Domain Science & Engineering*

A Precursor for Requirements Engineering

1

Tutorial Notes

Dines Bjørner
DTU Informatics, Techn.Univ.of Denmark
bjorner@gmail.com, www.imm.dtu.dk/~dibj

August 10, 2012: 09:44

2
3
4

Abstract

This tutorial covers a **new science & engineering of domains** as well as a **new foundation for software development**. We treat the latter first. Instead of commencing with requirements engineering, whose pursuit may involve repeated, but unstructured forms of domain analysis, we propose a predecessor phase of domain engineering.

That is, we single out domain analysis as an activity to be pursued prior to requirements engineering. In emphasising domain engineering as a predecessor phase we, at the same time, 5
introduce a number of facets that are **not present,** we think, in current software engineering studies and practices.

(i) **One facet** is **the construction of separate domain descriptions.** Domain descriptions are void of any reference to requirements and encompass the modelling of domain phenomena without regard to their being computable. 6

(ii) **Another facet** is **the pursuit of domain descriptions as a free-standing activity.** In this tutorial we emphasize domain description development need not necessarily lead to software development. This gives a new meaning to business process engineering, and should lead to a deeper understanding of a domain and to possible non-IT related business process re-engineering of areas of that domain. In this tutorial we shall investigate a method for analysing domains, for constructing domain descriptions and some emerging scientific bases. 7

Our contribution to **domain analysis** is that we view domains as having the following **ontology**. There are the entities (Pages 14–58) that we can describe and then there is "the rest" which we leave un-described.

We analyse entities into endurant entities (Pages 17–34) and perdurant entities (Pages 34–58), that is, parts and materials as endurant entities (Pages 17–32) and discrete actions, discrete events and behaviours as perdurant entities (Pages 34–58), respectively.

Another way of looking at entities is as discrete entities (Pages 17–28 and 34–53), or as continuous entities (Pages 28–32 and Pages 53–58). 8

We also contribute to the analysis of discrete endurants in terms of the following notions: part types and material types (Pages 18–21 and Pages 28–32), part unique identifiers (Pages 22–22), part mereology (Pages 22–25) and part attributes and material attributes (Pages 26–27, Pages 30–31) and **material laws** (Pages 31–32).

Of the above we point to the introduction, into computing science and software engineering of the notions of materials (Pages 28–32) and continuous behaviours (Pages 53–58) **as novel.**

*This is **version 8** of a paper for a tutorial to be presented Tuesday 28 August 2012 at the FM 2012 International Symposium at CNAM (Conservatoire National des Arts et Métiers), Paris, France. The version of this document that participants in this tutorial (and, in general, FM 2012 tutorial participants) will receive was a July 15, 2012 **version 3**. Vertical margin bars, if present, show changes in this document wrt. a July 15 **version 3** of this document. Versions or slight revisions (same version number) subsequent to **version 7** can be retrieved from http://www.imm.dtu.dk/~dibj/dsae-p.pdf. The document http://www.imm.dtu.dk/~dibj/4-dsae-s.pdf represents the four-slides-per-page slides corresponding to dsae-p.pdf. Margin numerals in the dsae-p.pdf document refer to slide numbers in the dsae-s.pdf and 4-dsae-s.pdf documents.

Contents

Part I
Opening

1 Introduction 9

We beg the reader to re-read the **abstract**, Page 1, as for the **contributions** of this tutorial.

This is primarily a methodology paper. By a method we shall understand a set of **principles** for **selecting** and **applying** a number of **techniques** and **tools** in order to **analyse** a **problem** and **construct** an **artefact**. By methodology we shall understand the study and knowledge about methods. 10

This tutorial contributes to the study and knowledge of software engineering development methods. Its contributions are those of suggesting and exploring domain engineering and domain engineering as a basis for requirements engineering. We are not saying *"thou must develop software this way"*, but we do suggest that since it is possible and makes sense to do so it may also be wise to do so.

1.1 Domains: Some Definitions 11

By a domain we shall here understand an area of human activity characterised by observable phenomena: entities whether endurants (manifest parts and materials) or perdurants (actions, events or behaviours), whether discrete or continuous; and of their properties. 12

Example: 1 Some Domains. Some examples are:

air traffic,
airport,
banking,
consumer market,
container lines,
fish industry,
health care,
logistics,
manufacturing,
pipelines,
securities trading,
transportation
etcetera. ■

1.1.1 Domain Analysis

13

By domain analysis we shall understand an inquiry into the domain, its entities and their properties.

14

Example: 2 A Container Line Analysis. *parts:* container, vessel, terminal port, etc.; *actions:* container loading, container unloading, vessel arrival in port, etc.; *events:* container falling overboard; container afire; etc.; *behaviour:* vessel voyage, across the seas, visiting ports, etc. Length of a container is a container *property*. Name of a vessel is a vessel *property*. Location of a container terminal port is a port *property*.

1.1.2 Domain Descriptions

15

By a domain description we shall understand a narrative description tightly coupled (say line-number-by-line-number) to a formal description. To develop a domain description requires a thorough amount of domain analysis.

16

Example: 3 A Transport Domain Description.

- *Narrative:*

  ⌦ a transport net, n:N,
  consists of an aggregation of hubs, hs:HS,
  which we "concretise" as a set of hubs, H-**set**, and
  an aggregation of links, ls:LS, that is, a set L-**set**,

- *Formalisation:*

```
type N, HS, LS, Hs = H-set, Ls = L-set, H, L
value
    obs_HS: N→HS,
    obs_LS: N→LS.
    obs_Hs: HS→H-set,
    obs_Ls: LS→L-set.
```

This example will be a leading one throughout this tutorial. ■

An interesting domain description is usually a document of a hundred pages or so. Each page "listing" pairs of enumerated informal, i.e., narrative descriptions with formal descriptions.

1.1.3 Domain Engineering

17

By domain engineering we shall understand the engineering of a domain description, that is, the rigorous construction of domain descriptions, and the further analysis of these, creating theories of domains[1], etc. The size[2], structure[3] and complexity[4] of interesting domain descriptions is usually such as to put a special emphasis on engineering: the management and organisation of several, typically 5–6 collaborating domain describers, the ongoing check of description quality, completeness and consistency, etcetera.

18

1.1.4 Domain Science

19

By domain science we shall understand two things: the general study and knowledge of how to create and handle domain descriptions (a general theory of domain descriptions) and the specific study and knowledge of a particular domain. The two studies intertwine.

[1] Examples of such theories, albeit in rather rough forms, are given in Appendices B–C.
[2] usually, say a hundred pages
[3] usually a finely sectioned document of may subsub···subsections
[4] having many cross-references between subsub···subsections

1.2 The Triptych of Software Development

We suggest a "dogma": before software can be designed one must understand[5] the requirements; and before requirements can be expressed one must understand[6] the domain.

We can therefore view software development as ideally proceeding in three (i.e., TripTych) phases: an initial phase of domain engineering, followed by a phase of requirements engineering, ended by a phase of software design.

In the domain engineering phase[7] ($\mathcal{D}$) a domain is analysed, described and "theorised", that is, the beginnings of a specific domain theory is established. In the requirements engineering phase[8] ($\mathcal{R}$) a requirements prescription is constructed — significant fragments of which are "derived", systematically, from the domain description. In the software design phase[9] ($\mathcal{S}$) a software design is derived, systematically, rigorously or formally, from the requirements prescription. Finally the $\mathcal{S}$oftware is proven correct with respect to the $\mathcal{R}$equirements under assumption of the $\mathcal{D}$omain: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$.

By a machine we shall understand the hardware and software[10] of a target, i.e., a required IT system.

In [11, 22, 13] we indicate how one can "derive" significant parts of requirements from a suitably comprehensive domain description – basically as follows. Domain projection: from a domain description one projects those areas that are to be somehow manifested in the software. Domain initialisation: for that resulting projected requirements prescription one initialises a number of part types as well as action and behaviour definitions, from less abstract to more concrete, specific types, respectively definitions. Domain determination: hand-in-hand with domain initialisation a[n interleaved] stage of making values of types less non-deterministic, i.e., more deterministic, can take place. Domain extension: Requirements often arise in the context of new business processes or technologies either placing old or replacing human processes in the domain. Domain extension is now the 'enrichment' of the domain requirements, so far developed, with the description of these new business processes or technologies. Etcetera. The result of this part of "requirements derivation" is the domain requirements.

A set of domain-to-requirements operators similarly exists for constructing interface requirements from the domain description and, independently, also from knowledge of the machine for which the required IT system is to be developed. We illustrate the techniques of domain requirements and interface requirements in Sect. 12.

Finally machine requirements are "derived" from just the knowledge of the machine, that is, the target hardware and the software system tools for that hardware. Since the domain does not "appear" in the construction of the machine requirements we shall not illustrate that aspect of requirements prescription in Sect. 12. When you review this section ('A Triptych of Software Development') then you will observe how 'the domain' predicates both the requirements and the software design. For a specific domain one may develop many (thus related) requirements and from each such (set of) requirements one may develop many software designs. We may characterise this multitude of domain-predicated requirements and designs as a product line [16]. You may also characterise domain-specific developments as representing another 'definition' of domain engineering.

[5] Or maybe just: have a reasonably firm grasp of
[6] See previous footnote!
[7] See Sect. 2
[8] See Sect. 12
[9] We do not illustrate the software design phase in this paper.
[10] By software we shall understand all the development documentation, from domain descriptions via requirements prescriptions to software design; all verification data: the formal tests, model checkings and proofs; the development contracts, the management plans, the budgets and accounts; the staffing plans; the installation manuals, the user manuals, the (perfective, adaptive, corrective, etc.) maintenance manuals, and the development methodology manuals; as well as all the software development tools used in the actual development.

1.3 Issues of Domain Science & Engineering 26

We specifically focus on the following issues of domain science &[11] engineering: (i) which are the "things" to be described[12], (ii) how to analyse these "things" into constituent description structures[13], (iii) how to describe these "things" informally and formally, (iv) how to further structure descriptions[14], and a further study of (v) mereology[15].

1.4 Structure of Paper 27

It is always a good idea to consult and study the *table of contents* listing of the paper one is studying. Therefore one is brought here:

---

[11] When we put '&' between two terms that the compound term forms a whole concept.
[12] endurants [manifest entities henceforth called parts and materials] and perdurants [actions, events, behaviours]
[13] atomic and composite, unique identifiers, mereology, attributes
[14] *intrinsics, support technology, rules & regulations, organisation & management, human behaviour* etc.
[15] the study and knowledge of parts and relations of parts to other parts and a "whole".

First, Sect. 1, we introduce the problem. And that was done above. Then, in Sects. 2–10 we bring a rather careful analysis of the concept of the observable, manifest phenomena that we shall refer to as entities. We strongly think that these sections of this tutorial brings, to our taste, a simple and elegant reformulation of what is usually called *"data modelling"*, in this case for domains — but with major aspects applicable as well to requirements development and software design. That analysis focuses on endurant entities, also called parts and materials, those that can be observed at no matter what time, i.e., entities of substance or continuant, and perdurant entities: action, event and behaviour entities, those that occur, that happen, that, in a sense, are accidents. **We think** that this "decomposition" of the "data analysis" problem into discrete parts and continuous materials, atomic and composite parts, their unique identifiers and mereology, and their attributes **is novel**, and differs from past practices in domain analysis.

In Sect. 11 we suggest for each of the entity categories parts, materials, actions, events and behaviours, a calculus of meta-functions: analytic functions, that guide the domain description developer in the process of selection, and so-called discovery functions, that guide that person in "generating" appropriate domain description texts, informal and formal. The domain description calculus is to be thought of as directives to the domain engineer, mental aids that help a team of domain engineers to steer it simply through the otherwise daunting task of constructing a usually large domain description. Think of the calculus as directing a human calculation of domain descriptions. Finally the domain description calculus section suggests a number of laws that the domain description process ought satisfy. Finally, in Sect. 12, we bring a brief survey of the kind of requirements engineering that one can now pursue based on a reasonably comprehensive domain description. We show how one can systematically, but not automatically "derive" significant fragments of requirements prescriptions from domain descriptions.

• • •

The formal descriptions will here be expressed in the RAISE [45] Specification Language, RSL. We otherwise refer to [7]. Appendix D brings a short primer, mostly on the syntactic aspects of RSL. But other model-oriented formal specification languages can be used with equal success; for example: Alloy [58], Event B [1], VDM [24, 25, 41] and Z [99].

Part II
Entities

2 Domain Entities

The world is divisible into two kinds of people: those who divide the population into two kinds of people and the others. Aristoteles (384 BC 322 BC) viewed the physical world as consisting

of four earthly matters and one "divine" matter: *earth* , *water*, *air*, *fire* and *aether* — the four first corresponding to modern day *solids*, *liquids*, *gas* and *heat*. In this tutorial we shall divide the phenomena we can observe and whose properties we can ascertain into two kinds: the endurant entities (Sects. 4–5) and the perdurant entities (Sects. 8–9). Another "division" is of the phenomena and their properties into the discrete entities (Sect. 4 and Sect. 8) and the continuous entities (Sect. 5 and Sect. 9). You can have it, i.e., the the analysis and the presentation, either way. 38

By a **domain** we shall understand a suitably delineated set of observable entities and abstractions of these, that is, of discrete parts (Sect. 4) and continuous materials (Sect. 5) and, discrete actions (Sect. 8.2) (operation applications causing state changes), discrete events (Sect. 8.3) ("spurious" state changes not [intentionally] caused by actions) discrete discrete behaviours (Sect. 8.4) (seen as sets of sequences of actions, events and behaviours) and continuous behaviours (Sect. 9) (abstracted as continuous functions in space and/or time).

2.1 From Observations to Abstractions 39

When we observe a domain we observe instances of entities; but when we describe those instances (which we shall call values) we describe, not the values, but their type and properties. Parts and materials have types and values; actions, events and behaviours, all, have types and values, namely as expressed by their signatures; and actions, events and behaviours have properties, namely as expressed by their function definitions.

2.2 Algebras 40

Algebra: Taking a clue from mathematics, an algebra is considered a set of endurants: a set of parts and a set of materials (endurant entities) and a set of perdurants: operations on entities (actions, events, behaviours perdurant entities). These operations yield parts or materials. With that in mind we shall try view a domain as an algebra[16], of some kind, of parts and actions, events and behaviours. We shall soon discuss the use of the terms entities, parts, actions, events and behaviours.

2.3 Domain Phenomena 41

By a domain phenomenon we shall understand something that can be observed by the human senses or by equipment based on laws of physics and chemistry. Those phenomena that can be observed by the human eye or touched, for example, by human hands, we call parts and materials. Those phenomena that can be observed of parts and materials can usually be measured and we call them properties of these parts and those materials.

2.4 Ontological Engineering

Ontological concepts permeate studies of domains.

Ontology: There are two, related meanings to this term: an ontology is the *formal, explicit specification of a shared conceptualisation* [46], and ontology is the philosophical study of the nature of being, existence, or reality, as well as the basic categories of being and their relations. Thus ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences [32, 47]. In our study of domains we shall therefore often "border" on issues of philosophy.

