Domain Analysis

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

August 9, 2013: 10:25 am

Abstract

We suggest a new approach to *domain analysis*. A domain is seen as a mapping from *entities* to *qualities*, that is, a mapping from manifest phenomena to usually non-manifest qualities. A type is here considered a set of all those *entities* that all have exactly the same *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 **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, in this paper, focus on the analysis of *endurant entities*, leaving a deeper study of *perdurant entities* to another paper. There will, however, be a section (Sect. **??**) on *perdurant entities*. It has one contribution and otherwise serves to illustrate the use of parts, materials, unique identifiers, attributes and mereology in describing the signatures of actions, events and behaviours. The contribution of the perdurant section is that of relating parts to behaviours, specifically in showing basic behavioural (viz., CSP process) schemas, that is, to relate mereology to process structures. 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*.

The study behind this paper represents an ambitious undertaking: Within the relatively short span of some 49 pages[1] 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.**

[1] Almost 20 of the paper's 49 pages are examples.

1 Introduction

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

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

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

1.1.1 Domains and Entities

By a **'domain'** we shall here understand an area of human activity characterised by observable phenomena[2]: entities, whether endurants (manifest parts and materials) or perdurants (actions, events and behaviours), whether discrete or continuous; and of their qualities.

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

1.1.2 The TripTych Approach to Software Engineering

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 and domain description. References [11, 13, 14] show how to "refine" domain descriptions into requirements prescriptions, and more general relations between domain descriptions and domain demos, domain simulators and more general domain specific software.

1.1.3 Towards a Methodology of Domain Description

By a **'domain description'** we shall understand a text which describes the entities of the domain: whether endurant or perdurant, whether discrete or continuous, whether atomic or composite; as well as the qualities of these entities focusing in this paper on discrete endurants, i.e., parts and possible subparts: their unique identification, their mereology, and their attributes.

Practicalities of Domain Description How does one go about 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 &[3] domain engineering A derivative purpose of this paper is to unveil aspects of domain science & domain engineering. The

[2] PHILOSOPHICAL NOTE: As to what characterises an observable phenomenon we shall not speculate. This would entail a longer discourse [2, 72, 63, 84, 73, 83, 71, 25, 76, 60, 37, 77, 56, 85, 26].

[3] We use the ampersand '&' in order to emphasise that the concepts A and B when "connected" by '&' stands for one, "inseparable" concept.

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 ?*

The Four Domain Analysis "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 description. (i) The *domain* is there. The domain analyser cannot change the domain. Analysing and describing the domain does not change it[4]. 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/engineer[5], 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 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, may, for example, have become wiser from a project to the next. (iv) Finally there is the evolving *domain 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 assume that the "input" domain description is changed[6] and that it helps determine the added text.

An Interactive Domain Analysis 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, say `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, say `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 (DA&DL) The DA&DL thus consists of a number of meta-functions ([I–...]), the prompts. The meta-functions have names (say `is_endurant`) and

[4] PHILOSOPHICAL 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.

[5] Note: At 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.

[6] for example being "stylistically" revised.

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 (DDL) The DDL is RSL [39], the RAISE Specification Language [40]. With suitable, simple adjustments it could also be either of Alloy [48], Event B [1], VDM-SL [20, 21, 35] or Z [86]. We have chosen RSL because of its simple provision for defining sorts, expressing axioms, and postulating observers over sorts.

1.2 Our Contribution — Structure of Paper

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 6.

[0] Domain Engineering as a Separate Activity There is, however, a more important contribution hidden here: it is that of propagating 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. We shall not claim this contribution as we do not, in this paper, substantiate such "free-standing" uses of domain analysis & description

[1] Domain Description Components 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.

[2] Domain Observer Functions 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 = Type_Expr). That is, firstly the focus in 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* ($\mathcal{E}(\iota P_1,\iota P_2,...,\iota P_m)$)[7], and (b_3) the notion of *part attribute types* (say A, B, ..., C) and *part attribute type observers* (**attr_**A, **attr_**B, ..., **attr_**C). That is, secondly the focus in internal qualities of endurants.

[3] The Domain Analysis & Description Prompts 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_unique_identifier`, `obs_mereology`, `attribute_names` and `obs_attributes`.

• • •

Section 13.2 reviews the above claims.

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

2 Formal Concept Analysis

2.1 A Formalisation

This section is a transcription of Ganter & Wille's [38] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.

Definition: 1 Formal Context: A *'formal context'* $\mathbb{K} := (\mathbb{ES}, \mathbb{I}, \mathbb{QS})$ consists of two sets; $\mathbb{ES}$ of entities and $\mathbb{QS}$ 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* $\mathbb{Q}$*"*.
Example endurant 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. One can also list perdurant entities. Example endurant entity qualities are has mobility, has velocity ($\geq$0), has acceleration ($\geq$0), has length ($>$0), has location, has traffic state, etcetera. One can also list perdurant entity qualities.

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

$\mathcal{DQ} : \mathcal{E}\textbf{-set} \to (\mathcal{E}\textbf{-set} \times \mathcal{I} \times \mathcal{Q}\textbf{-set}) \to \mathcal{Q}\textbf{-set}$
$\mathcal{DQ}(s\mathbb{ES})(\mathbb{ES}, \mathbb{I}, \mathbb{QS}) \equiv \{\mathbb{Q} \mid \mathbb{Q}{:}\mathcal{Q}, \mathbb{E}{:}\mathcal{E} \bullet \mathbb{E} \in s\mathbb{ES} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\}$
pre: $s\mathbb{ES} \subseteq \mathbb{ES}$

"the set of qualities common to entities in $s\mathbb{ES}$*"*. ■

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

$\mathcal{DE}$: $\mathcal{Q}\textbf{-set} \to (\mathcal{E}\textbf{-set} \times \mathcal{I} \times \mathcal{Q}\textbf{-set}) \to \mathcal{E}\textbf{-set}$
$\mathcal{DE}(s\mathbb{QS})(\mathbb{ES}, \mathbb{I}, \mathbb{QS}) \equiv \{\mathbb{E} \mid \mathbb{E}{:}\mathcal{E}, \mathbb{Q}{:}\mathcal{Q} \bullet \mathbb{Q} \in s\mathbb{Q} \wedge \mathbb{E} \cdot \mathbb{I} \cdot \mathbb{Q}\},$
pre: $s\mathbb{QS} \subseteq \mathbb{QS}$

"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
  - $\mathcal{DQ}(s\mathbb{E})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{Q}$ and
  - $\mathcal{DE}(s\mathbb{Q})(\mathbb{E}, \mathbb{I}, \mathbb{Q}) = s\mathbb{E}$;
- $s\mathbb{Q}$ is called the *'intent'* of $\mathbb{K}$ and $s\mathbb{E}$ is called the *'extent'* of $\mathbb{K}$. ■

2.2 Types Are Formal Concepts

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

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

that is, refutations to the claim. 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

The formal concepts of a domain form Galois Connections. We must 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

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* ■

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

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* ■

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

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.*

Example 3 Road Traffic 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).* ■

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

Now follows the main sections of this chapter.

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.*

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

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* ■

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.* ■

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* ■

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.* ■

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 25 will cover parts containing materials in a bit more detail.

3.4 Atomic and Composite Parts

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* ■

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

3.5.1 Types and Sorts

We use the term 'sort' when we wish to speak of an abstract type [74], that is, a type for which we have no model[8]. We shall use the term 'type' to cover both abstract types and concrete types.

3.5.2 On Discovering Part Sorts

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_i)

directs the domain analyser to observe the subparts of p_i; let us say they are: $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$ ■

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

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_{i_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_{i_k}s.

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

3.5.3 Part Sort Observer Functions

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

Domain Description Prompt 1 *observe_part_sortsobserve-part-sortsIf* `is_composite`(p) *holds, then the analyser "applies" the description language observer prompt*

- `observe_part_sorts`(p) [a.]

