 - type, 37
 - type, 37
 - function, 37
- external
 - endurant
 - quality, 14
 - qualities, 1
 - quality
 - endurant, 14
- facet
 - description
 - domain, 47
 - domain
 - description, 47
- formal
 - concept, 6
 - text, 4
- formal concept analysis, 6
- frame
 - problem, 45
- frames
 - problem, 45
- function
 - constructor
 - type, 37
 - expression
 - type, 37
 - name, 37
 - type
 - constructor, 37
 - expression, 37
- hardware, 45
- has_ concrete_ type
 - prerequisite
 - prompt, 11
 - prompt
 - prerequisite, 11
- has_ mereology
 - prerequisite
 - prompt, 17
 - prompt
 - prerequisite, 17
- head, 36
- highlighted terms, 2
- human behaviour
 - criminal, 47
 - delinquent, 47
 - diligent, 47
 - sloppy, 47
- identifier
 - unique, 14
- identity, 4
- imperative
 - language
 - programming, 43
 - programming
 - language, 43
- internal
 - endurant
 - quality, 14
 - qualities, 1
 - quality
 - endurant, 14
- interval
 - time, 34
- is_ composite
 - prerequisite
 - prompt, 10
 - prompt
 - prerequisite, 10
- is_ discrete
 - prerequisite
 - prompt, 8
 - prompt
 - prerequisite, 8
- is_ entity
 - prerequisite
 - prompt, 6, 7
 - prompt
 - prerequisite, 6, 7
- join
 - lattice, 13
- junk, 29
- knowledge

- bases, 43
 - engineering, 43
 - representation, 43
- language
 - domain
 - specific, 44, 45
 - imperative
 - programming, 43
 - programming
 - imperative, 43
 - specific
 - domain, 44, 45
- lattice
 - join, 13
- machine, 45
- manifest
 - phenomena, 1
- material, 2, 4
- mereology, 4
 - observer, 16
 - type, 16
- method, 2
- methodology, 2
- model-oriented
 - development
 - software, 45
 - software
 - development, 45
- modelling
 - domain, 28, 44, 45
 - requirements, 28
- name
 - function, 37
- narrative
 - text, 4
- non-manifest
 - qualities, 1
- obligation
 - proof, 30
- observable
 - phenomena, 2
 - phenomenon, 2
- observe, 6
- observe_ part_ type
 - prerequisite
 - prompt, 11
 - prompt
 - prerequisite, 11
- observer
 - mereology, 16
- ontological
 - engineering, 44
- ontology
 - upper, 43, 44
- part, 2, 4, 8
 - composite, 4
 - concrete
 - type, 5
 - sort, 9
 - type
 - concrete, 5
- perdurant, 2, 3
- phenomena
 - manifest, 1
 - observable, 2
- phenomenon
 - observable, 2
- prerequisite
 - has_ concrete_ type
 - prompt, 11
 - has_ mereology
 - prompt, 17
 - is_ composite
 - prompt, 10
 - is_ discrete
 - prompt, 8
 - is_ entity
 - prompt, 6, 7
 - observe_ part_ type
 - prompt, 11
 - prompt
 - has_ concrete_ type, 11
 - has_ mereology, 17
 - is_ composite, 10
 - is_ discrete, 8
 - is_ entity, 6, 7
 - observe_ part_ type, 11
- prescription
 - requirements, 2, 44–46
- problem
 - analysis
 - world, 45
 - frame, 45
 - frames, 45
 - world, 45
 - analysis, 45
- process
 - schema, 46
- product line
 - analysis, 44
 - engineering
 - software, 45
 - software, 45

- engineering, 45
- programming
 - imperative
 - language, 43
 - language
 - imperative, 43
- prompt, 3–4
 - analysis
 - domain, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description
 - domain, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - domain
 - analysis, 4, 6–9, 11, 16, 20, 26, 33, 48, 49
 - description, 4, 11, 14, 16, 20, 26, 33, 48, 49
 - has_ concrete_ type
 - prerequisite, 11
 - has_ mereology
 - prerequisite, 17
 - is_ composite
 - prerequisite, 10
 - is_ discrete
 - prerequisite, 8
 - is_ entity
 - prerequisite, 6, 7
 - observe_ part_ type
 - prerequisite, 11
 - prerequisite
 - has_ concrete_ type, 11
 - has_ mereology, 17
 - is_ composite, 10
 - is_ discrete, 8
 - is_ entity, 6, 7
 - observe_ part_ type, 11
- proof
 - obligation, 30
- qualities, 2–4
 - external, 1
 - internal, 1
 - non-manifest, 1
- quality, 2
 - endurant
 - external, 14
 - internal, 14
 - external
 - endurant, 14
 - internal
 - endurant, 14
- regulation
 - domain, 47
- representation
 - knowledge, 43
- requirements, 45, 46
 - development, 45
 - engineer, 45
 - engineering, 2, 44
 - modelling, 28
 - prescription, 2, 44–46
- reusable
 - component
 - software, 45
 - software
 - component, 45
- reuse, 45
- rule
 - domain, 47
- schema
 - process, 46
- science
 - domain, 3
- scientist
 - domain, 3
- sharing, 14
- simulator
 - domain, 2
- sloppy human behaviour, 47
- software, 45
 - architecture, 45
 - component, 45
 - reusable, 45
 - design, 2, 45, 46
 - development, 2
 - model-oriented, 45
 - domain
 - specific, 2, 45
 - engineer, 3, 45
 - engineering
 - product line, 45
 - model-oriented
 - development, 45
 - product line, 45
 - engineering, 45
 - reusable
 - component, 45
 - specific
 - domain, 2, 45
- sort, 4, 6
 - axiom
 - well-formedness, 29
 - part, 9
 - well-formedness
 - axiom, 29
- specific
 - domain
 - language, 44, 45

- software, 2, 45
- language
 - domain, 44, 45
- software
 - domain, 2, 45
- stake-holder
 - domain, 42
- state, 33
 - change, 36
- sub-part, 8
- subpart, 4
- text
 - formal, 4
 - narrative, 4
- time, 33, 34
 - continuous, 36
 - interval, 34
- TripTych, 2, 6, 43–46
- type, 6
 - abstract, 4
 - concrete
 - part, 5
 - constructor
 - function, 37
 - expression, 37
 - function, 37

- function
 - constructor, 37
 - expression, 37
- mereology, 16
- part
 - concrete, 5
- Unified Modelling Language
 - UML, 46
- unique
 - identifier, 14
- upper
 - ontology, 43, 44
- value
 - abstract, 14
- well-formedness
 - axiom
 - sort, 29
 - sort
 - axiom, 29
- world
 - analysis
 - problem, 45
 - problem, 45
 - analysis, 45