Domain Ontology: A domain ontology (or a domain specific ontology) models a specific domain. with a domain description describing this model.

Ontological Engineering: By ontological engineering we understand (i) the study and knowledge of methods and methodologies for building ontologies, and (ii) the practice of these methods.

[16] What we here refer to as an algebra are in some software engineering, notably OO: object-oriented contexts referred to as an object.

Ontological engineering thus has some relations to the TripTych approach's domain engineering, but will first elaborate on that in Sect. 13.1.1.1 (Page 82).

2.5 Entities

42

By a domain entity we shall understand a manifest domain phenomenon or a concept, i.e., an abstraction, derived from a domain entity.

The distinction between a manifest domain phenomenon and a concept thereof, i.e., a domain concept, is important. Really, what we describe are the domain concepts derived from domain
43 phenomena or from other domain concepts.

Ontologically we distinguish between two kinds of domain entities: endurant entities and perdurant entities. We shall characterise these two terms: endurants in Sect. 3.1, Page 17 and perdurants in Sect. 8.1, Page 34[17]. This distinction is supported by current literature on ontology [86]. In this section we shall briefly enter a discourse on *"things"*, *entities*, *objects*, etcetera.

2.5.1 Things, Entities, Objects

In colloquial language, that is, in the vernacular, everyday speech, we may refer to that which we are talking about, describing, identifying, as the 'thing', 'entity', 'object', 'individual', 'unit', 'term', 'particular', 'quantity', [83, Page 43]. etcetera. The problem is: we need words and we use words, and, to avoid confusion, we should use one word, one term, per concept. So where some may use the term *'thing'* or the term *'term'*, or the term *'object'*, etcetera, interchangeably, we shall use the term *'entity'* consistently.

2.5.2 Spatial and Temporal, Manifest and Abstract

Some philosophers "divide" entities (disjointly) into spatial entities, corresponding to endurants, and temporal entities corresponding to perdurants. Some authors use the term object either as manifest object (concrete object, tangible object) or as fiat object[18] or abstract object.

Example: 4 Spatial Entities.

- manifest entity: a geographical well delineated area;
- abstract entity: the border of that area. ■

2.5.3 A Description Bias

44

One of several "twists" that make the TripTych form of domain engineering distinct from that of ontological engineering[19] is that we use a model-oriented formal specification approach[20] where usual ontology formalisation languages are variants of Lisp's [67] S-expressions. KIF: Knowledge Interchange Format, http://www.ksl.stanford.edu/knowledge-sharing/kif/ is a leading ex-
45 ample. The bias is now this: The model-oriented languages mentioned in this section all share the following: (a) a type concept and facilities for defining types, that is: endurants (parts), and (b) a function concept and facilities for defining functions (notably including predicates), that is: perdurants (actions and events). (c) RSL further has constructs for defining processes, which we shall use to model behaviours.

2.5.4 An 'Upper Ontology'

46

By an upper ontology we shall understand a relatively small, ground set of ontology expressions
47 which form a basis for a usually very much larger set of ontology expressions. The need for

[17] The reader may wish to briefly review those two definitions now.
[18] From Latin: an authoritative determination
[19] Other, more substantial differences are touched upon in Sect. 13.1.1.1, Page 82.
[20] RAISE [45]. Our remarks in this section, Sect. 2.5.3, apply equally well had we instead chosen either of the Alloy [58], Event B [1], VDM [24, 25, 41] or Z [99] formal specification languages.

introducing the notion of an upper ontology arose, in the late 1980s to early 1990s as follows: usually an ontology was (is) expressed in some very basic language, viz., Lisp-like S-expressions[21]. This was necessitated by the desire to be able to share ontologies between many computing applications worldwide. Then it was found that several ontologies shared initial bases in terms of which the rest of their ontologies were formulated. These shared bases were then referred to as upper ontologies — and a need to "standardise" these arose [48, 88]. 48

We therefore consider the following model-oriented specification language constructs as forming an upper ontology: types, ground types, type expressions and type definitions; functions, function signatures and function definitions; processes, process signatures and process definitions, as constituting an upper level ontology for TripTych domain descriptions. That is, every domain description is structured with respect to: parts and materials using types, actions using functions, events using predicates, discretebehaviours using processes and continuous behaviours using partial differentialequations. Appendix A on page 97 shows a draft upper ontology for the domain of domain science & engineering.

3 Endurants 49

There is sort of a dichotomy buried in our treating endurants before perdurants. The dichotomy is this: one could claim that the perdurants, i.e., the actions, events and behaviours is *"what it, the domain, is all about"*; To describe these, however, we need refer to endurants!

3.1 General 50

Wikipedia: *By an endurant (also known as a continuant or a substance) we shall understand an entity that can be observed, i.e., perceived or conceived, as a complete concept, at no matter which given snapshot of time. Were we to freeze time we would still be able to observe the entire endurant.*

3.2 Discrete and Continuous Endurants

We distinguish between discrete endurants, which we shall call parts, and continuous endurants, which we shall call materials. 51

We motivate and characterise this distinction.

By a discrete endurant, that is, a part, we shall understand something which is separate or distinct in form or concept, consisting of distinct or separate parts.

By a continuous endurant, that is, a material, we shall understand something which is prolonged without interruption, in an unbroken series or pattern. We shall first treat the idea of discrete endurant, that is, a part (Sect. 4, Pages 17–28), then the idea of continuous endurant, that is, a material (Sect. 20, Pages 28–32).

4 Discrete Endurants: Parts 52

The reader may well ask: why, of many possible terms, have we chosen the term 'part' for the purpose it serves in TripTych's domain science & engineering; why not use another term, for example data, individual, object, value, particular, thing, quantity, unit, etcetera? The answer is simple, pragmatic: first, we need two terms: part and material to cover the concept of endurant; second, the term part has been used by a number of philosophers for exactly that which we wish to designate; third, the other alternatives already have strong connotations in the computing community, connotations that we wish to *"put a distance to"*; and fourth, as long as we have two good terms, then that is fine.

[21] Ontology languages: KIF http://www.ksl.stanford.edu/knowledge-sharing/kif/#manual, OWL [Ontology Web Language] [72], ISO Common Logic [37]

4.1 What is a Part?

By a part we mean an observable manifest endurant.

4.1.1 Classes of "Same Kind" Parts

We repeat: the domain describer does not describe instances of parts, but seeks to describe classes of parts of the same kind. Instead of the term 'same kind' we shall use either the terms part sort or part type.

By a same kind class of parts, that is a part sort or part type we shall mean a class all of whose members, i.e., parts, enjoy "exactly" the same properties where a property is expressed as a
53 proposition.

Example: 5 Part Properties. Examples of part properties are: *has unique identity* (was exemplified, will be properly defined), *has mereology* (was exemplified, will be properly defined), *has length*, *has location*, *has traffic movement restriction* (as for vehicles along a link, one direction, both directions or closed), *has position* (example: vehicle position), *has velocity* and *has acceleration* (the last two holds for vehicles). ■

4.1.2 Concept Analysis as a Basis for Part Typing 54

The domain analyser examines collections of parts. (i) In doing so the domain analyser discovers and thus identifies and lists a number of properties. (ii) Each of the parts examined usually satisfies only a subset of these properties. (iii) The domain analyser now groups parts into collections such that each collection have its parts satisfy the same set of properties, such that no two distinct collections are indexed, as it were, by the same set of properties, and such that all parts are put in some collection. (iv) The domain analyser now assigns distinct type names (same as sort names) to distinct collections. That is how we assign types to parts. The quality of the part type universe depends on how throroughly the domain analysers do their job: (α) collecting sufficiently many examples of parts, (β) enumerating sufficiently many examples of property propositions, and (γ) "assigning" appropriate properties to parts. This step of domain description development is crucial to the appropriateness and acceptability of the resulting domain description. Examining too few parts, enumerating too few and/or irrelevant property propositions sloppyness ingeneral can often result in domain models that turn out to be "unwieldy", models that do not capture, sufficiently elegantly the core domain concepts. For good advice in seeking elegance in models see [60, M.A. Jaclson: Lexicon ...]. We shall return later to a proper treatment of formal concept analysis [43].

4.2 Atomic and Composite Parts 55

Parts may be analysed into disjoint sets of atomic parts and composite parts.

Atomic parts are those which, in a given context, are deemed *not* to consist of meaningful, separately observable proper sub-parts.

Composite parts are those which, in a given context, are deemed to *indeed* consist of mean-
56 ingful, separately observable proper sub-parts. A sub-part is a part.

Example: 6 Atomic and/or Composite Parts. To one person a part may be atomic; to another person the same part may be composite. It is the domain describer who decides the outcome of this aspect of domain analysis. In some domain analysis a 'person' may be considered an atomic part. For the domain of ferrying cars with passengers persons are considered parts. In some other domain analysis a 'person' may be considered a composite part. For the domain of medical surgery persons may be considered composite parts. ■

As to what is an endurant, i.e., a part, as opposed to what is a perdurant, and again, as to what is neither, are "tricky" issues. Philosophers are still grappling with those issues. A "cop-out", for us, would be to say that endurants (parts), when a domain description may eventually
57 be developed into software, end up as data, whereas perdurants end up as code.

Example: 7 Container Lines. We shall presently consider containers (as used in container line shipping) to be atomic parts. And we shall consider a container vessel to be a composite part consisting of an indexed set of container bays where each container bay consists of indexed set of container rows where each container row consists of indexed set of container stacks where each container stack consists of a linearly indexed sequence of containers. Each stack index, when considered across rows and bays, represents a tier. Thus container vessels, container bays, container rows and container stacks are composite parts. ■

4.2.1 An Essay on Abstraction

Example 3 (Page 10) abstracted the description of a claimed transport domain by focusing on terms nets, hubs and links. What kind of transport domain is that? Is it for *road, rail, air,* or *ship transport*? Well, the domain describer (i.e., I) apparently decided that the net, hub, link model should be for all these forms of transport. Had the focus been more "narrow", on for example only one of the *road, rail, air,* or *ship transports*, then the terms hub and link might instead have been *street segment* and *intersection*; *rail line* and *switch, simple crossover, switchable crossover,* etc.; *air lane and airport*; *sea lane* and *harbour*. respectively. So much for the use of 'abstract' terms. But that use covered over a view of a domain, for example, a transport domain, to cover what may appear, at first sight, to be four entirely distinct domains — possibly in the hope that some general understanding of transport domains could be achieved and possibly in the hope that some common sets of requirements and hence common sets of software designs[22] could be achieved. And then that, "lifted" transport domain was not modelled as a graph, as one might conventionally expect it, but, as we shall see more of, as a set of hubs and a set of links *related* by a mereology! I shall not here even try to attempt to give general guide lines for abstraction, for where "to start looking", etcetera, Instead we shall advice the reader to study existing domain descriptions and otherwise, individually, or as a team of domain describers experiment with abstraction levels and "starting points". As for abstraction level you may think the one chosen for transport nets to be too abstract — too many properties left out! Well, as we shall soon see, when introducing properties of parts, these can, in most cases, cover the levels of details that you may be looking for.

4.2.2 Atomic Parts

58

When we observe what we have decided, i.e., analysed, to be an endurant, more specifically an atomic part, of a domain, we are observing an instance of an atomic part. When we describe those instances we describe, not their values, i.e., the instances, but their type and properties. In this 59
section on endurant entities we shall unfold what these properties might be. But, for now, we focus on the type of the observed atomic part. So the situation is that we are observing a number of atomic parts and we have furthermore decided that they are all of *"the same kind"*. What does 60
it mean for a number of atomic parts to be of *"the same kind"* ? It means that we have decided, for any pair of parts considered of the same kind, that the kinds of properties, for such two parts, are *"the same"*, that is, of the same type, but possibly of different values, and that a number of different, other "facets", are not taken into consideration. That is, we abstract a collection of 61
atomic parts to be of the same kind, thereby "dividing the domain of endurants" into possibly two distinct sets those that are of the analysed kind, and those that are not. It is now our description 62
choice to associate with a set of atomic parts of *"the same kind"* a part type (by suggesting a name for that type, for example, T) and a set of properties (of its values): unique identifier, mereology and attributes. Unique identifiers, mereology and attributes will be covered shortly. In fact, this association of part types and triplet of properties: unique identifier, mereology and attributes, is also to be performed when modelling composite parts. We introduce the concepts of unique identifiers, mereology and attributes in Sects. 4.3.2–4.3.3. Later we shall introduce discrete perdurants (actions, 63
events and behaviours) whose signatures involves (possibly amongst others) type T. Now we can characterise *"of the same kind"* atomic part facets[23] being of the same, named part type, having

[22] Cf. Software Product Lines in Sect, 13.1.4 on Page 84.
[23] as well as "of the same kind" composite part facets.

the same unique identifier type, having the same mereology (but not necessarily the same mereology values), and having the same set of attributes (but not necessarily of the same attribute values), 64 The *"same kind"* criteria apply equally well to composite part facets.

Example: 8 Transport Nets: Atomic Parts (I). The types of atomic transportation net parts are: hubs, say of type H, and links, say of type L. The chosen mereology associates with every hub and link a distinct unique identifiers (of types HI and LI respectively), and, *vice versa*, how hubs 65 and links are connected: hubs to any number of links and links to exactly two distinct hubs. The chosen attributes of hubs include hub location, hub design[24], hub traffic state[25], hub traffic state space[26], etc.; and of links include link location, link length, link traffic state[27], link traffic state space[28], etc. With these mereologies and attributes we see that we can consider hubs and links 66 as different kinds of atomic parts. ■

Observers for Atomic Parts

- Let the domain describer decide
  - ⋙ that a type, A (or Δ), is atomic,
  - ⋙ hence that it does not consists of sub-parts.
- Hence there are no observer to be associated with A (or Δ).

4.2.3 Composite Parts 67

The domain describer has chosen to consider a part (i.e., a part type) to be a composite part (i.e., a composite part type). Now the domain describer has to analyse the types of the sub-parts of the composite part. There may be just one *"kind of"* sub-part of a composite part[29], or there may be more than one *"kind of"*[30]. For each such sub-part type the domain describer decides on an 68 appropriate, distinct type name and a sub-part observer (i.e., a function signature).

Example: 9 Container Vessels: Composite Parts. We bring pairs of informal, narrative description texts and formalisations. For a container vessel, say of type V, we have *Narrative:* A container vessel, v:V, consists of container bays, bs:BS. A container bay, b:B, consists of container rows, rs:RS. A container row, r:R, consists of container stacks, ss:SS. A container stack, s:S, consists of a linearly indexed sequence of containers. *Formalisation:*

type V,BS, **value** obs_BS: V→BS,
type B,RS, **value** obs_RS: B→RS,
type R,SS, **value** obs_CS: R→SS,
type SS,S, **value** obs_S: SS→S,
type S = C*. ■

4.2.4 Abstract Types, Sorts, and Concrete Types 69

By an abstract type, or a sort, we shall understand a type which has been given a name but is otherwise undefined, that is, is a space of undefined mathematical quantities, where these are 70 given properties which we may express in terms of axioms over sort (including property) values.