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, ..., P_m
value
[o] **obs_**P_i: $P \to P_i$ [$1{\leq}i{\leq}m$]
[i] **is_**P_i: $P_i \to$ **Bool** [$1{\leq}i{\leq}m$]

proof obligation [*Disjointness of part sorts*]
[p] $\forall\ p{:}(P_1|P_2|...|P_m)$ •
[p] $\bigwedge\{\textbf{is\_}P_i(p) \equiv \bigvee \sim\{\textbf{is\_}P_j(p) \mid j \in \{1..m\} \setminus \{i\}\} \mid i \in \{1..m\}\}$

`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.

type
1. N, F, M
value

1. **obs_**N:$\Delta\rightarrow N$, **obs_**F:$\Delta\rightarrow F$, **obs_**M:$\Delta\rightarrow M$
type
2. HC, LC
value
2. **obs_**HC:$N\rightarrow HC$, **obs_**LC:$N\rightarrow LC$
type
3. VC
value
3. **obs_**VC:$F\rightarrow VC$

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 [19] covers [road] transportation in some detail. ■

Part sort identifiers P_1, P_2, ..., P_m are distinct and are chosen by the domain describer. So was the composite part sort identifier P. When the domain analyser decides that two or more proper parts, p_i:P_i, p_j:P_j, ..., p_j:P_k of P are really of the same type, for example named P_{ijk} (chosen by the domain describer), then the domain analyser must still name them by the distinct P_i, P_j, ..., P_k and then augment the above description text by:

type
  P_{ijk}
value
  obs_P_{ijk}: $\mathrm{P}_i \rightarrow \mathrm{P}_{ijk}$
  obs_P_{ijk}: $\mathrm{P}_j \rightarrow \mathrm{P}_{ijk}$
  ...
  obs_P_{ijk}: $\mathrm{P}_k \rightarrow \mathrm{P}_{ijk}$

3.5.4 On Discovering Concrete Part Types

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'**:

- *has_concrete_type(p).*

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

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

[b.]
- *observe_part_type*[9]

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

2. observe_part_type schema

Narration:
  [*t*] *... narrative text on types ...*
  [*t*] *... narrative text on types ...*
  [*o*] *... narrative text on type observers ...*

[9] has_concrete_type is a **'prerequisite prompt'** of observe_part_type.

Formalisation:

type
[*t*] *Q, R, ..., S*
[*t*] $T = \mathcal{E}(Q,R,...,S)$
value
[*o*] **obs_***T*: $P \rightarrow T$

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

The type names, T, of the concrete type, as well as those of the auxiliary types, Q, R, ..., 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 9:*

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*-**set**, *Ls* = *L*-**set**
5. *H*, *L*
6. *Vs* = *V*-**set**
7. *V*

value
4. **obs_***Hs:HC*→*H*-**set**, **obs_***Ls:LC*→*L*-**set**
6. **obs_***Vs:VC*→*V*-**set**

■

3.5.5 Forms of Part Types

Usually it is wise to restrict the part type definitions, $\mathsf{T}_i = \mathcal{E}_i(\mathsf{Q},\mathsf{R},...,\mathsf{S})$, to simple type expressions. T=A-**set** or T=A* or $\mathsf{T}=\mathsf{ID} \xrightarrow[m]{} \mathsf{A}$ or $\mathsf{T}=\mathsf{A}_t|\mathsf{B}_t|...|\mathsf{C}_t$ where ID is a sort of unique identifiers, $\mathsf{T}=\mathsf{A}_t|\mathsf{B}_t|...|\mathsf{C}_t$ defines the disjoint types $\mathsf{A}_t{=}{=}\mathsf{mkA}_s(\mathsf{s}{:}\mathsf{A}_s)$, $\mathsf{B}_t{=}{=}\mathsf{mkB}_s(\mathsf{s}{:}\mathsf{B}_s)$, ..., $\mathsf{C}_t{=}{=}\mathsf{mkC}_s(\mathsf{s}{:}\mathsf{C}_s)$, and where A, A_s, B_s, ..., C_s are sorts. Instead of $\mathsf{A}_t{=}{=}\mathsf{mkA}(\mathsf{a}{:}\mathsf{A}_s)$, etc., we may write $\mathsf{A}_t{::}\mathsf{A}_s$ etc.

3.5.6 Part Sort and Type Derivation Chains

Let P be a composite sort. Let P_1, P_2, ..., P_m be the part sorts "discovered" by means of `observe_part_sorts(p)` where p:P. We say that P_1, P_2, ..., 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_k, and, generally, 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

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 — generate 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 ares 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 -m-> V,
  PS, PI, P, Ps = PI -m-> P
value
  obs_VS: CL → VS
  obs_Vs: VS → Vs
  obs_PS: CL → PS
  obs_Ps: CTPS → CTPs
type
  CSA
value
  obs_CSA: V → CSA
  obs_CSA: P → CSA
```