[24] Design: simple crossing, freeway "cloverleaf" interchange, etc.
[25] A hub traffic state is (for example) a set of pairs of link identifiers where each such pair designates that traffic can move from the first designated link to the second.
[26] A hub state space is (for example) the set of all hub traffic states that a hub may range over.
[27] A link traffic state is (for example) a set of zero to two distinct pairs of the hub identifiers of the link mereology.
[28] A link traffic state space is (for example) the set of all link traffic states that a link may range over.
[29] that is, only one sub-part type
[30] that is, more than one sub-part type

By a concrete type we shall understand a type, T, which has been given both a name and a defining type expression of, for example the form T = A-**set**, T = A-**infset**, T = A×B×···×C, T = A*, T = A$^\omega$, T = A $\overrightarrow{m}$ B, T = A→B, T = A$\xrightarrow{\sim}$B, or T = A|B|···|C. where A, B, ..., C are type names or type expressions. 71

Example: 10 **Container Bays.** We continue Example 9 on the preceding page.

type Bs = BId $\overrightarrow{m}$ B, **value** obs_Bs: BS→Bs,
type Rs = RId $\overrightarrow{m}$ R, **value** obs_Rs: B→Rs,
type Ss = SId $\overrightarrow{m}$ S, **value** obs_Ss: R→Ss,
type S = C*. ■

72

Observers for Composite Parts

- Let the domain describer decide
  - that a type, A (or Δ), is composite
  - and that it consists of sub-parts of types B, C, ..., D.
- We can initially consider these types B, C, ..., D, as abstract types, or sorts, as we shall mostly call them.
- That means that there are the following formalisations:
  - **type** A, B, C, ..., D;
  - **value** obs_B: A→B, obs_C: A→C, ..., obs_D: A→D.
- We can also consider the types B, C, ..., D, as concrete types, 73
  - **type** Bc = TypBex, Cc = TypCex, ..., Dc = TypDex;
  - **value** obs_Bc: B→Bc, obs_Cc: C→Cc, ..., obs_Dc: D→Dc,
  - where TypBex, TypCex, ..., TypDex are type expressions as, for example, hinted at above.
- The prefix obs_ distinguishes part observers
  - from mereology observers (uid_, mereo_) and
  - attribute observers (attr_).

4.3 Properties 74

Endurants have properties. Properties are what makes up a parts (and materials) and, with property values distinguishes one part from another part and one material from another material. We name properties. Properties of parts and materials can, without loss of generality, be given distinct names. We let these names also be the property type name. Hence two parts (materials) of the same part type (material type) have the same set of property type names. Properties are all that 75 distinguishes parts (and materials). The part types (material types) in themselves do not express properties. They express a class of parts (respectively materials). All parts (materials) of the same type have the same property types. Parts (materials) of the different types have different sets of property types, 76

For pragmatic reasons we distinguish between three kinds of properties: unique identifiers, mereology, and attributes. If you "remove" a property from a part it "looses" its (former) part

type, to, in a sense, attain another part type: perhaps of another, existing one, or a new "created" one. *But we do not know* how to model *removal of a property* from an endurant value![31]

Example: 11 Atomic Part Property Kinds. We distinguish between two kinds of persons: 'living persons' and 'deceased persons'; they could be modelled by two different part types: LP: living person, with a set of properties, DP: deceased person, with a, most likely, different set of properties. All persons have been born, hence have a birth date (static attributes). Only deceased persons have a (well-defined) death date. All persons also have height and weight profiles (i.e., with dated values, i.e., dynamic attributes). One can always associate a unique identifier with each person. Persons are related, family-wise: have parents (living or deceased), (up to four known) grandparents, etc., may have brothers and sisters (zero or more), may have children (zero or more), etc. These family-relations can be considered the mereology for living persons. ■

4.3.1 Unique Identification

We can assume that all parts of the same part type can be uniquely distinguished, hence can be given unique identifications.

Unique Identification

- With every part, whether atomic or composite we shall associate a unique part identifier, of just unique identifier.
- Thus we shall associate with part type T
  - ⋙ the unique part type identifier type TI,
  - ⋙ and a unique part identifier observer function, uid_TI: T→TI.
- These associations (TI and uid_TI) are, however,
  - ⋙ usually expressed explicitly,
  - ⋙ whether they are ("subsequently") needed!

The unique identifier of a part can not be changed; hence we can say that no matter what a given part's property values may take on, that part cannot be confused with any other part.

Since we can talk about this concept of unique identification, we can abstractly describe it — and do not have to bother about any representation, that is, whether we can humanly observe unique identifiers.

4.3.2 Mereology

Mereology [30][32] (from the Greek μερος 'part') is *"the theory of part-hood relations: of the relations of part to whole and the relations of part to part within a whole"*. For pragmatic reasons we choose to model the mereology of a domain in either of two ways either by defining a concrete type as a model of the composite type, or by endowing the sub-parts of the composite part with structures of unique part identifiers. or by suitable combinations of these.

Example: 12 Container Bays, Etcetera: Mereology. First we show how to model indexed set of container bays, rows and stacks for the previous example.

- *Narrative:*

[31] And we see no need for describing such type-changes. Crude oil does not "morph" into fuel oil, diesel oil, kerosene and petroleum. Crude oil is consumed and the fractions result from distillation, for example, in an oil refinery.

[32] Achille Varzi: Mereology, http://plato.stanford.edu/entries/mereology/

- (i) An indexed set, bs:BS, of bays is a bijective map from unique bay identifiers, bid:BId, to bays, b:B.
- (ii) An indexed set, rs:RS, of rows is a bijective map from unique row identifiers, rid:RId, to rows, r:R.
- (iii) An indexed set, ss:SS, of stacks is a bijective map from unique stack identifiers, sid:SId, to stacks, s:S.
- (iv) A stack is a linear indexed sequence of containers, c:C.

85

- *Formalisation:*
  - (i) **type** BS, B, BId, **value** obs_Bs: BS→Bs — Bs=BId $\overrightarrow{m}$ B, (or obs_Bs: BS→(BId $\overrightarrow{m}$ B));
  - (ii) **type** RS, R, RId, **value** obs_Rs: RS→Rs — Rs=RId $\overrightarrow{m}$ R, (or obs_Rs: RS→(RId $\overrightarrow{m}$ R));
  - (iii) **type** SS, S, SId, — Ss=SId $\overrightarrow{m}$ S;
  - (iv) **type** C, — S=C*. ■

86

Example: 13 Transport Nets: Mereology. We show how to model a mereology for a transport net of links and hubs.

- *Narrative:*
  - (i) Hubs and links are endowed with unique hub, respectively link identifiers.
  - (ii) Each hub is furthermore endowed with a hub mereology which lists the unique link identifiers of all the links attached to the hub.
  - (iii) Each link is furthermore endowed with a link mereology which lists the set of the two unique hub identifiers of the hubs attached to the link.
  - (iv) Link identifiers of hubs and hub identifiers of links must designate hubs, respectively links of the net.

87

- *Formalisation:*

(i) **type** H, HI, L, LI;

value

(ii) uid_HI:H→HI, uid_LI:L→LI,

mereo_H:H→LI-**set**, mereo_L:L→HI-**set**,

axiom

(iii) ∀ l:L • **card** mereo_L(l) = 2

(iv) ∀ n:N, l:L, h:H • l ∈ obs_Ls(obs_LS(n)) ∧ h ∈ obs_Hs(obs_HS(n))
∀ hi:HI • hi ∈ mereo_L(l) ⇒
∃ h′:H•h′ ∈ obs_Hs(obs_HS(n)) ∧ uid_HI(h)=hi
∧ ∀ li:LI • li ∈ mereo_H(h) ⇒
∃ l′:L•l′ ∈ obs_Ls(obs_LS(n)) ∧ uid_LI(l)=li

■

88

Concrete Models of Mereology

The concrete mereology example models above illustrated maps and sequences as such models.

- In general we can model mereologies in terms of

- ⋙ (i) sets: A-**set**,
- ⋙ (ii) Cartesians: $A_1 \times A_2 \times ... \times A_m$,
- ⋙ (iii) lists: A^*, and
- ⋙ (iv) maps: $A \xrightarrow[m]{} B$,

where A, A_1, A_2,...,A_m and B are types [we assume that they are type names] and where the A_1, A_2,...,A_m type names need not be distinct.

- Additional concrete types, say D, can be defined by concrete type definitions, D=E, where E is either of the type expressions (i–iv) given above or (v) $E_i|E_j$, or (vi) (E_i). where E_k (for suitable k) are either of (i–vi).
- Finally it may be necessary to express well-formedness predicates for concretely modelled mereologies.

89

Abstract Models of Mereology

Abstractly modelling mereology of parts, to us, means the following.

- With part types P_1, P_2, ..., P_n
  - ⋙ is associated the unique part identifier types, Π_1, Π_2, ..., Π_n,
  - ⋙ that is uid_Π_i: $P_i \to \Pi_i$ for $i \in \{1..n\}$,
- and with each part type, P_i,
  - ⋙ is then associated a mereology observer,
  - ⋙ mereo_P_i: $P_i \to \Pi_j$-**set**$\times \Pi_k$-**set**$\times ... \times \Pi_\ell$-**set**,
- such that for all p:Pi we have that
  - ⋙ if mereo_P_i(p) = ({..., π_{j_a}, ...},{..., π_{k_b}, ...},...,{..., π_{ℓ_c}, ...})
  - ⋙ for $i, j, k, ...\ell \in \{1..n\}$
  - ⋙ then part p:P_i is connected (related) to the parts identified by ..., π_{j_a}, ... π_{k_b}, ..., π_{ℓ_c}, ...
- Finally it may be necessary to express axioms for abstractly modelled mereologies.

The prefixes uid_ and mereo_ distinguish mereology observers from part observers (obs_) and attribute observers (attr_). How parts are related to other parts is really a modelling choice, made by the domain describer. It is not necessarily something that is obvious from observing the parts.

90
91

Example: 14 Pipelines: A Physical Mereology. Let pipes of a pipe line be composed with valves, pumps, forks and joins of that pipe line. Pipes, valves, pumps, forks and joins (i.e., pipe line units) are given unique pipe, valve, pump, fork and join identifiers. A mereology for the pipe line could now endow pipes, valves and pumps with one input unique identifier, that of the predecessor successor unit, and one output unique identifier, that of the successor unit. Forks would then be endowed with two input unique identifiers, and one out put unique identifier; and joins "the other way around".

92

Example: 15 Documents: A Conceptual Mereology. The mereology of, for example, this document, that is, of the lecture notes, is determined by the author. There unfolds, while writing the document, a set of unique identifiers for section, subsection, sub-subsection, paragraph, etc., units. and between texts of a "paper version" of the document and slides of a "slides version" of the document. This occurs as the author necessarily inserts cross-references, in unit texts to other units, and from unit texts to other documents (i.e., 'citations'); and while inserting "page"

93

shifts for the slides. From those inserted references there emerges what we could call the document mereology.
So the determination of a, or the, mereology of composite parts is either given by physical considerations, or are given by (more-or-less) logical (or other) considerations, or by combinations of these. The "design" of mereologies improves with experience. 94

Example: 16 Pipelines: Mereology. We divert from our line of examples centered around transport nets and, to some degree, container transport, to bring a second, in a series of examples on pipelines (for liquid or gaseous material flow). In this example we shall, for the first time, be enumerating the narrative clauses and let this enumeration carry over to the formalisation for easy cross-reference. 95

1. A pipeline consists of connected units, u:U.
2. Units have unique identifiers.
3. And units have mereologies, ui:UI:
   a pump[33], pu:Pu, pipe, pi:Pi, and valve[34], va:Va, units have one input connector and one output connector;
   b fork, fo:Fo, [join, jo:Jo] units have one [two] input connector[s] and two [one] output connector[s];
   c well[35], we:We, [sink[36], si:Si] units have zero [one] input connector and one [zero] output connector.
   d Connectors of a unit are designated by the unit identifier of the connected unit.
   e The auxiliary sel_UIs_in selector funtion selects the unique identifiers of pipeline units providing input to a unit;
   f sel_UIs_out selects unique identifiers of output recipients.

96

```
type
1. U = Pu | Pi | Va | Fo | Jo | Si | We
2. UI
value
2. uid_U: U → UI
3. mereo_U: U → UI-set × UI-set
3. wf_mereo_U: U → Bool
3. wf_mereo_U(u) ≡
3a.     is_(Pu|Pi|Va)(u) → card iusi = 1 = card ouis,
3b.     is_Fo(u) → card iuis = 1 ∧ card ouis = 2,
3b.     is_Jo(u) → card iuis = 2 ∧ card ouis = 1,
3c.     is_We(u) → card iuis = 0 ∧ card ouis = 1,
3d.     is_Si(u) → card iuis = 1 ∧ card ouis = 0

3e. sel_UIs_in
3e. sel_UIs_in(u) ≡ let (iuis,_)=mereo_U(u) in iuis end
3f. sel_out: U → UI-set
3f. sel_UIs_out(u) ≡ let (_,ouis)=mereo_U(u) in ouis end
```

97

We omit treatment of axioms for pipeline units being indeed connected to existing other pipeline units. We refer to Example 24 on page 29 and 25 on page 30.

[33] We abstract from such distinctions between *oil pipeline pumps* and *gas pipeline compressors*.
[34] We abstract *regulator stations* (where the pipeline operator can release some of the pressure from the pipeline) and *block valve stations* (where the operator can isolate any segment of a pipeline for maintenance work or isolate a rupture or leak) into valves.
[35] We abstract wells into initial injection stations where the liquid or gaseous material is injected into the line.
[36] We abstract partial and final delivery stations into sinks, places where the material is delivered to an agent outside the pipeline system.

4.3.3 Attributes

98

By an attribute of a part, p:P, we shall understand some observable property, some phenomenon, that is not a sub-part of p but which characterises p such that all parts of type P have that attribute and such that "removing" that attribute from p (if such was possible) "renders" the type of p undefined.

99 We ascribe types to attributes — not, therefore, to be confused with types of (their) parts.

Example: 17 Attributes. Example attributes of links of a transport net are: length LEN, location LOC, state LΣ and state space LΩ, Example attributes of a person could be: name NAM,
100 birth date BID, gender GDR, weight WGT, height HGT and address ADR. Example attributes of a transport net could be: name of the net, legal owner of the net, a map of the net, etc. Example attributes of a container vessel could be: name of container vessel, vessel dimensions, vessel tonnage (TEU), vessel owner, current stowage plan, current voyage plan, etc. ■
101

[1] Static and Dynamic Attributes By a static attribute we mean an attribute (of a part) whose value remains fixed. By a dynamic attribute we mean an attribute (of a part) whose value may
102 vary.

Example: 18 Static and Dynamic Attributes. The length and location attributes of links are static. The state and state space attributes of links and hubs are dynamic. The birth-date attribute of a person is considered static. The height and weight attributes of a person are dynamic. The map of a transport net may be considered dynamic. The current stowage and the current voyage plans of a vessel should be considered dynamic. ■
103

Attribute Types and Observers

- Let the domain describer decide that parts of type P
- have attributes of types A_1, A_2, ..., A_t.
- This means that the following two formal clauses arise:
  - P, A_1, A_2, ..., A_t and
  - attr_A_1:P→A_1, attr_A_2:P→A_2, ..., attr_A_t:P→A_t

104

- We may wish to annotate the list of attribute type names as to whether they are static or dynamic, that is,
  - whether values of some attribute type
  - vary or
  - remain fixed.
- The prefix attr_ distinguishes attribute observers from part observers (obs_) and mereology observers (uid_, mereo_).

4.4 Shared Attributes and Properties

105

Shared attributes and shared properties play an important rôle in understanding domains.

4.4.1 Attribute Naming

We now *impose a restriction* on the naming of part attributes. If attributes of two different parts of different part types are identically named then attributes must be somehow related, over time! The "somehow" relationship must be described. 106

Example: 19 Shared Bus Time Tables. Let our domain include that of *bus time tables* for *busses* on a *bus transport net* as described in many examples in this tutorial. We can then imagine a *bus transport net* as containing the following parts: a *net*, a *bus management system*, a set of *busses*. For the sake of argument we consider a *bus time table* to be an attribute of the *bus management system*. And we also consider *bus time tables* to be attributes of *busses*. 107

We think of the *bus time table* of a *bus* to be that subset of the *bus management system bus time table* which corresponds to the *bus*' *line number*. By saying that *bus time tables* "corresponds" to well-defined subsets of the *bus management system bus time table* we mean the following The value of the *bus bus time table* must at every time be equal to the corresponding *bus line entry* in the *bus management system bus time table*. ■

4.4.2 Attribute Sharing 108

We say that two parts, of no matter what part type, *share* an attribute, if the following is the case: the corresponding part types (and hence the parts) have identically named attributes. We say that identically named attributes designate shared attributes. We do not present the corresponding invariants over parts with identically named attributes.

4.5 Shared Properties 109

We say that two parts, of no matter what part type, *share* a property, if either of the following is the case: (i) either the corresponding part types (and hence the parts) have shared attributes; (ii) or the unique identifier type of one of the parts potentially[37] is in the mereology type of the other part; (iii) or both. We do not present the corresponding invariants over parts with shared properties.

4.6 Summary of Discrete Endurants 110

We have introduced the endurant notions of atomic parts and composite parts: part types, part observers (**obs_**), sort observers, and concrete type observers; part properties: unique identifiers: unique part identifier observers (**uid_**), unique part identifier types, mereology: part mereologies, part mereology observers (**mereo_**); and attributes: attribute observers (**attr_**) and attribute types. 111

The unique identifier property cannot necessarily be observed: it is an abstract concept and can be objectively "assigned". That is: unique identifiers are not required to be manifest. The mereology property also cannot usually be observed: it is also an abstract concept, but can be deduced from careful analysis. That is: mereology is not required to be manifest. The attributes can be observed: usually by simple physical measurements, or by deduction from (conceptual) facts, That is: attributes are usually only "indirectly" manifest. 112

Discrete Endurant Modelling

Faced with a phenomenon the domain analyser has to decide

- whether that phenomenon is an entity or not, that is, whether
  - ⋘ an endurant or
  - ⋘ a perdurant or
  - ⋘ neither.

[37] For given parts they may thus "share" these identifiers, or they may not.

- If endurant and if discrete, then whether it is
  - an atomic part or
  - a composite part.
- Then the domain analyser must decide on its type,
  - whether an abstract type (a sort)
  - or a concrete type, and, if so, which concrete form.
- Next the unique identifier and the mereology of the part type (e.g., P) must be dealt with: 113
  - type name (e.g., PI) for and, hence, unique identifier observer name (uid_PI) of unique identifiers and the
  - part mereology types and mereology observer name (mereo_P).
- Finally the designer must decide on the part type attributes for parts p:P:
  - for each such a suitable attribute type name, for example, A_i for suitable i,
  - a corresponding attribute observer signature, attr_A_i:P→A_i,
  - and whether an attribute is considered static or dynamic.

5 Continuous Endurants: Materials 116

We refer to Sect. 3.2 on page 17.

Let us start with examples of materials.

Example: 20 Materials. Examples of endurant continuous entities are such as coal, air, natural gas, grain, sand, iron ore[38], minerals, crude oil, solid waste, sewage, steam and water. ■

The above materials are either liquid materials (crude oil, sewage, water), gaseous materials (air, 117 gas, steam), or granular materials (coal, grain, sand, iron ore, mineral, or solid waste).

Endurant continuous entities, or materials as we shall call them, are the core endurants of process domains, that is, domains in which those materials *form the basis* for their *"raison d'être"*.

Example: 21 Material Processing. (i) Oil or gas materials are ubiquitous to pipeline systems. (ii) Sewage is ubiquitous to, well, sewage systems. (iii) Water is ubiquitous to systems composed from reservoirs, tunnels and aqueducts which again are ubiquitous to hydro-electric power plants or irrigation systems. ■ 118

Ubiquitous means 'everywhere'. A continuous entity, that is, a material is a core material, if it is "somehow related" to one or more parts of a domain.

5.1 "Somehow Related" Parts and Materials

119 We explain our use of the term "somehow related".

Example: 22 "Somehow Related" Parts and Materials. (i) Oil is pumped from wells, runs through pipes, is "lifted" by pumps, diverted by forks, "runs together" by means of joins, and is delivered to sinks – *and is hence a core endurant.* (ii) Grain is delivered to silos by trucks, piped through a network of pipes, forks and valves to vessels, etc. – *and is hence a core endurant.* (iii) Gravel, minerals (including) iron ore is mined, conveyed by belts to lorries or trains or cargo

[38] – whether molten or not

vessels and finally deposited. For minerals typically in mineral processing plants – *and is hence a core endurant*. (iv) Iron ore, for example, is conveyed into smelters, roasted, reduced and fluxed, mixed with other mineral ores to produced a molten, pure metal, which is then "collected" into ingots, etc. – *and is hence a core endurant* ■

5.2 Material Observers

120

When analysing domains a key question, in view of the above notion of core continuous endurants (i.e., materials) is therefore: does the domain embody a notion of core continuous endurants (i.e., materials); if so, then identify these "early on" in the domain analysis. Identifying materials — their types and attributes — is slightly different from identifying discrete endurants, i.e., parts.

121

Example: 23 Pipelines: Core Continuous Endurant. The core continuous endurant, i.e., material, of (say oil) pipelines is, yes, oil:

```
type
    O  material
value
    obs_Materials: PLS → O
```

The keyword **material** is a pragmatic. ■
Materials are "few and far between" as compared to parts, we choose to mark the type definitions which designate materials with the keyword **material**. In contrast, we do not mark the type definitions which designate parts with the keyword **discrete**. First we do not associate the notion of atomicity or composition with a material. Materials are continuous. Second, amongst the attributes, none have to do with geographic (or cadestral) matters. Materials are moved. And materials have no unique identification or mereology. No "part"[39] of a material distinguishes it from other "parts". But they do have other attributes when occurring in connection with, that is, related to parts, for example, volume or weight.

122

123

Example: 24 Pipelines: Parts and Materials. We refer to Example 16 on page 25.

4. From an oil pipeline system one can, amongst others,
   a observe the finite set of all its pipeline bodies,
   b units are composite and consists of a unit,
   c and the oil, even if presently, at time of observation, empty of oil.
5. Whether the pipeline is an oil or a gas pipeline is an attribute of the pipeline system.
   a The volume of material that can be contained in a unit is an attribute of that unit.
   b There is an auxiliary function which estimates the volume of a given "amount" of oil.
   c The observed oil of a unit must be less than or equal to the volume that can be contained by the unit.

124

```
type
4.    PLS, B, U, O, Vol
value
4a.  obs_Bs: PLS → B-set
4b.  obs_U: B → U
4c.  obs_O: B → O
5.   attr_PLS_Type: PLS → {"oil"|"gas"}
5a.  attr_Vol: U → Vol
5b.   vol: O → Vol
axiom
5c.  ∀ pls:PLS,b:B•b ∈ obs_Bs(pls)⇒vol(obs_O(b))≤attr_Vol(obs_U(b))
```

[39] The term part is not the technical term for discrete endurants, but the more conventional term.

Notice how bodies are composite and consists of a discrete, atomic part, the unit, and a material endurant, the oil. We refer to Example 25. ■

5.3 Material Properties

125

These are some of the key concerns in domains focused on materials: transport, flows, leaks and losses, and input to systems and output from systems, Other concerns are in the direction of dynamic behaviours of materials focused domains (mining and production), including stability, periodicity, bifurcation and ergodicity. In this tutorial we shall, when dealing with systems focused on materials, concentrate on modelling techniques for transport, flows, leaks and losses, and input
126 to systems and output from systems.

Formal specification languages like `Alloy` [58], `Event B` [1], `CASL` [33]`CafeOBJ` [42], `RAISE` [45], `VDM` [24, 25, 41] and `Z` [99] do not embody the mathematical calculus notions of continuity, hence do not "exhibit" neither differential equations nor integrals. Hence cannot formalise dynamic systems within these formal specification languages. We refer to Sect. 9 where we discuss these issues at
127 some length.

Example: 25 Pipelines: Parts and Material Properties. We refer to Examples 16 on page 25 and 24 on the preceding page.

6. Properties of pipeline units additionally include such which are concerned with flows (F) and leaks (L) of materials[40]:
   a current flow of material into a unit input connector,
   b maximum flow of material into a unit input connector while maintaining laminar flow,
   c current flow of material out of a unit output connector,
   d maximum flow of material out of a unit output connector while maintaining laminar flow,
   e current leak of material at a unit input connector,
   f maximum guaranteed leak of material at a unit input connector,
   g current leak of material at a unit input connector,
   h maximum guaranteed leak of material at a unit input connector,
   i current leak of material from "within" a unit,
   j maximum guaranteed leak of material from "within" a unit.

128

type
6. F, L

value
6a. attr_cur_iF: U → UI → F
6b. attr_max_iF: U → UI → F
6c. attr_cur_oF: U → UI → F
6d. attr_max_oF: U → UI → F
6e. attr_cur_iL: U → UI → L
6f. attr_max_iL: U → UI → L
6g. attr_cur_oL: U → UI → L
6h. attr_max_oL: U → UI → L
6i. attr_cur_L: U → L
6j. attr_max_L: U → L

129

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 as dynamic attributes.

[40] Here we think of flows and leaks as measured in terms of volume per time unit.

7. Properties of pipeline materials may additionally include
   a kind of material[41],
   b paraffins,
   c naphtenes,
   d aromatics,
   e asphatics,
   f viscosity,
   g etcetera.

We leave it to the reader to provide the formalisations. ■

5.4 Material Laws of Flows and Leaks 130

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. 131

Example: 26 Pipelines: Intra Unit Flow and Leak Law.

8. For every unit of a pipeline system, except the well and the sink units, the following law apply.
9. 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.

132

```
axiom
8.  ∀ pls:PLS,b:B\We\Si,u:U •
8.      b ∈ obs_Bs(pls)∧u=obs_U(b)⇒
8.      let (iuis,ouis) = mereo_U(u) in
9.      sum_cur_iF(iuis)(u) =
9a.        sum_cur_iL(iuis)(u)
9b.      ⊕ attr_cur_L(u)
9c.      ⊕ sum_cur_oF(ouis)(u)
9d.      ⊕ sum_cur_oL(ouis)(u)
8.      end
```

133

10. The sum_cur_iF (cf. Item 9) sums current input flows over all input connectors.
11. The sum_cur_iL (cf. Item 9a) sums current input leaks over all input connectors.
12. The sum_cur_oF (cf. Item 9c) sums current output flows over all output connectors.
13. The sum_cur_oL (cf. Item 9d) sums current output leaks over all output connectors.

[41] For example `Brent Blend` Crude Oil

10. sum_cur_iF: UI-**set** → U → F
10. sum_cur_iF(iuis)(u) ≡ ⊕ {attr_cur_iF(ui)(u)|ui:UI•ui ∈ iuis}
11. sum_cur_iL: UI-**set** → U → L
11. sum_cur_iL(iuis)(u) ≡ ⊕ {attr_cur_iL(ui)(u)|ui:UI•ui ∈ iuis}
12. sum_cur_oF: UI-**set** → U → F
12. sum_cur_oF(ouis)(u) ≡ ⊕ {attr_cur_iF(ui)(u)|ui:UI•ui ∈ ouis}
13. sum_cur_oL: UI-**set** → U → L
13. 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: 27 Pipelines: Inter Unit Flow and Leak Law.

14. 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.

14. ∀ pls:PLS,b,b′:B,u,u′:U•
14. {b,b′}⊆obs_Bs(pls)∧b≠b′∧u′=obs_U(b′)
14. ∧ **let** (iuis,ouis)=mereo_U(u),(iuis′,ouis′)=mereo_U(u′),
14. ui=uid_U(u),ui′=uid_U(u′) **in**
14. ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
14a. attr_cur_oF(us′)(ui′) − attr_leak_oF(us′)(ui′)
14b. = attr_cur_iF(us)(ui) + attr_leak_iF(us)(ui)
14. **end**
14. **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. ■

Continuous Endurant Modelling

As one of the first steps

- in domain analysis
- determine if the domain is materials-focused.

If so, then determine

- the material types,

  type M1, M2, ... Mn **material**

- the parts, that is, the part types, with which the materials are "somehow related",

  value obs_Mi: Pi → Mi, obs_Mj: Pj → Mj, ..., obs_Mk: Pk → Mk

- the relevant flow or transport and/or leak or loss attributes, if any,
- and the possible laws related to these attributes.

6 States 137

6.1 General

The above Wikipedia characterisation of the concept of perdurant mentioned time, but implied a concept that we shall call state. In this version of this tutorial we shall not cover the modelling of time phenomena — but we shall model that some actions occur before others. 138

By a state we shall understand a collection of parts such that each of these parts have dynamic attributes. We can characterise the state by giving it a type, for example, Σ, where the state type definition $\Sigma = \mathsf{S}_1 \times \mathsf{S}_2 \times \cdots \times \mathsf{S}_s$ assembles the types of the parts making up the state — where we assume that types S_1, S_2, ..., S_s are types of parts such that no S_i is a sub-part (of a subpart, ...) of some S_j, and such that each part has dynamic attributes. 139

Example: 28 Net and Vessel States. We may consider a transport net, n:N, to represent a state (subject to the actions of maintaining a net: adding or removing a hub, adding or removing a link, etc.). We may also consider a hub, h:H, to represent a state (subject to the changing of a hub traffic signal: from red to green, etc., for specific directions through the hub). We may consider a container vessel to represent a state (subject to adding or removing containers from, respectively onto the top of stacks). ■

Thus the context determines how wide a scope the domain designer chooses for the state concept.

6.2 State Invariants 140

States are subject to invariants.