```
type
  BAYS, BI, BAY, Bays=BI -m-> BAY
  ROWS, RI, ROW, Rows=RI -m-> ROW
  STKS, SI, STK, Stks=SI -m-> STK
  C
value
  obs_BAYS: CSA → BAYS,
  obs_Bays: BAYS → Bays
  obs_ROWS: BAY → ROWS,
  obs_Rows: ROWS → Rows
  obs_STKS: ROW → STKS,
  obs_Stks: STKS → Stks
  obs_Stk: STK → C*
```

Note that `observe_part_sorts(v:V)` *and* `observe_part_sorts(p:P)` *both yield* CSA ■

3.5.8 More On Part Sorts and Types

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 9: 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 $\{p_{i_1},p_{i_2},\ldots,p_{i_m}\}$, of $p{:}P$, may themselves be composite.

Example 13. We comment on Example 9 on Page 9: 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 is composed from a definite number of abstract or concrete subparts of distinct types.

Example 14. We comment on Example 9 on Page 9: 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 9: 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* → *U*-**set**

type
17. *U == We | Pi | Pu | Va | Fo | Jo | Si*
18. *We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo:Fork, Jo::Join, Si::Sink*

The experimental research report [18] 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 [6].

Example 17 Derivation Chains *Figure 1 on the following page illustrates two part sort and type derivation chains. based on Examples 9 on Page 9 and 11 on the facing page, respectively. The "–>" of Fig. 1 stands for* $\overrightarrow{m}$ ■

3.6 Syntactic and Semantic Qualities of Endurants

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

We suggest that the internal qualities of parts be analysed into three categories: (i) a category of unique part identifiers, (ii) a category of general attributes and (iii) a category of mereological quantities. 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).

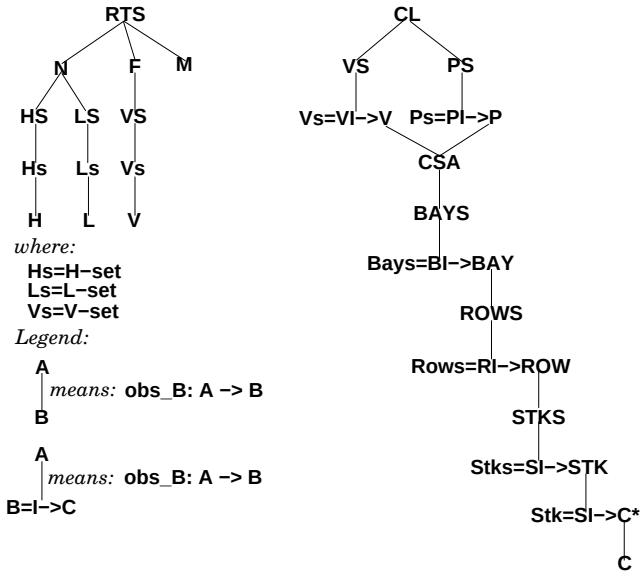

Figure 1: Two Domain Lattices: Examples 9 on Page 9 and 11 on Page 12

3.8 Unique Part Identifiers

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 PI_j, and that all π_i:PI_i and π_j:PI_j are distinct, and that the observer function **uid_**P applied to p yields the unique identifier, say π:PI, of p.

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

[c.]

- *observe_unique_identifier*

to parts p:P resulting in the analyser writing down the unique identifier types and observers domain description text according to the followig schema:[10]

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*] $\mathcal{U}$

[10] **Note:** Do we need to secure disjointness of all unique identifier sorts ?

$\mathcal{U}$ 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. The `has_unique_identifier` *is a* 'prerequisite prompt' *of* `observe_unique_identifier` ■

Example 18 Unique Transportation Net Part Identifiers: *We continue Example 9 on Page 9.*

19. *Links and hubs have unique identifiers*
20. *and unique identifier observers.*

type
19. *LI, HI*
value
20. **uid**_*LI*: $L \to LI$
20. **uid**_*HI*: $H \to HI$
axiom [*Well–formedness of Links, L, and Hubs, H*]
19. $\forall\ l,l':L \bullet l \neq l' \Rightarrow$ **uid**_*LI*$(l) \neq$ **uid**_*LI*(l'),
19. $\forall\ h,h':H \bullet h \neq h' \Rightarrow$ **uid**_*HI*$(h) \neq$ **uid**_*HI*(h')

■

3.9 Discrete Endurant Attributes

To recall: there are three sets of internal (i.e., semantic) endurant 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.

3.9.1 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 touched[11], or seen[12], but can be objectively measured[13]. Thus, in our quest for describing domains where humans play an active rôle, we rule out subjective "attributes": feelings, sentiments, moods. Thus we shall abstain, in our domain science also from matters of aesthetics. We 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)!

3.9.2 Attribute Quality and Attribute Value

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

Example 19 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*

[11] One can see the red colour of a wall, but one touches the wall.
[12] One cannot see electric current, and one may touch an electric wire, but only if it conducts reasonably high voltage can one feel it.
[13] PHILOSOPHICAL 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.

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 will be other components of the n value.* ■

3.9.3 Endurant Attributes: Types and Functions

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 20 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 21 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 25), have attributes. The question is now, in general, how many and, particularly, which.

Endurant Analysis Prompt 12 attribute_names: *The* '**domain analysis prompt**' **attribute_names** *when applied to a part p yields the set of names of its attribute types:*

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

η is a type operator. Applied to a type A it yields is name[14] ■

We cannot automatically, that is, syntactically, guarantee that our domain descriptions secure that the various attribute types for the emerging part sorts 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 ηA∈**attribute_names**(p). It yields the value of attribute A of p.

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

- **obs_attribs**

when applied, **obs_attribs**(e), to an endurant 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, ..., A_n are the types of the n attributes of e then for all i ($1 \leq i \leq n$) **attr_**A_i(e) = **attr_**A_i(**obs_attribs**(e)).

Domain Description Prompt 4 ***observe_attributes*****:** *The domain analyser experiments, thinks and reflects about part (and material, see later) attributes. That process is initated by the* '**domain description prompt**'*:*

[d.] - ***observe_attributes***.

[14] Normally, 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.

The result of that **'domain description prompt'** *writes down the attribute types and observers domain description text according to the followig schema:*

d. `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:$A_i \rightarrow$**Bool** [$1 \leq i \leq n$]
axiom $\forall$ *i*:**Nat** • $1 \leq i \leq n \Rightarrow$ **attr_**$A_i(p) =$ **attr_**A_i(***obs_attribs***(p))
proof obligation [*Disjointness of Attribute Types*]
[*p*] $\forall$ δ:Δ
[*p*] **let** *P be any part sort* **in** *[the* Δ *domain description]*
[*p*] **let** *a*:$(A_1|A_2|...|A_n)$ **in is_**$A_i(a) \neq$ **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, ..., A_n *inform us that the domain analyser has decided to focus on the distinctly named* A_1, A_2, ..., A_n *attributes.*[15] *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* ■

The fact that, for example, A_1, A_2, ..., 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, ..., a_n:A_n respectively.

Example 22 Road Hub Attributes: *After some analysis a domain analyser may arrive at some interesting hub attributes:*

21. *hub state: from which links (by reference) can one reach which links (by reference),*
22. *hub state space: the set of all potential hub states that a hub may attain,*
23. *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.*
24. *Etcetera — i.e., there are other attributes not mentioned here.*

type
21. $H\Sigma = (LI \times LI)$*set*

[15] The attribute type names are not like type names of, for example, a programming language. Instead they are chosen by the domain analyser to reflect on domain phenomena. Cf. Example 20 on the preceding page and Example 21.

22. $H\Omega$ = $H\Sigma$**-set**
value
21. **attr_**$H\Sigma$:$H\rightarrow H\Sigma$
22. **attr_**$H\Omega$:$H\rightarrow H\Omega$
axiom [*Well–formedness of Hub States,* $H\Sigma$]
23. $\forall$ *h*:H • **let** *lis* = **mereo_**H*(h)* **in**
23. **let** $h\sigma$ = **attr_**$H\Sigma$*(h)* **in**
23a.. $\{li,li'|li,li':LI\bullet(li,li')\in h\sigma\}\subseteq lis$
23b.. $\wedge$ $h\sigma \in$ **attr_**$H\Omega$*(h)*
23. **end end**
type
24. ..., ...
value
24. **attr_**..., ...

■

3.9.4 Attribute Categories

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 [50, 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 → 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 23 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 24 Static and Dynamic Link Attributes:

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

    can be considered static,
26. *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,
27. *Finally link attributes*
    a. *link state–of–repair,*
    b. *date last maintained,*

    can be considered inert.

type
25a.. *LEN*
value
25a.. **obs_***LEN: L* → *LEN*
type
25b.. *Name*
value
25b.. **obs_***Name: L* → *Name*
type
26a.. *LΣ′=(HI×HI)***-set**
26a.. *LΣ={|lσ:LΣ* • **card** *lσ* ≤ *2|}*
value
26a.. **obs_***LΣ: L* → *LΣ*
type
26b.. *LΩ′=LΣ***-set**
26b.. *LΩ={|lω:LΩ* • **card** *lω* = *1|}*
value
26b.. **obs_***LΩ: L* → *LΩ*
type
27a.. *LSoR*
27b.. *DLM*
value
27a.. **obs_***LSoR: L* → *LSoR*
27b.. **obs_***DLM: L* → *DLM*

■

Example 25 Autonomous and Programmable Hub Attributes: *We continue Example 24. 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.* ■

3.9.5 Shared Attributes

Normally part attributes of different part sorts are distinctly named. If, however, `observe_attributes`(p$_{ik}$:P$_i$) and `observe_attributes`(p$_{j\ell}$:P$_j$), 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(p$_i$) (and a′:**attr_**A(p$_j$)) is a 'shared attribute' (with a=a′ always (□) holding).

Example 26. 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 units[16], 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., $\{(\pi_i, \pi_j)\}$. ■

Example 27 Shared Timetables: *The fleet and vehicles of Example 9 on Page 9 and Example 10 on Page 11 is that of a bus company.*

28. *From the fleet and from the vehicles we observe unique identifiers.*
29. *Every bus mereology records the same one unique fleet identifier.*
30. *The fleet mereology records the set of all unique bus identifiers.*
31. *A bus timetable is a share fleet and bus attribute.*

type
28. FI, VI, BT
value
28. **uid_***F: F* → *FI*
28. **uid_***V: V* → *VI*
29. **mereo_***F: F* → *VI***-set** *[cf. Sect. 3.10.2 on Page 22]*
30. **mereo_***V: V* → *FI*
31. **attr_***BT: (F|V)* → *BT*
axiom
□ ∀ *f:F* ⇒ ∀ *v:V* • *v* ∈ **obs_***Vs*(**obs_***VC(f)*) • **attr_***BT(f)* = **attr_***BT(v)*
[which is the same as]
□ ∀ *f:F* ⇒ {**attr_***BT(f)*}={**attr_***BT(v)*:*v:V*•*v* ∈ **obs_***Vs*(**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 28 Shared Passbooks:

32. *A banking system contains*
    - *an administration and*
    - *a set of customers.*
33. *The administration contains a general ledger.*
34. *An attribute of a general ledger is a set of passbooks.*
35. *An attribute of a customer is that of a passbook.*
36. *Passbooks are uniquely identified by unique customer identifiers.*

[16] See upcoming Example 33 on Page 24

type
32. [*parts*] *BS, AD, GL, CS, Cs* = *C*-**set**
35. [*attributes*] *PB*
value
32. **obs_***AD: BS* → *AD*
33. **obs_***GL: AD* → *GL*
34. **attr_***PBs: GL* → *PB*-**set**
32. **obs_***CS: BS* → *CS*
32. **obs_***Cs: BS* → *Cs*
35. **attr_***PB: C* → *PB*
axiom
□ ∀ *bs:BS* •
attr_*PBs*(**attr_***GL*(**obs_***AD*(*bs*)))
= {**attr_***PB*(*c*)|*c:C*•*c* ∈ **obs_***Cs*(**obs_***CS*(*bs*))}

■

3.9.6 Update of Dynamic Attributes

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

Attribute Update Function: **upd_attr**

- **upd_attr**: P × A → P

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

type
P, A
value
upd_attr: P × A → P
upd_attr(p)(a) **as** p′
pre: *η*A ∈ `attribute_names`(p)
post: **attr_**A(p′) = a ∧ no other qualities of P than A are affected

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. We shall presently leave it with the above!

Example 29 Hub State Update: *We continue Example 22 on Page 17. Hub states can be considered abstractions of road intersection signals, you know:* **red**/**yellow**/**green**.

37. We consider set_hub_state *are primitive operation*

a. which takes a hub, h, and a hub state, hσ, as arguments
b. and "delivers the same" hub
c. with unchanged hub state space (etcetera)
d. but with hσ being the new value of the HΣ attribute of h.

value
37. *set_hub_state:*

37a.. $H \rightarrow H\Sigma \rightarrow$
37b.. H
37. *set_hub_state(h)(hσ)* **as** h'
37. **pre**: $h\sigma \in$ **attr**_$H\Omega(h)$
37. **post**: $h\sigma \in$ **attr**_$H\Omega(h')$
37c.. $\wedge\ h' = upd\_H\Sigma\_of\_H(h)(h\sigma)$

■

3.10 Mereology

Mereology is the study and knowledge of parts and part relations. Mereology as a logical/philosophical discipline can perhaps best be attributed to the Polish mathematician/logician Stanisław Leśniewski [57, 79, 80].

3.10.1 Part Relations

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

- Two otherwise distinct parts may share attribute values.[17]

  Example 30 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 31 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"!

3.10.2 Part Mereology: Types and Functions

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

type
  P, A, ..., C
value
  obs_A: P → A, ..., **obs**_C: P → C.

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

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

[17] For the concept of attribute value see Sect. 3.9.2 on Page 15.

then the mereology need to be further elaborated.

Endurant Analysis Prompt 13 has_mereology: *To do so the analyser can be said to endow a truth value* **true** *to the* '**domain analysis prompt**'*:*

- *has_mereology*

When the domain analyser decides that some parts are related in a specifically enunciated mereology, 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 $\mathcal{E}$xpression over unique [part] identifier types. We generalise to unique [part] identifier over a definite collection of part sorts, P1, P2, ..., Pn, where the parts p1:P1, p2:P2, ..., pn:Pn are not necessarily (immediate) sub-parts of some part p:P.

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

Domain Description Prompt 5 *observe_mereology*: *If has_mereology(p) holds for parts p of type P, then the analyser can apply the* '**domain description prompt**'*:*

- *observe_mereology* [e.]

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

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

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

Example 32 Road Net Part Mereologies: *We continue Example 9 on Page 9 and Example 18 on Page 15.*

38. *Links are connected to exactly two distinct hubs.*
39. *Hubs are connected to zero or more links.*
40. *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.*

type
38. $LM' = HI\text{-}\mathbf{set}$, $LM = \{|his{:}HI\text{-}\mathbf{set} \bullet \mathbf{card}(his)=2|\}$
39. $HM = LI\text{-}\mathbf{set}$
value
38. **mereo_***L*: $L \to LM$
39. **mereo_***H*: $H \to HM$
axiom [*Well–formedness of Road Nets, N*]
40. $\forall$ *n:N,l:L,h:H*• $l \in$ **obs_***Ls*(**obs_***LC*(*n*))$\wedge h \in$ **obs_***Hs*(**obs_***GC*(*n*))
40. **let** *his*=*mereology_H*(*l*), *lis*=*mereology_H*(*h*) **in**
40. *his*$\subseteq\cup\{$**uid_***H*(*h*) | $h \in$ **obs_***Hs*(**obs_***HC*(*n*))$\}$
40. $\wedge$ *lis*$\subseteq\cup\{$**uid_***H*(*l*) | $l \in$ **obs_***Ls*(**obs_***LC*(*n*))$\}$ **end**

The experimental research report [19] covers well-formedness of road nets. ■

Example 33 Pipeline Parts Mereology: *We continue Example 16 on Page 13. 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.*

41. Wells have exactly one connection to an output unit.

42. Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.

43. Forks have exactly one connection from an input unit and exactly two connections to distinct output units.

44. Joins have exactly one two connection from distinct input units and one connection to an output unit.

45. Sinks have exactly one connection from an input unit.

46. Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

type
46. $UM' = (UI\text{-}\mathbf{set} \times UI\text{-}\mathbf{set})$
46. $UM = \{|(iuis,ouis){:}UI\text{-}\mathbf{set} \times UI\text{-}\mathbf{set} \bullet iuis \cap ouis = \{\}|\}$
value
46. **mereo_***U*: *UM*
axiom [*Well–formedness of Pipeline Systems, PLS (0)*]
$\forall$ *pl:PL,u:U* • $u \in$ **obs_***Us*(*pl*) $\Rightarrow$
let *(iuis,ouis)*=**mereo_***U*(*u*) **in**
case *(***card** *iuis*,**card** *ouis)* **of**
41. *(0,1)* $\to$ **is_***We*(*u*),
42. *(1,1)* $\to$ **is_***Pi*(*u*)$\vee$**is_***Pu*(*u*)$\vee$**is_***Va*(*u*),
43. *(1,2)* $\to$ **is_***Fo*(*u*),
44. *(2,1)* $\to$ **is_***Jo*(*u*),
45. *(1,0)* $\to$ **is_***Si*(*u*)
end end

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

3.10.3 Update of Mereologies

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: **upd_mereology**

- **upd_mereology**: P → M → P

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