Example: 29 State Invariants: Transport Nets. Nets, hubs and links were first introduced in Example 3 on page 10 – and were and will be prominent in this tutorial, to wit, Examples 8–17 and 30– **??** on page ??. Net hubs and links may be inserted into and removed from nets. Thus is also introduced changes to the net mereology. Yet, the axioms, as illustrated in Example 13, must remain invariant. Likewise changes to dynamic attributes may well be subject to the holding of certain well-formedness constraints. We will illustrate this claim. 141

With each hub we associate a hub [link] state and a hub [link] state space.

15. A hub [link] state models the permissible routes from hub input links to (same) hub output links [respectively through a link].

16. A hub [link] state space models the possible set of hub [link] states that a hub [link] is intended to "occupy".

type
15. HΣ = (LI × LI)-set, LΣ = HI-set
16. HΩ = HΣ-set, LΩ = LΣ-set

value
15. attr_HΣ: H → HΣ, attr_LΣ: L → LΣ
16. attr_HΩ: H → HΩ, attr_LΩ: L → LΩ

142

17. For any given hub, h, with links, $l_1, l_2, ..., l_n$ incident upon (i.e., also emanating from) that hub, each hub state in the hub state space

18. must only contain such pairs of (not necessarily distinct) link identifiers that are identifiers of $l_1, l_2, ..., l_n$.

value
17. wf_HΩ: H → **Bool**
17. wf_HΩ(h) ≡ ∀ hσ:HΣ • hσ ∈ attr_HΩ(h) ⇒ wf_HΣ(h)

17. wf_HΣ: H → **Bool**
17. wf_HΣ(h) ≡
18. ∀ (li,li′):(LI×LI)•(li,li′)∈ attr_HΣ(h) ⇒ {li,li′} ⊆ mereo_H(h)

143

This well-formedness criterion is part of the state invariant over nets. We never write down the full state invariant for nets. It is tacitly assume to be the collection of all the axioms and well-formedness predicates over net parts. ■

7 A Final Note on Endurant Properties 144

The properties of parts and materials are fully captured by (i) the unique part identifiers, (ii) the part mereology and (iii) the full set ofpart attributes and material attributes We therefore postulate a property function when when applied to a part or a material yield this triplet, (i–iii), of properties in a suitable structure.

type
Props = {|PI|**nil**|} × {|(PI-**set**×...×PI-**set**)|**nil**|} × Attrs
value
props: Part|Material → Props

145

where Part stands for a part type, Material stands for a material type, PI stand for unique part identifiers and PI-**set**×...×PI-**set** for part mereologies. The {|...|} denotes a proper specification
146 language sub-type and **nil** denotes the empty type.
147

8 Discrete Perdurants 148

8.1 General

From Wikipedia: *Perdurant: Also known as occurrent, accident or happening. Perdurants are those entities for which only a fragment exists if we look at them at any given snapshot in time. When we freeze time we can only see a fragment of the perdurant. Perdurants are often what we know as processes, for example 'running'. If we freeze time then we only see a fragment of the running, without any previous knowledge one might not even be able to determine the actual process as being a process*
149 *of running. Other examples include an activation, a kiss, or a procedure.*

We shall consider actions and events to occur instantaneously, that is, in time, but taking no time Therefore we shall consider actions and events to be perdurants.

8.2 Discrete Actions 150

By a function we understand a thing which when applied to a value, called its argument, yields a value, called its result.
151 An action is a function invoked on a state value and is one that potentially changes that value.

Example: 30 Transport Net and Container Vessel Actions.

- *Inserting* and *removing* hubs and links in a net are considered actions.
- *Setting* the traffic signals for a hub (which has such signals) is considered an action.
- *Loading* and *unloading* containers from or unto the top of a container stack are considered actions. ■

8.2.1 An Aside on Actions

152

Think'st thou existence doth depend on time?
It doth; but actions are our epochs.

George Gordon Noel Byron,
Lord Byron (1788-1824) Manfred. Act II. Sc. 1.

"An action is something an agent does that was 'intentional under some description'" [35, Davidson 1980, Essay 3]. That is, actions are performed by agents. We shall not yet go into any deeper treatment of agency or agents. We shall do so in Sect. 8.4. Agents will here, for simplicity, be considered behaviours, and are treated in Sect. 8.4. As to the relation between intention and action 153
we note that Davidson wrote: 'intentional under some description' and take that as our cue: the agent follows a script, that is, a behaviour description, and invokes actions accordingly, that is, follow, or honours that script. The philosophical notion of 'action' is over-viewed in [98].

We observe actions in the domain but describe "their underlying" functions. Thus we abstract from the times at which actions occur.

8.2.2 Action Signatures

154

By an action signature we understand a quadruple: a function name, a function definition set type expression, a total or partial function designator ($\rightarrow$, respectively $\xrightarrow{\sim}$), and a function image set type expression: fct_name: A $\rightarrow$ Σ $(\rightarrow|\xrightarrow{\sim})$ Σ [$\times$ R], where $(X \mid Y)$ means either X or Y, and $[Z]$ means optional Z.

Example: 31 Action Signatures: Nets and Vessels.

insert_Hub: N$\rightarrow$H$\xrightarrow{\sim}$N;
remove_Hub: N$\rightarrow$HI$\xrightarrow{\sim}$N;
set_Hub_Signal: N$\rightarrow$HI$\xrightarrow{\sim}$H$\Sigma\xrightarrow{\sim}$N
load_Container: V$\rightarrow$C$\rightarrow$StackId$\xrightarrow{\sim}$V; and
unload_Container: V$\rightarrow$StackId$\xrightarrow{\sim}$(V$\times$C). ■

8.2.3 Action Definitions

155

There are a number of ways in which to characterise an action. One way is to characterise its underlying function by a pair of predicates: precondition: a predicate over function arguments — which includes the state, and postcondition: a predicate over function arguments, a proper argument state and the desired result state. If the precondition holds, i.e., is **true**, then the arguments, including the argument state, forms a proper 'input' to the action. If the postcondition holds, assuming that the precondition held, then the resulting state [and possibly a yielded, additional "result" (R)] is as they would be had the function been applied. 156

Example: 32 Transport Nets: Insert Hub Action. We give one example.

19. The insert action applies to a net and a hub and conditionally yields an updated net.
    a The condition is that there must not be a hub in the "argument" net with the same unique hub identifier as that of the hub to be inserted and
    b the hub to be inserted does not initially designate links with which it is to be connected.
    c The updated net contains all the hubs of the initial net "plus" the new hub.
    d and the same links.

157

value
19. insert_H: N $\rightarrow$ H $\xrightarrow{\sim}$ N
19. insert_H(n)(h) **as** n′, **pre**: pre_insert_H(n)(h), **post**: post_insert_H(n)(h)

```
19a. pre_insert_H(n)(h) ≡
19a.   ~∃ h′:H • h′ ∈ obs_Hs(n) ∧ uid_HI(h)=uid_HI(h′)
19b.     ∧ mereo_H(h) = {}

19c. post_insert_H(n)(h)(n′) ≡
19c.   obs_Hs(n) ∪ {h} = obs_Hs(n′)
19d.   ∧ obs_Ls(n) = obs_Ls(n′)
```

We refer to Appendix Sects. B.3.2–B.3.4 There you will find definitions of insert_link, remove_hub and remove_link action functions.

What is not expressed, but tacitly assume in the above pre- and post-conditions is that the state, here n, satisfy invariant criteria before (i.e. n) and after (i.e., n') actions, whether these be implied by axioms or by well-formedness predicates. over parts. This remark applies to any definition of actions, events and behaviours.

Example: 33 Action: Remove Container from Vessel. We give the second of two examples.

20. The remove_Container_from_Vessel action applies to a vessel and a stack address and conditionally yields an updated vessel and a container.
    a We express the 'remove from vessel' function primarily by means of an auxiliary function remove_C_from_BS, remove_C_from_BS(obs_BS(v))(stid), and some further post-condition on the before and after vessel states (cf. Item 20d).
    b The remove_C_from_BS function yields a pair: an updated set of bays and a container.
    c When obs_erving the BayS from the updated vessel, v′, and pairing that with what is assumed to be a vessel, then one shall obtain the result of remove_C_from_BS(obs_BS(v))(stid).
    d Updating, by means of remove_C_from_BS(obs_BS(v))(stid), the bays of a vessel must leave all other properties of the vessel unchanged.
21. The pre-condition for remove_C_from_BS(bs)(stid) is
    a that stid is a valid_address in bs, and
    b that the stack in bs designated by stid is non_empty.
22. The post-condition for remove_C_from_BS(bs)(stid) wrt. the updated bays, bs′, is
    a that the yielded container, i.e., c, is obtained, get_C(bs)(stid), from the top of the non-empty, designated stack,
    b that the mereology of bs′ is unchanged, unchanged_mereology(bs,bs′). wrt. bs. ,
    c that the stack designated by stid in the "input" state, bs, is popped, popped_designated_stack(bs,bs′)(stid), and
    d that all other stacks are unchanged in bs′ wrt. bs, unchanged_non_designated_stacks(bs,bs′)(stid).

```
value
20.  remove_C_from_V: V → StackId ~→ (V×C)
20.  remove_C_from_V(v)(stid) as (v′,c)
20c.   (obs_BS(v′),c) = remove_C_from_BS(obs_BS(v))(stid)
20d.  ∧ props(v)=props(v″)

20b. remove_C_from_BS: BS → StackId → (BS×C)
20a. remove_C_from_BS(bs)(stid) as (bs′,c)
21a.       pre: valid_address(bs)(stid)
21b.        ∧ non_empty_designated_stack(bs)(stid)
```

22a. **post**: c = get_C(bs)(stid)
22b. ∧ unchanged_mereology(bs,bs′)
22c. ∧ popped_designated_stack(bs,bs′)(stid)
22d. ∧ unchanged_non_designated_stacks(bs,bs′)(stid)

The props function was introduced in Sect. 7 on page 34. 162

This example hints at *a theory of container vessel bays, rows and stacks*. More on that is found in Appendix C on page 107. There you will find explanations of the valid_address (Item 185 on page 109), non_empty_designated_stack (Item 186 on page 110), unchanged_mereology (Item 187), popped_designated_stack (Item 188) and unchanged_non_designated_stacks (Item 189 on page 111) functions. ■

There are other ways of defining functions. But the form of these are not material to the aims of this tutorial. 163

Modelling Actions

- The domain describer has decided that an entity is a perdurant and is, or represents an action: was *"done by an agent and intentionally under some description"* [35].
  - The domain describer has further decided that the observed action is of a class of actions — of the "same kind" — that need be described.
  - By actions of the 'same kind' is meant that these can be described by the same function signature and function definition.

164

- First the domain describer must decide on the underlying function signature.
  - The argument type and the result type of the signature are those of either previously identified
    - parts and/or materials,
    - unique part identifiers, and/or
    - attributes.

165

- Sooner or later the domain describer must decide on the function definition.
  - The form[42] must be decided upon.
  - For pre/post-condition forms it appears to be convenient to have developed, "on the side", a theory of mereology for the part types involved in the function signature.

8.3 Discrete Events 166

By an event we understand a state change resulting indirectly from an unexpected application of a function, that is, that function was performed "surreptitiously". Events can be characterised by a pair of (before and after) states, a predicate over these and, optionally, a time or time interval.

Events are thus like actions: change states, but are usually either caused by "previous" actions, or caused by "an outside action". 167

Example: 34 Events. *Container vessel:* A container falls overboard sometimes between times t and t'. *Financial service industry:* A bank goes bankrupt sometimes between times t and t'.

[42] Only the pre/post-condition form has so far been illustrated. Other function definition forms, incl. predicate functions, will emerge in further examples below.

Health care: A patient dies sometimes between times t and t'. *Pipeline system:* A pipe breaks sometimes between times t and t'. *Transportation:* A link "disappears" sometimes between times t and t'.

8.3.1 An Aside on Events

168

We may observe an event, and then we do so at a specific time or during a specific time interval. But we wish to describe, not a specific event but a class of events of "the same kind". In this tutorial we therefore do not ascribe time points or time intervals with the occurrences of events[43].

8.3.2 Event Signatures

169

An event signature is a predicate signature having an event name, a pair of state types ($\Sigma \times \Sigma$), a total function space operator ($\rightarrow$) and a **Bool**ean type constant: evt: $(\Sigma \times \Sigma) \rightarrow$ **Bool**. Sometimes there may be a good reason for indicating the type, ET, of an event cause value, if such a value can be identified: evt: ET $\times$ $(\Sigma \times \Sigma) \rightarrow$ **Bool**.

8.3.3 Event Definitions

170

An event definition takes the form of a predicate definition: A predicate name and argument list, usually just a state pair, an existential quantification over some part (of the state) or over some dynamic attribute of some part (of the state) or combinations of the above a pre-condition expression over the input argument(s), an implication symbol ($\Rightarrow$), and a post-condition expression over the argument(s). $\mathsf{evt}(\sigma, \sigma') = \exists\ (\mathsf{ev{:}ET}) \bullet \mathsf{pre\_evt}(\mathsf{ev})(\sigma) \Rightarrow \mathsf{post\_evt}(\mathsf{ev})(\sigma, \sigma')$. There may be
171 variations to the above form.

Example: 35 Narrative of Link Event. The disappearance of a link in a net, for example due to a mud slide, or a bridge falling down, or a fire in a road tunnel, can, for example be described as follows:

23. Link disappearance is expressed as a predicate on the "before" and "after" states of the net. The predicate identifies the "missing" ℓink (!).
24. Before the disappearance of link ℓ in net n
    a the hubs h' and h'' connected to link ℓ
    b were connected to links identified by $\{l'_1, l'_2, \ldots, l'_p\}$ respectively $\{l''_1, l''_2, \ldots, l''_q\}$
    c where, for example, l'_i, l''_j are the same and equal to $\mathsf{uid\_\Pi}(\ell)$.

172

25. After link ℓ disappearance there are instead
    a two separate links, ℓ_i and ℓ_j, "truncations" of ℓ
    b and two new hubs h''' and h''''
    c such that ℓ_i connects h' and h''' and
    d ℓ_j connects h'' and h'''';
    e Existing hubs h' and h'' now have mereology
        i. $\{l'_1, l'_2, \ldots, l'_p\} \setminus \{\mathsf{uid\_\Pi}(\ell)\} \cup \{\mathsf{uid\_\Pi}(\ell_i)\}$ respectively
        ii. $\{l''_1, l''_2, \ldots, l''_q\} \setminus \{\mathsf{uid\_\Pi}(\ell)\} \cup \{\mathsf{uid\_\Pi}(\ell_j)\}$
26. All other hubs and links of n are unaffected. ■

173

Example: 36 Formalisation of Link Event. Continuing Example 35 above:

[43] As we do not ascribe time points or time intervals with neither actions nor behaviours.