```
type
    P, M
value
    upd_mereology: P → M → P
    upd_mereology(p)(m) as p′
        post: mereo_(p′) = m
```

Again, the above is a simplification. Remarks similar to those given for **upd_attr** apply (Page 21).

Example 34 Mereology Update: *The example is that of updating the mereology of a hub. Cf. Example 32 on Page 23.*

47. *Inserting a link, l:L, between two hubs, ha:H,hb:H require the update of the mereologies of these two existing hubs.*
48. *The unique identifier of the inserted link, l:L, is li, li=***uid_***L(l) and h is either ha or hb;*
49. *li is joined to the mereology of either ha or hb; and respective hubs are updated accordingly.*

```
    value
47. update_hub_mereology: H → LI → H
48. update_hub_mereology(h)(li) ≡
49.     let m = {li} ∪ mereo_(h) in upd_mereology(h)(m) end
```

■

3.11 Materials: Continuous Endurants

We refer to Page7 for a first coverage of the concept of materials.

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

- `is_material`$(m) \equiv$ `is_endurant`$(m) \wedge$`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 materials[18].

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

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

- *has_materials(p)*

[18] Most 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.

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

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

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

[f.] • `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
[t] M_i [$1{\leq}i{\leq}n$]
value
[o] **obs_**M_i: $P \to M_i$ [$1{\leq}i{\leq}n$]
[i] **is_**M_i: $M \to$ **Bool** [$1{\leq}i{\leq}n$]

proof obligation [*Disjointness of Material Sorts*]
[p] $\forall\ m_i{:}(M_1|M_2|...|M_n)$ •
[p] $\bigwedge\ \{\mathbf{is\_}M_i(m_i) \equiv \bigvee{\sim}\{\mathbf{is\_}M_j(m_i)|j \in \{1..m\}\backslash\{i\}\}|i \in \{1..m\}\}$

The M_i *are all distinct* ■

Example 36 Pipeline Material: *We continue Example 16 on Page 13 and Example 33 on Page 24.*

50. When we apply `obs_material_sorts_U` *to any unit* $u{:}U$ *we obtain*

a. a type clause stating the material sort LoG *for some further undefined liquid or gaseous material, and*

b. a material observer function signature.

type
50a. *LoG*
value
50b. **obs_***LoG*: $U \to LoG$

■

3.11.1 Material Qualities

It seems that we do not need to model unique identifier nor mereology qualities of materials[19]. But materials do have attributes. We extend the usual attribute-related analysis and synthesis prompts (`observe_attributes`) to apply also to materials.

[19] Note: We 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.

Example 37 Pipeline Material Attributes: *We continue Examples 16, 33 and 36 on the preceding page. 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

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

Example 38 Pipeline Material Flow: *We continue Examples 16, 33, 36 and 37. Let us postulate a[n attribute] sort* 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.

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,*
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 → UI → F*
53. **attr**_*max_iF: U → UI → F*
54. **attr**_*cur_oF: U → UI → F*
55. **attr**_*max_oF: U → UI → F*
56. **attr**_*cur_iL: U → UI → L*
57. **attr**_*max_iL: U → UI → L*
58. **attr**_*cur_oL: U → UI → L*
59. **attr**_*max_oL: U → UI → L*
60. **attr**_*cur_L: U → L*
61. **attr**_*max_L: U → 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

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. ∀ *pls:PLS,b:B\We\Si,u:U* •
62. *b* ∈ **obs_***Bs(pls)*∧*u*=**obs_***U(b)*⇒
62. **let** *(iuis,ouis)* = **mereo_***U(u)* **in**
63. *sum_cur_iF(iuis)(u)* =
63a.. *sum_cur_iL(iuis)(u)*
63b.. ⊕ **attr_***cur_L(u)*
63c.. ⊕ *sum_cur_oF(ouis)(u)*
63d.. ⊕ *sum_cur_oL(ouis)(u)*
62. **end**

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: UI-**set** → U → F
64. sum_cur_iF(iuis)(u) ≡ ⊕ {**attr_**cur_iF(ui)(u)|ui:UI•ui ∈ iuis}
65. sum_cur_iL: UI-**set** → U → L
65. sum_cur_iL(iuis)(u) ≡ ⊕ {**attr_**cur_iL(ui)(u)|ui:UI•ui ∈ iuis}
66. sum_cur_oF: UI-**set** → U → F
66. sum_cur_oF(ouis)(u) ≡ ⊕ {**attr_**cur_iF(ui)(u)|ui:UI•ui ∈ ouis}
67. sum_cur_oL: UI-**set** → U → L
67. sum_cur_oL(ouis)(u) ≡ ⊕ {**attr_**cur_iL(ui)(u)|ui:UI•ui ∈ ouis}
    ⊕: (F|L) × (F|L) → F

where ⊕ is both an infix and a distributed-fix function which adds flows and or leaks. ■

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. ∀ *pls:PLS,b,b′:B,u,u′:U*•
68. {*b,b′*}⊆**obs_***Bs(pls)*∧*b*≠*b′*∧*u′*=**obs_***U(b′)*
68. ∧ **let** *(iuis,ouis)*=**mereo_***U(u),(iuis′,ouis′)*=**mereo_***U(u′)*,
68. *ui*=**uid_***U(u),ui′*=**uid_***U(u′)* **in**
68. *ui* ∈ *iuis* ∧ *ui′* ∈ *ouis′* ⇒
68a.. **attr_***cur_oF(u′)(ui′)* − **attr_***leak_oF(u′)(ui′)*
68b.. = **attr_***cur_iF(u)(ui)* + **attr_***leak_iF(u)(ui)*
68. **end**
68. **comment:** *b′ precedes b*

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"

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

- Page 15 for *Well-formedness of Links, L, and Hubs, H*,
- Page 23 for *Well-formedness of Domain Mereologies*,
- Page 24 for *Well-formedness of Road Nets, N*,
- Page 24 for *Well-formedness of Pipeline Systems, PLS (0)*,
- Page 18 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 9 for *Disjointness of Part Sorts*,
- Page 17 for *Disjointness of Attribute Types* and
- Page 26 for *Disjointness of Material Sorts*.

Example 41 No Pipeline Junk: *We continue Example 16 on Page 13 and Example 33 on Page 24. We define a pipeline route to be a sequence of connected pipeline units in the direction of the flow.*

69. *A well-formed pipeline system is then one that does not allow "junky" routes, i.e., routes*
    a. *that are circular,*
    b. *that requires each well to be connected to at least one sink, and*
    c. *where each sink is reachable from at least one well.*

To formalise the above we describe some auxiliary notions.

Pipe Routes

70. *A route (of a pipeline system) is a sequence of connected units (of the pipeline system).*

71. *A route descriptor is a sequence of unit identifiers and the connected units of a route (of a pipeline system).*

```
    type
70.     R′ = U^ω
70.     R = {| r:Route′•wf_Route(r) |}
71.     RD = UI^ω
    axiom [ Well−formedness of Pipeline Route Descriptors, RD ]
71.     ∀ rd:RD • ∃ r:R•rd=descriptor(r)
    value
71.     descriptor: R → RD
71.     descriptor(r) ≡ ⟨uid_UI(r[ i ])|i:Nat•1≤i≤len r⟩
```

72. Two units are adjacent if the output unit identifiers of one shares a unique unit identifier with the input identifiers of the other.

```
    value
72.   adjacent: U × U → Bool
72.   adjacent(u,u′) ≡
72.     let (,ouis)=mereo_U(u),
72.         (iuis,)=mereo_U(u′) in
72.     ouis ∩ iuis ≠ {} end
```

73. 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 $\langle$**uid_**UI(u)$\rangle$ is a route of pls.
    c. Let u, u' be any units of pls, such that an output unit identifier of u is the same as an input unit identifier of u' 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 $r^\frown$**tl** r' is a route of pls.
    e. No sequence of units is a route unless it follows from a finite (or an infinite) number of applications of the basis and induction clauses of Items 73a.–73d..

```
    value
73.     Routes: PLS → RD-infset
73.     Routes(pls) ≡
73a..       let rs = ⟨⟩
73a..           ∪ {⟨uid_UI(u)⟩|u:U•u ∈ obs_Us(pls)}
73c..           ∪ {⟨uid_UI(u),uid_UI(u′)⟩|u,u′:U•{u,u′}⊆obs_Us(pls) ∧ adjacent(u,u′)}
73d..           ∪ {r^tl r′|r,r′:R•{r,r′}⊆rs∧r[len r]=hd r′}
73e..       in rs end
```

Well-formed Routes

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

```
    value
74.  acyclic_Route: R → Bool
74.  acyclic_Route(r) ≡ ∼∃ i,j:Nat•{i,j}⊆inds r ∧ i≠j ∧ r[i]=r[j]
```

Well-formed Pipeline Systems

75. 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) ]
75.  ∀ pls:PLS •
75a..     non_circular(pls)
75b..    ∧ are_embedded_in_well_to_sink_Routes(pls)
    value
75.  non_circular_PLS: PLS → Bool
75.  non_circular_PLS(pls) ≡
75.      ∀ r:R•r ∈ routes(p)∧acyclic_Route(r)
```

76. 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
76.  well_to_sink_Routes: PLS → R-set
76.  well_to_sink_Routes(pls) ≡
76.    let rs = Routes(pls) in
76.    {r|r:R•r ∈ rs ∧ is_We(r[1]) ∧ is_Si(r[len r])} end
```

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

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

Embedded Routes

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

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

A Theorem

79. 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:
79.  ∀ pls:PLS •
79.  let rs = Routes(pls),
79.      wsrs = well_to_sink_Routes(pls) in
79a..  rs =
79b..      wsrs ∪
79c..      ∪ {{r′|r′:R • r′ ∈ embedded_Routes(r″)} | r″:R • r″ ∈ wsrs}
78.  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

In Sect. 3.5.2 on Page 8 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.

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 Introduction

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.

The reason for giving this albeit cursory overview of perdurants is that we can justify our detailed study of endurants, their part and subparts, their unique identifiers, attributes and mereology. This justification is found 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 notion related to endurants and now justified by their use in describing perdurants.

Perdurants can perhaps best be explained in terms of a notion of time and a notion of state. We shall in this paper not go into notions of time, but refer to [45, 33, 23, 82].

5 States

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, LΣ, Example 22 on Page 17. A road net can be a state – since its hubs can be. Container stowage areas. CSA, Example 11 on Page 12, of container vessels and container terminal ports can be states as containers can be removed from and put on top of container stacks* ■

6 Actions, Events and Behaviours

To us perdurants are further analysed into actions, events, and behaviours. 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.

6.1 Time Considerations

We shall, without loss of generality, assume that actions and events are atomic and that behaviours are composite. Atomic perdurants may "occur" during some time interval, but we omit consideration of and concern for what actually goes on during such an interval. Composite perdurants can be analysed into "constituent" actions, events and "sub-behaviours". We shall also omit consideration of temporal properties of behaviours. Instead we shall refer to two seminal textbooks: Duration Calculus: A Formal Approach to Real-Time Systems [87, Zhou ChaoChen and Michael Rweichhardt Hansen] and Specifying Systems [55, Leslie Lamport]

6.2 Actors

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

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 9 and 27 on Page 20 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 13 and Examples 16 on Page 13 and 33 on Page 24 will be considered actors. The bank general ledger and each bank customer of the Shared Passbooks example, Example 28 on Page 20, will be considered actors* ■

6.3 Parts, Attributes and Behaviours

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'[20]. 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* ■

7 Discrete Actions

Definition 11 Discrete Action: *By a 'discrete action' [85] 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* ■

8 Discrete Events

Definition 12 Event: *By an 'event' we shall understand some unforeseen thing, that is, some unplanned-for "action" which surreptitiously, non-deterministically changes a well-formed state into another, but usually not a well-formed state, and for which no particular 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 [31, 68, 59, 30, 42, 4, 54, 27, 64, 26]. We note, in particular, [30, 4, 54].

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* ■

9 Discrete Behaviours

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* ■

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: Sequences of movements (by container terminal port cranes) of containers from vessel to port (unloading), interleaved by sequences of movements (by cranes) from port to vessel (loading), including the dropping of containers by cranes* ■

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

9.1 Channels and Communication

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

type M
channel ch M

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

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

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

9.2 Relations Between Attribute Sharing and Channels

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.

Thus, for every pair of parts, p_{ik}:P_i and $\mathsf{p}_{j\ell}$: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 $\mathsf{p}_{j\ell}$:P_j, of these sorts, then just a simple channel, say ch_{P_i,P_j}.

channel 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 $\{p_{j1}, p_{j2}, ..., p_{jn}\}$ be all the part of the domain of sort P_j. Then $uids : \{uid_{p_{j1}}, uid_{p_{j2}}, ..., uid_{p_{jn}}\}$ is the set of their unique identifiers. Now a schematic channel array declaration can be suggested:

channel {ch[{π_i,π_j}]|π_i=**uid**_P_i(p_i)∧π_j ∈ uids}:A.

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

Example 48 Bus System Channels *We extend Example 27 on Page 20. We consider the fleet and the vehicless to be behaviours.*

80. *We assume some transportation system, δ. From that system we observe*
81. *the fleet and*
82. *the vehicles.*
83. *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
80. *δ:Δ,*
81. *f:F* = **obs**_*F(δ)*,
82. *vs:V*-**set** = **obs**_*Vs*(**obs**_*VC*((**obs**_*F(δ)*)))
channel
83. {*fch*[{**uid**_*F(f)*,**uid**_*V(v)*}]|*v:V•v* ∈ *vs*}:*BT* ■

Example 49 Bank System Channels *We extend Example 28 on Page 20. We consider the general ledger and the customers to be behaviours.*

84. *We assume some bank system. From the bank system*
85. *we observe the general ledger.*
86. *and the set of customers.*
87. *We consider passbooks to be the only message communicated between the general ledger and the customer behaviours.*

value
84. *bs:BS*
85. *gl*=**obs_***GL*(*bs*):*GL*
86. *cs*=**obs_***Cs*(*bs*):*C*-**set**

channel
87. {*bsch*[{**uid_***GL*(*gl*),**uid_***C*(*c*)}]|*c*:*C*•*c* ∈ *cs*}:*PB* ■

10 Continuous Behaviours

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

Example 50 Flow in Pipelines *We refer to Examples 33, 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[21]) then there will be oil flowing from the pump outlet into adjacent pipeline units.* ■

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 hydro-dynamics using such second order differential equations as first derived by Bernoulli (1700–1782) and Navier–Stokes (1785–1836 and 1819–1903).

11 Perdurant Signatures and Definitions

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

11.1 Function Signatures

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* $\xrightarrow{\sim}$ *(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**, $\times$, *, $\overrightarrow{m}$ and | type constructors.

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

11.2 Action Signatures and Definitions

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

action: VAL → Σ ~→ Σ
action(v)(σ) **as** σ′
  pre: $\mathcal{P}$(v,σ)
  post: $\mathcal{Q}$(v,σ,σ′)

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 ~→ shall indicate that action(v)(σ) may not be defined for the argument, i.e., initial state σ and/or the argument v:VAL, hence the precondition $\mathcal{P}$(v,σ). The postcondition $\mathcal{Q}$(v,σ,σ′) characterises the "after" state, σ′:Σ, with respect to the "before" state, σ:Σ, and possible arguments (v:VAL).

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

88. *Insertion of a hub requires*
89. *that no hub exists in the net with the unique identifier of the inserted hub,*
90. *and then results in an updated net with that hub.*

value
88. *insert_H: H → N ~→ N*
88. *insert_H(h)(n)* **as** *n′*
89. **pre**: ∼∃ *h′:H•h′* ∈ **obs_**Hs(**obs_**HS(n))•**uid_**H(h)=**uid_**H(h′)
90. **post**: **obs_**Hs(**obs_**HS(n′))=**obs_**Hs(**obs_**HS(n))∪{h} ■

Which could be the argument values, v: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 analysis thus proceeds as follows: identifying relevant actions, assigning names to these, delineating the "smallest" relevant state[22], 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".

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 × (HI × HI) → N ~→ N*

Removing a hub and removing a link:

value remove_H: HI → N ~→ N
  remove_L: LI → N ~→ N

Changing a hub state.

value change_HΣ: HI × HΣ → N ~→ N ■

[22] Experience shows that the domain analyser cum describer should strive for identifying the smallest state.

11.3 Event Signatures and Definitions

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

value
  event: $\Sigma \times \Sigma \rightarrow$ **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 Disappearence Formalisation *We formalise aspects of the above-mentioned link disappearance event:*

91. The result net is not well-formed.

92. For a link to disappear there must be at least one link in the net;

93. and such a link may disappear such that

94. it together with the resulting net makes up for the "original" net.

value
91. *link_diss_event: $N \times N' \times$* **Bool**
91. *link_diss_event(n,n′)* **as** *tf*
92. **pre**: **obs_**Ls(**obs_**LS(n))≠{}
93. **post**: $\exists$ *l*:L•$l \in$ **obs_**Ls(**obs_**LS(n)) ⇒
94. $l \notin$ **obs_**Ls(**obs_**LS(n')) $\wedge$ $n' \cup \{l\} =$ **obs_**Ls(**obs_**LS(n)) ■

11.4 Discrete Behaviour Signatures and Definitions

We shall only cover behaviour signatures when expressed in RSL/CSP [39]. The behaviour functions are now called processes. As shown in [15] 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 therefore:

behaviour: π:Π × p:P × VAL → **out** ochs **in** ichns → **process**

where π:Π is the unique identifier of part p, i.e., π=**uid_**P(p), and ochs, ichs are channel expressions, generally of the form

ochs,ichs: {ch[i]|i:IDE•$\mathcal{P}$(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 $p_a, p_b, \ldots, p_c$: 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) ≡
        p: π:Π × p:P × attrs:P_ATTRS →
            in,out {ch[{π,pi}]•pi ∈ mereo_P(p)} process
        p(π:uid_(p),p,attrs:obs_attribs(p)) ≡
            M_P(π,p,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: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) ≡
        p: π:Π × p:P × attrs:P_ATTRS →
            in,out {ch[{π,pi}]•pi ∈ mereo_P(p)} process
        p(π:uid_(p),p,attrs:obs_attribs(p)) ≡
            M_P(π,p,attrs)
          ‖ ‖{comp_process(q)|q:Q•q ∈ qs}
```

Example 54 Bus Timetable Coordination *We refer to Examples 9 on Page 9, 10 on Page 11, 27 on Page 20 and 48 on Page 35.*

95. *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.*

96. *The fleet process is compiled as per Process Schema II (Page 39)*

```
    type
        Δ, F, VC        [Example 9 on Page 9]
        V, Vs=V-set     [Example 10 on Page 11]
        FI, VI, BT      [Example 27 on Page 20]
    channel
        {fch...}        [Example 48 on Page 35]
    value
95.     δ:Δ,
95.     f:F = obs_F(δ),
95.     vs:V-set = obs_Vs(obs_VC(f)),
95.     bt:BT = attr_BT(f)
    axiom
95.     ∀ v:V•v ∈ vs ⇒ bt = attr_BT(v) [Example 27 on Page 20]
```

value
96. *fleet: fi:FI×f:F×BT* → **in**,**out** {*fch*[{*fi*,**uid_***V(v)*}]|*v:V•v* ∈ *vs*} **process**
96. *fleet(fi,f,bt)* ≡ $\mathcal{M}_F$*(fi,f,bt)* ‖ ‖ {*vehicle*(**uid_***V(v),v,bt)*|*v:V•v* ∈ *vs*}
96. *vehicle: vi:VI×v:V×bt:BT* → **in**,**out** *fch*[{*fi,vi*}] **process**
96. *vehicle(vi,v,bt)* ≡ $\mathcal{M}_V$*(vi,v,bt)*

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

value
$\mathcal{M}_F$: FI×F×BT → **in**,**out** {fch[{fi,**uid_**V(v)|v:V•v ∈ vs} **process**
$\mathcal{M}_F$(fi,f,bt) ≡ **let** bt′ = $\mathcal{F}$(fi,f,bt) **in** $\mathcal{M}_F$(fi,f,bt′) **end**

$\mathcal{M}_V$: VI×V×BT → **in**,**out** fch[{fi,vi}] **process**
$\mathcal{M}_V$(vi,v,bt) ≡ **let** bt′ = $\mathcal{V}$(vi,v,bt) **in** $\mathcal{M}_V$(vi,v,bt′) **end**

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

value
$\mathcal{F}$: FI×F×BT → **in**,**out** {fch[{fi,**uid_**V(v)|v:V•v ∈ vs} BT
$\mathcal{F}$(fi,f,bt) ≡ ...