23. link_disappearance: N × N → **Bool**
23. link_disappearance(n,n′) ≡
23. ∃ ℓ:L • pre_link_dis(n,ℓ) ⇒ post_link_dis(n,ℓ,n′)

24. pre_link_dis: N × L → **Bool**
24. pre_link_dis(n,ℓ) ≡ ℓ ∈ obs_Ls(n)

174

27. We shall "explain" *link disappearance* as the combined, instantaneous effect of
    a first a remove link "event" where the removed link connected hubs hi_j and hi_k;
    b then the insertion of two new, "fresh" hubs, h_α and h_β;
    c "followed" by the insertion of two new, "fresh" links $l_{j\alpha}$ and $l_{k\beta}$ such that
        i. $l_{j\alpha}$ connects hi_j and h_α and
        ii. $l_{k\beta}$ connects hi_k and $h_{k\beta}$

175

value
27. post_link_dis(n,ℓ,n′) ≡
27a. **let** n″ = remove_L(n)(uid_L(ℓ)) **in**
27b. **let** h_α,h_β:H • {h_α,h_β}∩obs_Hs(n)={} **in**
27b. **let** n‴ = insert_H(n″)(h_α) **in**
27b. **let** n⁗ = insert_H(n‴)(h_β) **in**
27c. **let** $l_{j\alpha}$,$l_{k\beta}$:L • {$l_{j\alpha}$,$l_{k\beta}$}∩obs_Ls(n)={} **in**
27(c)i. **let** n⁗⁗ = insert_L(n⁗)($l_{j\alpha}$) **in**
27(c)ii. n′ = insert_L(n⁗⁗)($l_{k\beta}$) **end end end end end end**

We refer to Appendix Sects. B.3.2–B.3.4 There you will find definitions of insert_link, remove_hub and remove_link action functions. ■

176

Modelling Events

- The domain describer has decided that an entity is a perdurant and is, or represents an event: occurred surreptitiously, that is, was not an action that was *"done by an agent and intentionally under some description"* [35].
  - The domain describer has further decided that the observed event is of a class of events — of the "same kind" — that need be described.
  - By events of the 'same kind' is meant that these can be described by the same predicate function signature and predicate function definition.

177

- First the domain describer must decide on the underlying predicate function signature.
  - The argument type and the result type of the signature are those of either previously identified
    - parts,
    - unique part identifiers, or
    - attributes.
- Sooner or later the domain describer must decide on the predicate function definition.
  - For predicate function definitions it appears to be convenient to have developed, "on the side", a theory of mereology for the part types involved in the function signature.

178
179

8.4 Discrete Behaviours 180

We shall distinguish between discrete behaviours (this section) and continuous behaviours (Sect. 9). Roughly discrete behaviours proceed in discrete (time) steps — where, in this tutorial, we omit considerations of time. Each step corresponds to an action or an event or a time interval between these. Actions and events may take some (usually inconsiderable time), but the domain analyser has decided that it is not of interest to understand what goes on in the domain during that time
181 (interval). Hence the behaviour is considered discrete.

Continuous behaviours are continuous in the sense of the calculus of mathematical; to qualify as a continuous behaviour time must be an essential aspect of the behaviour. We shall treat continuous behaviours in Sect. 9.

Discrete behaviours can be modelled in many ways, for example using CSP [53]. MSC [57], Petri Nets [81] and Statechart [50]. We refer to Chaps. 12–14 of [8]. In this tutorial we shall use RSL/CSP.

8.4.1 What is Meant by 'Behaviour'? 182

We give two characterisations of the concept of 'behaviour'. a "loose" one and a "slanted one.

A loose characterisation runs as follows: by a behaviour we understand a set of sequences of
183 actions, events and behaviours.

A "slanted" characterisation runs as follows: by a behaviour we shall understand either a sequential behaviour consisting of a possibly infinite sequence of zero or more actions and events; or one or more communicating behaviours whose output actions of one behaviour may synchronise and communicate with input actions of another behaviour; and or two or more behaviours acting
184 either as internal non-deterministic behaviours ($\lceil\rceil$) or as external non-deterministic behaviours ($[]$).

This latter characterisation of behaviours is "slanted" in favour of a CSP, i.e., a communicating sequential behaviour, view of behaviours. We could similarly choose to "slant" a behaviour characterisation in favour of Petri Nets, or MSCs, or Statecharts, or other.

8.4.2 Behaviour Narratives 185

Behaviour narratives may take many forms. A behaviour may best be seen as composed from several interacting behaviours. Instead of narrating each of these, as will be done in Example **??**, one may proceed by first narrating the interactions of these behaviours. Or a behaviour may best be seen otherwise, for which, therefore, another style of narration may be called for, one that "traverses the landscape" differently. Narration is an art. Studying narrations – and practice – is a good way to learn effective narration.

8.4.3 An Aside on Agents, Behaviours and Processes 186

"In philosophy and sociology, agency is the capacity of an agent (a person or other entity) to act in a world. In philosophy, the agency is considered as belonging to that agent even if that agent represents a fictitious character, or some other non-existent entity." That is, we consider agents to be those persons or other entities that are in the domain and observes the domain evaluates what
187 is being observed and invokes actions. We describe agents by describing behaviours. A behaviour description denotes a process, that is, a set of actions, events and processes. We shall not enter into any further speculations on agency, agents and how agents observe, including what they know and believe (epistemic logic), what is necessary and possible (deontic logic) and what is true at some tie and what is always true (temporal logic). A proper domain science and engineering must, however, eventually examine these (modal logic) issues.

8.4.4 On Behaviour Description Components 188

When narrating plus, at the same time, formalising, i.e., textually alternating between narrative texts and formal texts, one usually starts with what seems to be the most important behaviour

concepts of the given domain: which are the important part types characterising the domain; which of these parts will become a basis for behaviour processes; how are these behaviour processes to interact, that is, which channels and what messages may possibly be communicated. 189

Example: 37 A Road Traffic System. We continue our long line of examples around transport nets. The present example interprets these as road nets.

[1] Continuous Traffic For the road traffic system perhaps the most significant example of a behaviour is that of its traffic

28. the continuous time varying discrete positions of vehicles, vp:VP[44],
29. where time is taken as a dense set of points.

type
29. cT
28. cRTF = cT → (V $\overrightarrow{m}$ VP)

190

[2] Discrete Traffic We shall model, not continuous time varying traffic, but

30. discrete time varying discrete positions of vehicles,
31. where time can be considered a set of linearly ordered points.

31. dT
30. dRTF = dT $\overrightarrow{m}$ (V $\overrightarrow{m}$ VP)

32. The road traffic that we shall model is, however, of vehicles referred to by their unique identifiers.

type
32. RTF = dT $\overrightarrow{m}$ (VI $\overrightarrow{m}$ VP)

191

[3] Time: An Aside We shall take a rather simplistic view of time [27, 68, 80, 93].

33. We consider dT, or just T, to stand for a totally ordered set of time points.
34. And we consider TI to stand for time intervals based on T.
35. We postulate an infinitesimal small time interval δ.
36. T, in our presentation, has lower and upper bounds.
37. We can compare times and we can compare time intervals.
38. And there are a number of "arithmetics-like" operations on times and time intervals.

192

type
33. T
34. TI
value
35. δ:TI
36. MIN, MAX: T → T
36. <,≤,=,≥,>: (T×T)|(TI×TI) → **Bool**

[44] For VP see Item 47a on page 43.

37. $-$: $\mathbb{T}\times\mathbb{T} \to \mathbb{TI}$
38. $+$: $\mathbb{T}\times\mathbb{TI},\mathbb{TI}\times\mathbb{T} \to \mathbb{T}$
38. $-,+$: $\mathbb{TI}\times\mathbb{TI} \to \mathbb{TI}$
38. $*$: $\mathbb{TI}\times\mathbf{Real} \to \mathbb{TI}$
38. $/$: $\mathbb{TI}\times\mathbb{TI} \to \mathbf{Real}$

193

39. We postulate a global clock behaviour which offers the current time.

40. We declare a channel clk_ch.

```
value
39. clock: 𝕋 → out clk_ch Unit
39. clock(t) ≡ ... clk_ch!t ... clock(t ⌈⌉ t+δ)
channnel
40. clk_ch:𝕋
```

194

[4] Road Traffic System Behaviours

41. Thus we shall consider our road traffic system, rts, as
    a. the concurrent behaviour of a number of vehicles and, to "observe", or, as we shall call it, to monitor their movements,
    b. the monitor behaviour.

```
value
41. trs() =
41a.     ‖ {veh(uid_V(v))(v)|v:V•v ∈ vs}
41b.     ‖ mon(m)([])
```

where the "extra" monitor argument ([]) records the discrete road traffic, RTF, initially set to the
195 empty map (of, "so far no road traffic"!).

[5] Globally Observable Parts

There is given

42. a net, n:N,
43. a set of vehicles, vs:V-**set**, and
44. a monitor, m:M.

The n:N, vs:V-**set** and m:M are observable from the road traffic system domain.

```
value
42. n:N = obs_N(Δ)
42. ls:L-set = obs_Ls(obs_LS(n)), hs:H-set = obs_Hs(obs_HS(n)),
42. lis:LI-set = {uid_L(l)|l:L•l ∈ ls}, his:HI-set = {uid_H(h)|h:H•h ∈ hs}
43. vs:V-set = obs_Vs(obs_VS(obs_F(Δ))), vis:V-set = {uid_V(v)|v:V•v ∈ vs}
44. m:obs_M(Δ)
```

196

[6] Channels In order for the monitor behaviour to assess the vehicle positions these vehicles communicate their positions to the monitor via a vehicle to monitor channel. In order for the monitor to time-stamp these positions it must be able to "read" a clock.

45. Thus we declare a set of channels indexed by the unique identifiers of vehicles and communicating vehicle positions; and
46. a single clock to monitor channel.

channel
45. {vm_ch[vi]|vi:VI•vi ∈ vis}:VP
46. clkm_ch:dT

197

[7] An Aside: Attributes of Vehicles

47. Dynamic attributes of vehicles include
    a position
        i. at a hub (about to enter the hub — referred to by the link it is coming from, the hub it is at and the link it is going to, all referred to by their unique identifiers or
        ii. some fraction "down" a link (moving in the direction from a from hub to a to hub — referred to by their unique identifiers)
        iii. where we model fraction as a real between 0 and 1 included.
    b velocity, acceleration, etcetera.

type
47a. VP = atH | onL
47(a)i. atH :: fli:LI × hi:HI × tli:LI
47(a)ii. onL :: fhi:HI × li:LI × frac:FRAC × thi:HI
47(a)iii. FRAC = **Real**, **axiom** ∀ frac:FRAC • 0 ≤ frac ≤ 1
47b. Vel, Acc, ...

198

[8] Behaviour Signatures

48. The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**); and "behaves", that is, continues forever (hence the last **Unit**).
49. The vehicle behaviours are indexed by the unique identifier, uid_V(v):VI, the vehicle part, v:V and the vehicle position; offers communication to the monitor behaviour (on channel vm_ch[vi]); and behaves "forever".
50. The monitor behaviour takes the so far unexplained monitor part, m:M, as one argument and the discrete road traffic, drtf:dRTF, being repeatedly "updated" as the result of **in**put communications from (all) vehicles; the behaviour otherwise runs forever.

value
48. rts: **Unit** → **Unit**
49. veh: vi:VI → v:V → VP → **out** vm_ch[vi] **Unit**
50. mon: m:M → RTF → **in** {vm_ch[vi]|vi:VI•vi ∈ vis},clkm_ch **Unit**

199

[9] The Vehicle Behaviour

51. A vehicle process is indexed by the unique vehicle identifier vi:VI, the vehicle "as such", v:V and the vehicle position, vp:VP.

    The vehicle process communicates with the monitor process on channel vm[vi] (sends, but receives no messages), and otherwise evolves "infinitely" (hence **Unit**).

52. We describe here an abstraction of the vehicle behaviour at a Hub (hi).
    a Either the vehicle remains at that hub informing the monitor,
    b or, internally non-deterministically,
        i. moves onto a link, tli, whose "next" hub, identified by thi, is obtained from the mereology of the link identified by tli;
        ii. informs the monitor, on channel vm[vi], that it is now on the link identified by tli,
        iii. whereupon the vehicle resumes the vehicle behaviour positioned at the very beginning (0) of that link,
    c or, again internally non-deterministically,
    d the vehicle "disappears — off the radar" !

```
52.   veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
52a.        vm_ch[ vi ]!vp ; veh(vi)(v)(vp)
52b.      ⌈⌉
52(b)i.     let {hi′,thi}=mereo_L(get_L(tli)(n)) in assert: hi′=hi
52(b)ii.    vm_ch[ vi ]!onL(tli,hi,0,thi) ;
52(b)iii.   veh(vi)(v)(onL(tli,hi,0,thi)) end
52c.      ⌈⌉
52d.        stop
```

53. We describe here an abstraction of the vehicle behaviour on a Link (ii).
    Either
    a the vehicle remains at that link position informing the monitor,
    b or, internally non-deterministically,
    c if the vehicle's position on the link has not yet reached the hub,
        i. then the vehicle moves an arbitrary increment δ along the link informing the monitor of this, or
        ii. else, while obtaining a "next link" from the mereology of the hub (where that next link could very well be the same as the link the vehicle is about to leave),
            A. the vehicle informs the monitor that it is now at the hub identified by thi,
            B. whereupon the vehicle resumes the vehicle behaviour positioned at that hub.
54. or, internally non-deterministically,
55. the vehicle "disappears — off the radar" !

```
51.   veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
53a.        vm_ch[ vi ]!vp ; veh(vi)(v)(vp)
53b.      ⌈⌉
53c.        if f + δ<1
53(c)i.         then vm_ch[ vi ]!onL(fhi,li,f+δ,thi) ;
```

```
53(c)i.                veh(vi)(v)(onL(fhi,li,f+δ,thi))
53(c)ii.          else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
53(c)iiA.              vm_ch[ vi ]!atH(li,thi,li′);
53(c)iiB.              veh(vi)(v)(atH(li,thi,li′)) end end
54.       ⌈⌉
55.         stop
```

204

[10] The Monitor Behaviour

56. The monitor behaviour evolves around the attributes of an own "state", m:M, a table of traces of vehicle positions, while accepting messages about vehicle positions and otherwise progressing "in[de]finitely".
57. Either the monitor "does own work"
58. or, internally non-deterministically accepts messages from vehicles.
    a A vehicle position message, vp, may arrive from the vehicle identified by vi.
    b That message is appended to that vehicle's movement trace,
    c whereupon the monitor resumes its behaviour —
    d where the communicating vehicles range over all identified vehicles.

205

```
56.  mon(m)(rtf) ≡
57.      mon(own_mon_work(m))(rtf)
58.    ⌈⌉
58a.    [] { let ((vi,vp),t) = (vm_ch[ vi ]?,clkm_ch?), in
58b.         let rtf′ = rtf † [ t ↦ rtf(max dom rtf) † [ vi ↦ vp ] ] in
58c.         mon(m)(rtf′) end
58d.         end | vi:VI • vi ∈ vis }

57.  own_mon_work: M → TBL → M
```

We do not describe the clock behaviour by other than stating that it continually offers the current time on channel clkm_ch. ■

8.4.5 A Model of Parts and Behaviours

206

How often have you not "confused" the perdurant notion of a train process: progressing from railway station to railway station, with the endurant notion of the train, say as it appears listed in a train time table, or as it is being serviced in workshops, etc. There is a reason for that — as we shall now see: parts may be considered syntactic quantities denoting semantic quantities. We therefore describe a general model of parts of domains and we show that for each instance of such a model we can 'compile' that instance into a CSP'program'.