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

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 28 on Page 20.*

97. *bs is the bank system,*
98. *gl is the general ledger of the bank administration,*
99. *pbs is the set of passbooks attribute of the general ledger and*
100. *cs is the set of bank customers.*
101. *bank is the overall bank system behaviour.*
102. *gen_ldgr is the behaviour of the general ledger, that is, the demand/deposit activities of the bank.*
103. *clients is the overall behaviour of the ensemble of bank demand/deposit [account] customers.*
104. *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 38) – two "compilations"!

type
[parts] BS, AD, GL, CS, Cs, C [Example 28 on Page 20]
[attribute] PB [Example 28 on Page 20]
value
97. *bs:BS*
98. *gl:GL* = **obs_***GL*(**obs_***AD(bs)*), *glid:GLI* = **uid_***GL(gl)*
99. *pbs:PB*-**set** = **attr_***PSs(gl)*
100. *cs:C*-**set** = **obs_***Cs*(**obs_***CS(bs)*)
axiom
99. *pbs* = {**attr_***PS(c)*|*c:C:0c* ∈ *cs*} *[Example 28 on Page 20]*

value
99. bank: bs:BS → **process**
99. bank(bs) ≡ gen_ldgr(glid)(gl)(pbs) ‖ clients(cs)
100. gen_ldgr: π:GLI × GL × PB-**set** → **in**,**out** {bch[π,**uid**_C(c)]|c:C•c ∈ cs} **process**
100. gen_ldgr(π,gl,pbs) ≡ $\mathcal{M}_{GL}$(π,gl,pbs)
101. clients: C-**set** → **in**,**out** {bch[π,**uid**_C(c)]|c:C•c ∈ cs} **process**
101. clients(cs) ≡ ‖{customer(**uid**_C(c),c,**attr**_PB(c))|c:C•c ∈ cs}
102. customer: π:CI × C × PB → **in**,**out** {bsch[glid,π]} **process**
102. customer(**uid**_C(c),c,**attr**_PB(c)) ≡ $\mathcal{M}_C$(**uid**_C(c),c,**attr**_PB(c))

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 ■

12 Summary and Discussion of Perdurants

12.1 Summary

TO BE WRITTEN

12.2 Discussion

TO BE WRITTEN

13 Conclusion

The construction, existence, propagation & acceptance of domain descriptions serve a number of rôles: a construction process rôle is to explore, inquire and theorize about a domain; an existence rôle is 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; 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 [11, 14]; 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!

13.1 On Domain Description Languages

We have in this paper expressed the domain descriptions in the `RAISE` [40] specification language `RSL` [39]. With what is thought of as basically inessential, editorial changes, one can reformulate these domain description texts in either of `Alloy` [48] or `The B-Method` [1] or `VDM` [20, 21, 35] or `Z` [86]. 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` [69] or `MSC` [47] (Message Sequence Charts) or `StateCharts` [43]. 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` [87] or `TLA+: Temporal Logic of Actions` [55].

13.2 A Review of Our Claims: Interpretation and Evaluation

We structure this review according to that of Sect. 1.2 on Page 4.

[1] Domain Description Components Sections 3–3.3, as they appear before Sect. ??, 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 [74], 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").)

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`.

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!

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'.

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 domain modelling attributes one usually exerts great restraint. Usually one can associate "zillions" of attributes with even atomic parts. We advocate selecting as few attributes as needed.

The chosen domain description language, `RSL` [39], allow 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, [74], than from model-oriented ones.

[2] Domain Observer Functions The **obs_P**, **obs_M**, **uid_P**, **mereo_P** and **attr_A**, etcetera, literals are emphasized in bold face and underlined. 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.

14 Comparison to Other Work

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'!

[1] Ontological and Knowledge Engineering: Ontological engineering [5] 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. Instead the ontology databases appear to be used for the computerised discovery and analysis of relations between ontologies.

The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [34]: 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. A seminal text is that of [32]. 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 algorithmswhose 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 [22].

The concerns of `TripTych` domain science & engineering is based on that of algorithmic engineering. 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.

[2] Domain Analysis: Domain analysis, or product line analysis (see below), as it was then conceived in the early 1980s by James Neighbors 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. In this section we shall mainly be comparing the `TripTych` approach to domain analysis to that of Reubén Prieto-Dĩaz's approach [65, 66, 67]. Firstly, the two meanings of domain analysis basically coincide. Secondly, in, for example, [65], Prieto-Dĩaz'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-Dĩaz. 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-Dĩaz'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-Dĩaz does mention a need for domain specific languages, he does not show examples of domain descriptions in such `DSL`s. We, of course, basically use mathematics as the `DSL`. In our approach we do not consider requirements, let alone software components, as do Prieto-Dĩaz, but we find that that is not an important issue.

[3] Domain Specific Languages Martin Fowler[23] defines a *Domain-specific language* (`DSL`) *as a computer programming language of limited expressiveness focused on a particular domain* [36]. Other references are [61, 78]. Common to [78, 61, 36] is that they define a domain in terms of

[23] http://martinfowler.com/dsl.h

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.

[4] Feature-oriented Domain Analysis (FODA): 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 [53] 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 [29]. Since FODA "leans" quite heavily on 'Software Product Line Engineering' our remarks in that section, next, apply equally well here.

[5] Software Product Line Engineering: Software product line engineering, earlier known as domain engineering, is the entire process of reusing domain knowledge in the production of new software systems. Key concerns of software product line engineering are reuse, the building of repositories of reusable software components, and domain specific languages with which to more-or-less automatically build software based on reusable software components. These are not the primary concerns of TripTych domain science & engineering. But they do become concerns as we move from domain descriptions to requirements prescriptions. But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is TripTych domain engineering. It seems that software product line engineering is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [14] puts the ideas of software product lines and model-oriented software development in the context of the TripTych approach.

[6] Problem Frames: The concept of problem frames is covered in [51]. 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 [49] 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.

[7] Domain Specific Software Architectures (DSSA): It seems that the concept of DSSA was formulated by a group of ARPA[24] 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 [81]. The [81] 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 [62, 75, 58]. 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 descriptionss without, in principle, considering software components, whether already existing, or

[24] ARPA: The US DoD Advanced Research Projects Agency

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.

[8] Domain Driven Design (DDD) Domain-driven design (DDD)[25] *"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."*[26] We have studied some of the DDD literature, mostly only accessible on the Internet, but see also [44], and find that it really does not contribute to new insight into domains such as wee see them: it is just "plain, good old software engineering cooked up with a new jargon.

[9] Unified Modelling Language (UML) Three books representative of UML are [24, 70, 52]. 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, Items [3,4,5,7,8], with either software design (as it most often is), or requirements prescription. Certainly, in UML, in [24, 70, 52] as well as in most published papers claiming "adherence" to UML, that domain analysis usuallyis 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 perdurants, and 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 [3, Bachman, 1969] and [28, 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.

• • •

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, [41, 49], 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 andattributes. 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. 11.4 Pages 38–39.

[25] Eric Evans: http://www.domaindrivendesign.org/
[26] http://en.wikipedia.org/wiki/Domain-driven_design

14.1 What Have We Not Covered ?

The concept of domain science & engineering, such as basically formulated in this paper, was covered, in some other form, in [9]. The domain analysis development stage, such as covered in this paper, is, however, new.