207

A Model of Parts

59. The *whole* contains a set of *parts*.
60. *Parts* are either *atomic* or *composite*.
61. From *composite parts* one can observe a set of *parts*.
62. All *parts* have *unique identifiers*

```
type
59. W, P, A, C
60. P = A | C
value
61. obs_Ps: (W|C) → P-set
```

```
type
62. PI
value
62. uid_Π: P → Π
```

208

63. From a *whole* and from any *part* of that *whole* we can e**xtr**act all contained *part*s.
64. Similarly one can e**xtr**act the *unique identifier*s of all those contained *parts*.

```
value
63. xtr_Ps: (W|P) → P-set
63. xtr_Ps(w) ≡
63.     {xtr_Ps(p)|p:P•p ∈ obs_Ps(p)}
63.     pre: is_W(p)
63. xtr_Ps(p) ≡
63.     {xtr_Ps(p)|p:C•p∈ obs_Ps(p)}∪{p}
63.     pre: is_P(p)
64. xtr_Πs: (W|P) → Π-set
```

65. Each part may have a *mereology* which may be "empty".
66. A *mereology*'s *unique part identifier*s must refer to some other parts other than the part itself.

```
64. xtr_Πs(wop) ≡
64.     {uid_P(p)|p ∈ xtr_Ps(wop)}
65. mereo_P: P → Π-set
axiom
66. ∀ w:W
66.     let ps = xtr_Ps(w) in
66.     ∀ p:P • p ∈ ps •
66.         ∀ π:Π • π ∈ mereo_P(p) ⇒
66.             π ∈ xtr_Πs(p) end
```

209

67. An attribute map of a *part* associates with *attribute names*, i.e., *type names*, their *value*s, whatever they are.
68. From a *part* one can extract its attribute map.
69. Two *parts share attributes* if their respective attribute maps share *attribute names*.

```
type
67.  AttrNm, AttrVAL,
67.  AttrMap = AttrNm ~m~ AttrVAL
value
68.  attr_AttrMap: P → AttrMap
69.  share_Attributes: P×P → Bool
69.  share_Attributes(p,p′) ≡
```

70. Two *parts share properties* if the y
    a. either *share attributes*
    b. or the *unique identifier* of one is in the *mereology* of the other.

```
69.     dom attr_AttrMap(p) ∩
69.     dom attr_AttrMap(p′) ≠ {}
70.  share_Properties: P×P → Bool
70.  share_Properties(p,p′) ≡
70a.     share_Attributes(p,p′)
70b.  ∨ uid_P(p) ∈ mereo_P(p′)
70b.  ∨ uid_P(p′) ∈ mereo_P(p)
```

210

Conversion of Parts into CSP Programs

71. We can define the set of two element sets of *unique identifiers* where
    - one of these is a *unique part identifier* and
    - the other is in the mereology of some other *part*.
    - We shall call such two element "pairs" of *unique identifiers* connectors.
    - That is, a connector is a two element set, i.e., "pairs", of *unique identifiers* ≪ for which the identified parts share properties.
72. Let there be given a 'whole', w:W.
73. To every such "pair" of *unique identifiers* we associate a *channel*
    - or rather a position in a matrix of *channels* indexed over the "pair sets" of *unique identifiers*.
    - and communicating messages m:M.

```
type
71.  K = Π-set axiom ∀ k:K•card k=2
value
71.  xtr_Ks: (W|P) → K-set
71.  xtr_Ks(wop) ≡
71.     let ps = xtr_Ps(w) in
71.     {{uid_P(p),π}|p:P,π:Π•p∈ ps
71.         ∧ ∃ p′:P•p′≠p∧π=uid_P(p′)
71.         ∧ uid_P(p)∈uid_P(p′)} end
72.  w:W
73.  channel {ch[k]|k:xtr_Ks(w)}:M
```

211

74. Now the 'whole' *behaviour* whole is the parallel composition of *part processes*, one for each of the immediate parts of the *whole*.
75. A *part process* is
    a either an *atomic part process*, atom, if the *part* is an *atomic part*,
    b or it is a *composite part process*, comp, if the *part* is a *composite part*.

```
74. whole: W → Unit
74. whole(w) ≡
74.     ‖ {part(uid_P(p))(p) |
74.         p:P•p ∈ xtr_Ps(w)}
```

```
75. part: π:Π → P → Unit
75. part(π)(p) ≡
75b.     is_A(p) → atom(π)(p),
75b.     _   → comp(π)(p)
```

212

76. A *composite process*, part, consists of
    a a *composite core process*, comp_core, and
    b the parallel composition of *part processes* one for each *contained part* of part.
77. An *atomic process* consists of just an *atomic core process*, *atom_core*.

```
value
76. comp: π:Π → p:P →
76.     in,out {ch[ {π,π′}|{π′∈ mereo_P(p)} ]}
76.     Unit
76. comp(π)(p) ≡
76a.     comp_core(π)(p) ‖
76b.     ‖ {part(uid_P(p′))(p′) |
76b.         p′:P•p′ ∈ obs_Ps(p)}
```

```
77. atom: π:Π → p:P →
77.     in,out {ch[ {π,π′}|{π′∈ mereo_P(p)} ]}
77.     Unit
77. atom(π)(p) ≡ atom_core(π)(p)
```

213

78. The core behaviours both
    a update the part properties and
    b recurses with the updated properties,
    c without changing the part identification.

    We leave the update action undefined.

```
value
78. core: π:Π → p:P →
78.     in,out {ch[ {π,π′}|{π′∈ mereo_P(p)} ]}
78.     Unit
```

```
78. core(π)(p) ≡
78a.     let p′ = update(π)(p)
78b.     in core(π)(p′) end
78b.     assert: uid_P(p)=π=uid_P(p′)
```

214

The model of parts can be said to be a syntactic model. No meaning was "attached" to parts. The conversion of parts into CSP programs can be said to be a semantic model of parts, one which to every part associates a behaviour which evolves "around" a state which is that of the properties of the part.

8.4.6 Sharing Properties ≡ Mutual Mereologies

215

In the model of the tight relationship between parts and behaviours, Sect. 8.4.5, we "equated" two-element set of unique identifiers of parts that share properties with the concept of connectors, and these again with channels. We need secure that this relationship, between the two-element connector sets of unique identifiers of parts that share properties and the channels with the following theorem:

216

79. For every *whole*, i.e., domain,
80. if two distinct *parts* share properties
81. then their respective mereologies refer to one another,
82. and vice-versa if two distinct *parts* have their respective mereologies refer to one another, then they share properties.

theorem:

```
79. ∀ w:W,p,p′:P•p≠p′∧{p,p′}⊆xtr_Ps(w) ⇒
80.     share_Properties(p,p′)
```

82. ≡
81. uid_P(p)∈mereo_P(p′)∧uid_P(p′)∈mereo_P(p)

8.4.7 Behaviour Signatures 217

By a behaviour signature we shall understand the combination of three clauses: a message type clause,

- **type** M,

possibly a channel index type clause,

- **type** Idx,

a channel declaration clause

- **channel** ch:M or
  channel {ch[i]|i:Idx•i ∈is}:M

where is is a set of Idx values (defined somehow, e.g., **value** is:Idx-**set** = ... where ... is an expression of Idx values), and, finally, a behaviour function signature:

- **value** beh: Π → P → **out** ch **Unit** or
  value beh: Π → P → **out** ch **Unit** or
  value beh: Π → P → **in, out** ch **Unit** or
  value beh: Π → P → **in, out** {ch[i]|i:Idx• ∈is′} **Unit** or
  value beh: Π → P → **in** {ch[i]|i:Idx• ∈is′} **out** {ch[j]|j:Idx• ∈is′} **Unit**, etc.

The **Unit** ground term type expression denotes a meta-state: If argument type is **Unit**, then function invocation is designated by f(). If result type is **Unit** then the designated function may potentially not terminate. ARG indicates that the behaviour is either initialised with an initial argument, or "recurses" with a result in the form of an update argument.

Otherwise behaviour descriptions make full use of type definitions, unique part identifiers, 218 part mereologies, part attributes, actions, and events. The Conversion of Parts into CSP Programs "story" gives the general idea: To associate, in principle, with every part an own behaviour. (Example ?? (Pages ??–??) did not do that: in principle it did, but then it omitted describing behaviours of "un-interesting" parts!) Tentatively each behaviour signature, that is, each part behaviour, is specified having a unique identifier type, respectively given a unique identifier argument. Whether this tentative provision for unique identifiers is necessary will soon be revealed by further domain 219 analysis. Before defining the behaviour process signatures the domain analyser examines each of the chosen behaviours with respect to its interaction with other chosen behaviours in order to decide on interaction message types and "dimensionality" of channels, whether singular or an array. Then the message types can be *defined*, the channels *declared*, and the behaviour function signature can be *defined*, i.e., the full behaviour signature can be *defined*.

8.4.8 Behaviour Definitions 220

We observe from Sect. 8.4.5.4, the 'Conversion of Parts into CSP Programs', Page 46, that the "generation" of the core processes was syntax directed, yet "delivered" a "flat" structure of parallel processes, that is, no processes "running", *embedded*, within other processes. We make this remark since parts did not follow that prescription: parts can, indeed, be *embedded* within one 221 another. So our first "conclusion"[45], with respect to the structure of domain behaviours, is that we shall model all behaviours of the "whole" domain as a flat structure of concurrent behaviours — one for each part contained in the whole — which, when they need refer to properties of behaviours

of parts within which the part on which "their" behaviour is embedded then they interact with the behaviours of those parts, that is, communicate messages.

Section 8.4.5.4, the 'Conversion of Parts into CSP Programs', Page 46, then suggested that there be one atom core behaviour for each atomic part, and one composite core behaviour for each composite part of the domain. The domain analyser may find that some of these core behaviours are not necessary, that is, that they — for the chosen scope of the domain model — do not play a meaningful rôle.

Example: 38 "Redundant" Core Behaviours. We refer to the series of examples around the transport net domain. Transport nets, n:N, consist of sets, hs:HS, of hubs and sets, ls:LS, of links. Yet we may decide, for one domain scope, to model only hub, link and vehicle behaviours, and not 'set of hubs' and 'set of links' behaviours. ■

Then the domain analyser can focus on exploring each individual process behaviour. Again the Conversion of Parts into CSP Programs "story" gives the general ideas that motivate the following: For each of the parts, p, a behaviour expression can be "generated": beh_p(uid_P(p))(p). The idea is that (uid_P(p)) uniquely identifies the part behaviour and that the part properties of (p) serve as the local state for beh_p.

Now we present an analysis of part behaviours around three 'alternatives': (i) a part behaviour which basically represents a proactive behaviour; (ii) one which basically represents a reactive behaviour; and (iii) one which, so-to-speak alternates between proactive and reactive behaviours.

What we are doing now is to examine the form of the core behaviours, cf. Item 78 (Page 47).

• • •

(i) A proactive behaviour is characterised by three facets. (i.1) taking the initiative to interact with other part behaviours by offering output, (i.2) internally non-deterministically ($\sqcap$) ranging interactions over several alternatives, and (i.3) externally non-deterministically ($\Box$) selecting which other behaviour to interact with, i.e., to offer output to.

(i.1) A proactive behaviour takes the initiative to interact by expressing output clauses:

83. $\mathcal{O}_P$: ch ! val or ch[i] ! val or ch[i,j] ! val etc.

(i.2) The proactive behaviour interaction request may range over either of a finite number of alternatives, one for each alternative, a_i, "kind" of interaction. We may express such a non-deterministic (alternative) choice *either* as follows:

84. $\mathcal{NI}_P$: **type** Choice = $a_1 \sqcap a_2 \sqcap ... \sqcap a_n$
value let c:Choice **in case** c **of** $a_1 \rightarrow \mathcal{E}_1$, ..., $a_n \rightarrow \mathcal{E}_n$ **end end**

or, which is basically the same,

85. $\mathcal{NI}_P$: **value** ... $\mathcal{E}_1 \sqcap ... \sqcap \mathcal{E}_n$ ...

where each $\mathcal{E}_i$ usually contains an input clause, for example, ch ?.

(i.3) The proactive external non-deterministic choice is directed at either of a number of other part behaviours. This proactive selection is expressed

86. $\mathcal{NX}_P$: $\mathcal{C}_i \Box \mathcal{C}_j \Box ... \Box \mathcal{C}_k$

where each of the $\mathcal{C}$lauses express respective output clauses (usually) directed at different part behaviours, say ch[i] ! val. ch[j] ! val, etc., ch[k] ! val. Another way of expressing external non-deterministic choice selection is

87. $\mathcal{NX}_P$: □ { ...; ch[i] ! fct(i) ; ... | i:Idx•i ∈ is }

$\mathcal{O}$utput clauses [(i.1)], Item 84 $\mathcal{O}_P$, may [(i.2)] occur in the $\mathcal{E}_i$ clauses of $\mathcal{NI}_P$, Items 85 and 86 and must [(i.3)] occur in each of the $\mathcal{C}_i$ clauses of $\mathcal{NX}_P$, Item 87.

[45] We put double quotes around the term 'conclusion' (above) since that conclusion was and is a choice, that is, not governed by necessity.

• • •

(ii) A reactive behaviour is characterised by three (ii.1) offering to interact with other part behaviours by offering to accept input, (ii.2) internally non-deterministically (⌈⌉) ranging interactions over several alternatives, and (ii.3) externally non-deterministically ([]) selecting which other behaviour to interact with, i.e., to accept input from.

(ii.1) A reactive behaviour expresses input clauses:

88. $\mathcal{I}_R$: ch ? or ch[i] ? or ch[i,j] ? etc.

(ii.2) The reactive behaviour may range over either of a finite number of alternatives, one for each alternative, a_i, "kind" of interaction. We may express such a non-deterministic (alternative) choice *either* as follows:

89. $\mathcal{NI}_R$: **value let** c:Choice **in case** c **of** $a_1 \rightarrow \mathcal{E}_1$, ..., $a_n \rightarrow \mathcal{E}_n$ **end end**

where each of the expressions, $\mathcal{E}_i$, may, and usually contains a input clause ($\mathcal{I}$, Item 88). Thus the $\mathcal{NI}_R$ clause is almost identical to the $\mathcal{NI}_P$ clause, Item 85 on the preceding page. Hence another way of expressing external non-deterministic choice is

90. $\mathcal{NX}_R$: ⌈⌉ { ...; ch[i] ! fct(i) ; ... | i:Idx•i ∈ is }.