14.1.1 Domain Facets

A development stage that has not been covered in this paper is that of domain facet description. It was covered in [9] and then further developed in [12]. In [10] we list some research challenges based on that 2007–2008 understanding of domain science & engineering.

[0] Domain Facet 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: intrinsics, support technology, organisation & management, rules & regulations, script and behaviour.

[1] The Intrinsics Facet By ′**intrinsics**′ 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 intrinsics initially covering at least one specific, hence named, stakeholder view.

[2] The Support Technology Facet By the ′**support technology facet**′ of a domain we understand ways and means of implementing certain observed phenomena.

[3] 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.

[4] The Rules & Regulations Facet By a ′**rule**′ of a domain we shall understand some text (in the domain) which prescribes how people are 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.

[5] 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.

[6] 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 technologyto 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 [12] and [10] need be reviewed in the context of a much sharper understanding now of the concept of domain analysis.

14.2 Future Work

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.

14.2.1 Analysis of Perdurants

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, [46], there is graphic notations of Petri Nets, [69], Message Sequence Charts, [47], State-charts, [43], and others.

14.2.2 Commensurate Discrete and Continuous Models

Section 10 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 different equations (PDEs). The problem that arises in this situation is the following: there will be, say variable identifiers, e.g., x, y, ..., 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 languages[27] 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.

14.2.3 Interplay between Parts and Materials

Examples 37 on Page 27, 38 on Page 27, 39 on Page 28, 40 on Page 28 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 28, 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.

14.2.4 Towards a Mathematical Model of Domain Analysis & Description

There are two aspects to a precise descrition of the 'domain analysis prompt's and 'domain description prompt's. There is that of describing the indivial prompts as if they were "machine instructions" for an albeit strange manchine; and there is that of describing the interplay beween 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 [17, Processes]. And we are in the midst of describing and formalising the former in [16, Prompts].

14.2.5 Domains and Galois Connections

Section 2.4 on Page 6 very briefly mentioned that formal concepts form Galois Connections. In the seminal [38] 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 latttices. Further study need be done.

[27] Alloy [48], Event B [1], RSL [39], VDM-SL [20, 21, 35], Z, etc.

14.2.6 Domain Theories

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 descriprions. 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 descriprion 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 journay at that station plus the number of trains starting their journay at that station equals number of trains departing from that station.* ■

14.2.7 Precise Descriptions of Man-made Domains

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 [7] and [8, Part VII (Chapters 16–19)] can now be further developed into a theory and practice for deriving general software from formal domain descriptions [11].

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.

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

14.3 Acknowledgements

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:[28] 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

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

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.

15 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] C. Bachman. Data structure diagrams. *Data Base, Journal of ACM SIGBDP*, 1(2), 1969.

[4] A. Badiou. *Being and Event*. Continuum, 2005. (Lêtre et l'événements, Edition du Seuil, 1988).

[5] 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.

[6] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 3 edition, 1967.

[7] 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.

[8] 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.

[9] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] D. Bjørner. Domain Analysis: A Model of Prompts [Writing of crucial final section yet to begin] (paper[29], slides[30]). Research Report 2013-6, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Summer 2013. A first draft of this document will be written over the summer of 2013.

[17] D. Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model [Almost final draft] (paper[31], slides[32]). Research Report 2013-9, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, May 2013.

[18] D. Bjørner. Pipelines – a Domain Description[33]. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[19] D. Bjørner. Road Transportation – a Domain Description[34]. Experimental Research Report 2013-4, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[20] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[21] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[22] 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.

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

[24] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[25] R. Carnap. *Der Logische Aufbau der Welt*. Weltkreis, Berlin, 1928.

[26] R. Casati and A. Varzi. Events. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2010 edition, 2010.

[27] 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.

[28] P. P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst*, 1(1):9–36, 1976.

[29] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[30] D. Davidson. *Essays on Actions and Events*. Oxford University Press, 1980.

[31] F. Dretske. Can Events Move? *Mind*, 76(479-492), 1967. reprinted in [27], pp. 415-428.

[32] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.

[33] 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.

[29] http://www.imm.dtu.dk/~dibj/da-mod-p.pdf
[30] http://www.imm.dtu.dk/~dibj/da-mod-s.pdf
[31] http://www.imm.dtu.dk/~dibj/jaist-da.pdf
[32] http://www.imm.dtu.dk/~dibj/jaist-s.pdf
[33] http://www.imm.dtu.dk/~dibj/pipe-p.pdf
[34] http://www.imm.dtu.dk/~dibj/road-p.pdf

[34] E. A. Feigenbaum and P. McCorduck. *The fifth generation*. Addison-Wesley, Reading, MA, USA, 1st ed. edition, 1983.

[35] 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.

[36] M. Fowler. *Domain Specific Languages*. Signature Series. Addison Wesley, October 20120.

[37] 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.

[38] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag, January 1999. ISBN: 3540627715, 300 pages, Amazon price: US $ 44.95.

[39] 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 Hampstead, England, 1992.

[40] 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 Hampstead, England, 1995.

[41] 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.

[42] P. Hacker. Events and Objects in Space and Time. *Mind*, 91:1–19, 1982. reprinted in [27], pp. 429-447.

[43] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[44] D. Haywood. *Domain-Driven Design Using Naked Objects*. The Pragmatic Bookshelf (an imprint of 'The Pragmatic Programmers, LLC.'), http://pragprog.com/, 2009.

[45] M. Heidegger. *Sein und Zeit (Being and Time)*. Oxford University Press, 1927, 1962.

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

[47] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[48] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[49] 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.

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

[51] M. A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison-Wesley, England, 2001.

[52] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[53] K. C. Kang, S. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA). 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.

[54] J. Kim. *Supervenience and Mind*. Cambridge University Press, 1993.

[55] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[56] H. Laycock. Object. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2011 edition, 2011.

[57] E. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.

[58] 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.

[59] D. Mellor. Things and Causes in Spacetime. *British Journal for the Philosophy of Science*, 31:282–288, 1980.

[60] D. H. Mellor and A. Oliver, editors. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, May 1997. ISBN: 0198751761, 320 pages.

[61] 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.

[62] 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.

[63] 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.

[64] C.-Y. T. Pi. *Mereology in Event Semantics*. Phd, McGill University, Montreal, Canada, August 1999.

[65] R. Prieto-Díaz. Domain Analysis for Reusability. In *COMPSAC 87*. ACM Press, 1987.

[66] R. Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.

[67] R. Prieto-Díaz and G. Arrango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.

[68] A. Quinton. Objects and Events. *Mind*, 88:197–214, 1979.

[69] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[70] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[71] B. Russel. *"Preface," Our Knowledge of the External World*. G. Allen & Unwin, Ltd., London, 1952.

[72] B. Russell. On Denoting. *Mind*, 14:479–493, 1905.

[73] 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.

[74] D. Sannela and A. Tarlecki. *Foundations of Algebraic Semantcs and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.

[75] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[76] B. Smith. Ontology and the Logistic Analysis of Reality. In G. Haefliger and P. M. Simons, editors, *Analytic Phenomenology*. Dordrecht/Boston/London: Kluwer, Padua, Italy, 1993.

[77] 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.

[78] D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

[79] J. Srzednicki and Z. Stachniak, editors. *Leśniewksi's Lecture Notes in Logic*. Dordrecht, 1988.

[80] S. J. Surma, J. T. Srzednicki, D. I. Barnett, and V. F. Rickey, editors. *Stanisław Leśniewksi: Collected works (2 Vols.)*. Dordrecht, Boston – New York, 1988.

[81] W. Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.

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

[83] A. Whitehead. *The Concept of Nature*. Cambridge University Press, Cambridge, 1920.

[84] A. Whitehead. *An Enquiry Concerning the Principles of Natural Knowledge*. Cambridge University Press, Cambridge, 2929.

[85] G. Wilson and S. Shpall. Action. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[86] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[87] 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.

16 Indexes

I have yet to check the indexing.

17 Index of Endurant Analysis Prompts

18 Index of Attribute Analysis Prompts

19 Index of Domain Description Prompts

19.1 Index of Domain Description Schemas

19.2 Index of Description Language Observers and Functions

19.3 Index of Axioms

19.4 Index of Proof Obligations

19.5 Index of Definitions

19.6 Index of Examples

19.7 Index of Concepts