(ii.3) The reactive behaviour selection is directed at either of a number of other part behaviours. This external non-deterministic choice is expressed

91. $\mathcal{NX}_R$: $\mathcal{C}_i$ [] $\mathcal{C}_j$ [] ... [] $\mathcal{C}_k$

where each of the $\mathcal{C}$lauses express respective input clauses (usually) directed at different part behaviours, say ch[i] ?. ch[j] ?, etc., ch[k] ?. Another way of expressing external non-deterministic choice selection is

92. $\mathcal{NX}_R$: [] { ...; ch[i] ? ; ... | i:Idx•i ∈ is }

Thus the $\mathcal{NX}_R$ clauses are almost identical to the $\mathcal{NX}_P$ clauses, Items 86–87.

$\mathcal{I}$nput clauses [(ii.1)], Item 88 $\mathcal{I}_R$, may [(ii.2)] occur in the $\mathcal{E}_i$ clauses of $\mathcal{NI}_R$, Items 89–90 and must [(ii.3)] occur in each of the $\mathcal{C}_i$ clauses of $\mathcal{NX}_R$, Items 91–92.

• • •

(iii) An alternating proactive behaviour and reactive behaviour is characterised by expressing both reactive behaviour and proactive behaviours combined by either non-deterministic internal choice (⌈⌉) or non-deterministic external choice ([]) combinators. For example:

93. $(\mathcal{NI}_{P_i}[⌈⌉\text{or}[]]\mathcal{NX}_{P_j})[⌈⌉\text{or}[]](\mathcal{NI}_{R_k}[⌈⌉\text{or}[]]\mathcal{NX}_{R_\ell})$.

The meta-clause [⌈⌉or[]] stands for either ⌈⌉ or []. Here there usually is a disciplined use of input/output clauses.

Example: 39 A Pipeline System Behaviour. We refer to Examples 16 (Page 25) and 23–25 (Pages 29–31) and especially Examples 26–27 (Pages 31–32). We consider (cf. Example 24) the pipeline system units to represent also the following behaviours: pls:PLS, Item 4a on page 29, to also represent the system process, pipeline_system, and for each kind of unit, cf. Example 16, there are the unit processes: unit, well (Item 3c on page 25), pipe (Item 3a), pump (Item 3a), valve (Item 3a), fork (Item 3b), join (Item 3b) and sink (Item 3d on page 25).

channel
{ pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } MUPLS
{ u_u_ch[ui,uj]:ui,uj:UI•{ui,uj}⊆UIs(pls) } MUU
type

```
    MUPLS, MUU
value
    pipeline_system: PLS → in,out { pls_u_ch[ ui ]:ui:UI•i ∈ UIs(pls) }  Unit
    pipeline_system(pls) ≡ ‖ { unit(u)|u:U•u ∈ obs_Us(pls) }
    unit: U → Unit
    unit(u) ≡
3c.     is_We(u) → well(uid_U(u))(u),
3a.     is_Pu(u) → pump(uid_U(u))(u),
3a.     is_Pi(u) → pipe(uid_U(u))(u),
3a.     is_Va(u) → valve(uid_U(u))(u),
3b.     is_Fo(u) → fork(uid_U(u))(u),
3b.     is_Jo(u) → join(uid_U(u))(u),
3d.     is_Si(u) → sink(uid_U(u))(u)
```

We illustrate essentials of just one of these behaviours.

```
3b. fork: ui:UI → u:U → out,in pls_u_ch[ ui ],
                        in  { u_u_ch[ iui,ui ] | iui:UI • iui ∈ sel_UIs_in(u) }
                        out { u_u_ch[ ui,oui ] | iui:UI • oui ∈ sel_UIs_out(u) }  Unit
3b. fork(ui)(u) ≡
3b.     let u′ = core_fork_behaviour(ui)(u) in
3b.     fork(ui)(u′) end
```

The core_fork_behaviour(ui)(u) distributes what oil (or gas) in receives, on the one input sel_UIs_in(u) = {iui}, along channel u_u_ch[iui] to its two outlets sel_UIs_out(u) = {oui_1,oui_2}, along channels u_u_ch[oui_1], u_u_ch[oui_2].

The core_fork_behaviour(ui)(u) also communicates with the pipeline_system behaviour. What we have in mind here is to model a traditional supervisory control and data acquisition, SCADA system. SCADA is then part of the pipeline_system behaviour.

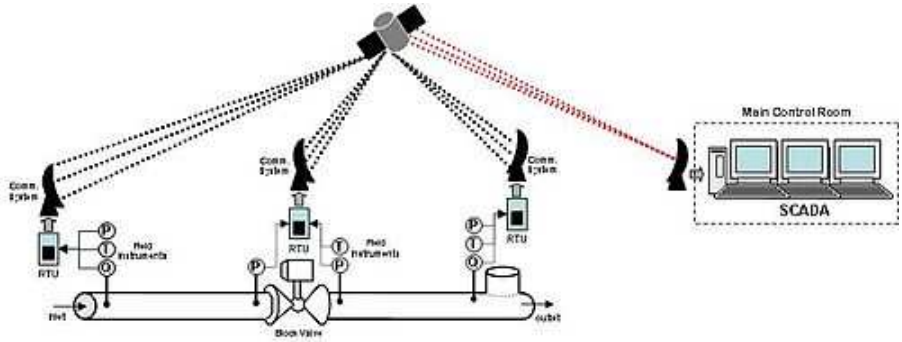


Figure 1: A supervisory control and data acquisition system

94.

```
94. pipeline_system: PLS → in,out { pls_u_ch[ ui ]:ui:UI•i ∈ UIs(pls) }  Unit
94.   pipeline_system(pls) ≡ scada(props(pls)) ‖ ‖{ unit(u)|u:U•u ∈ obs_Us(pls) }
```

props was defined in Sect. 7 Page 34.

95. scada non-deterministically (internal choice, ⌈⌉), alternates between continually

    a doing own work,

b acquiring data from pipeline units, and

c controlling selected such units.

```
type
95.  Props
value
95.  scada: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis }  Unit
95.  scada(props) ≡
95a.     scada(scada_own_work(props))
95b.   ⌈⌉ scada(scada_data_acqui_work(props))
95c.   ⌈⌉ scada(scada_control_work(props))
```

241

We leave it to the readers imagination to describe scada_own_work.

96. The scada_data_acqui_work

    a non-deterministically, external choice, [], offers to accept data,

    b and scada_input_updates the scada state —

    c from any of the pipeline units.

```
value
96.  scada_data_acqui_work: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis }  Props
96.  scada_data_acqui_work(props) ≡
96a.     [] { let (ui,data) = pls_ui_ch[ui] ? in
96b.          scada_input_update(ui,data)(props) end
96c.          | ui:UI • ui ∈ uis }

96b.  scada_input_update: UI × Data → Props → Props
type
96a.  Data
```

242

97. The scada_control_work

    a analyses the scada state (props) thereby selecting a pipeline unit, ui, and the controls, ctrl, that it should be subjected to;

    b informs the units of this control, and

    c scada_output_updates the scada state.

```
97.  scada_control_work: Props → in,out { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis }  Props
97.  scada_control_work(props) ≡
97a.     let (ui,ctrl) = analyse_scada(ui,props) in
97b.     pls_ui_ch[ui] ! ctrl ;
97c.     scada_output_update(ui,ctrl)(props) end

97c. scada_output_update UI × Ctrl → Props → Props
type
97a. Ctrl
```

We leave it to the reader to suggest definitions of the core SCADA functions: scada_own_work, analyse_scada and scada_internal_update These functions depend on the system being monitored & controlled. Typically they are formulated in the realm of automatic control theory. We hint at Sect. 9.4 (Pages 55–57). ■

243

Modelling Behaviours

- The domain describer has decided that an entity is a perdurant and is, or represents a behaviour.
  - The domain describer has further decided that the observed behaviour is of a class of behaviours — of the "same kind" — that need be described.
  - By behaviours of the 'same kind' is meant that these can be described by the same channel declarations, function signature and function definition.
- First the domain describer must decide on the underlying function signature. 244
  - It must be decided which synchronisation and communication
    - inputs and
    - outputs

    this behaviour requires, i.e., the **in,out** clause of the signature,
  - that also includes the "discovery" of necessary channel declarations.
- Finally the function definition must be decided upon.

9 Continuous Perdurants

245

Warning !

This section constitutes a preliminary rough sketch. Its presentation may be skipped, altogether, at the 28 August, 2012, FM 2012 International Symposium in Paris, France.

By a continuous perdurant we shall understand a continuous behaviour.

This section serves two purposes: to point out that believable system descriptions must entail both a discrete phenomena domain description and a continuous phenomena mathematical model. and this poses some semantics problems: the formal semantics of the discrete phenomena description language and the meta-mathematics of, for example, differential equations, at least as of today, August 10, 2012, are not commensurable! That is, we have a problem — as will be outlined in Sect. 9.6.

9.1 Some Examples

246

Example: 40 Continuous Behaviour: The Weather. We give a familiar example of continuous behaviour. The *weather* — understood as the time-wise evolution of a number of attributes of the *weather* material: *temperature*, *wind direction*, *wind force*, *atmospheric pressure*, *humidity*, *sky formation* (`clear, cloudy, ...`), *precipitation*, etcetera. That is, *weather* is seen as the state of the *atmosphere* as it evolves over time. ■

247

Example: 41 Continuous Behaviour: Road Traffic. We give another familiar example of continuous behaviour. The *automobile traffic* is the time-wise evolution of cars along a net has the

following additional attributes: *car identity* (CI), *position* (P, on the net), *direction* (D), *velocity* (V), *acceleration* (A), etcetera (...). The equation below captures this:

$$\mathsf{TF} = \mathsf{T} \rightarrow (\mathsf{CI} \overrightarrow{m} (\mathsf{P} \times \mathsf{D} \times \mathsf{V} \times \mathsf{A} \times ...))$$

We refer to Example **??** specifically the veh, hub and mon behaviours. These "mimic" a discretised version of the above:

$$\mathsf{TF} = \mathsf{T} \overrightarrow{m} (\mathsf{CI} \overrightarrow{m} (\mathsf{P} \times \mathsf{D} \times \mathsf{V} \times \mathsf{A} \times ...))$$

248

Example: 42 Pipeline Flows. A last example of continuous behaviour. We refer to Examples 14, 16, 23–27, 42–46 and 50. These examples focused on the atomicparts and the composite parts of pipelines, and dealt with the liquid or gas materials as they related to pipeline units. In the present example we shall focus on the overall material flow "across" a pipeline. in particular the
249 continuity as as contrasted with the pipeline unit discrete aspects of flow. Which, then, are these pipeline system continuity concerns ? In general we are interested in

1. *whether the flow is laminar or turbulent:*
   a *within a unit, or*
   b *within an entire, possibly intricately networked pipeline*;
2. *what the shear stresses are*;
3. *whether there are undesirable pressures*;
4. *whether there are leaks above normal values*;

etcetera. To answer questions like those posed in Items 1a and 2, we need not build up the models sketched in Examples 14, 16, 26, 27, 42–46 and 50. But for questions like those posed in Items 1b,
250 3 and 4 we need such models. To answer any of the above questions, and many others, we need establish, in the case of pipelines, *fluid dynamics models* [4, 91, 97, 39]. These models involve such mathematical as are based, for example, on *Newtonian Fluid Behaviours*, *Bernoulli Equations*, *Navier–Stokes Equations*, etcetera. Each of these mathematical models capture the dynamics of one specific pipeline unit, not assemblies of two or more.

9.2 Two Kinds of Continuous System Models 251

There are at least two different kinds of mathematical models for continuous systems. There are the models which are based on physics models mentioned above, for example the dynamics of flows in networks, and there are the models which builds on control theory to express automatic control solutions to the monitoring & control of pipelines, for example: the opening, closing and setting of pumps, and the opening, closing and setting of valves depending on monitored values of dynamic
252 well, pipe, pump, valve, fork, join and sink attributes. Example 42 assumes the fluid mechanics domain models to complement the discrete domain model of Example 39 on page 50, whereas Example 45 on page 56 builds on Examples 42 and 39 but assumes that automatic monitoring & control requirements prescriptions have been derived, in the usual way from the former fluid mechanics domain models.

9.3 Motivation for Consolidated Models 253

By a consolidated model we shall understand a formal description that brings together both discrete (for example `TripTych` style domain description) and continuous (for example classical mathematical
254 description) models of a system.

We shall **motivate** the need for consolidated models, that is **for building both** the novel **domain descriptions,** such as this tutorial suggests, with its many aspects of discreteness, and the the **classical mathematical models**, as this section suggests, including, for example, as in the case of Example 42, fluid dynamics mathematics. This motivation really provides the justification for bringing the two disciplines together: discrete system domain modelling with continuous system physics modelling in this tutorial.

The classical mathematical models of, for example, pipelines, model physical phenomena within parts or within materials; and also combinations of *neighbouring*, parts with parts and parts with materials. But classical mathematical modelling cannot model continuous phenomena for other than definite concrete, specific combinations of parts and/or materials.

The kind of domain modelling, that is brought forward in this tutorial can, within one domain description model a whole class, indeed an indefinite, class of systems.

9.4 Generation of Consolidated Models

The idea is therefore this create a domain description for a whole, the indefinite class of "alike" systems, to wit for an indefinite class of pipelines, for an indefinite class of container lines, for an indefinite class of health care systems, and then "adorn" such a description first with classical mathematical models of simple parts of such systems; and then "replicate" these mathematical models across the indefinite class of discrete models by "pairing" each definite classical concrete mathematical model with an, albeit abstract general discrete model.

9.4.1 The Pairing Process

The "pairing process" depends on a notion of boundary condition. The boundary conditions for mereology-related parts are, yes, expressed by their mereology, that is, by how the parts fit together. The boundary conditions for continuous models are understood as the set of conditions specified for the solution to a set of differential equations at the boundary between the parts being individually modelled.

In pairing we take the "cue", i.e., directives, from the discrete domain model for the generic part and its related material since it is the more general, and "match" its mereology with the continuous mathematics model of a part and its related material

9.4.2 Matching

Matching now means the following. Let $\mathcal{D}_{\mathsf{P,M}}$ designate a $\mathcal{D}$omain $\mathcal{D}$escription for a part and/or a material, of type P, respectively M, zero or one part type and zero or one material type(s) "at a time". Let $\mathcal{M}_{\mathsf{P,M}}$ designate a $\mathcal{M}$athematical $\mathcal{M}$odel for a part and/or a material of type P, respectively M, zero or one part type and zero or one material type(s) "at a time".

Example: 43 A Transport Behaviour Consolidation. An example $\mathcal{D}_{\mathsf{P,M}}$ could be the one, for vehicles, shown in Example **??** (Pages **??**–45) as specifically expressed in the two frames: 'The Vehicle Behaviour at Hubs' on Page 44 and 'The Vehicle Behaviour along Links' on Page 44. On Page 44 of Example **??** notice vehicle vi movement at hub in formula line52a — apparently not showing any movement[46] and 52(b)iii — showing movement from hub onto link. On Page 44 of Example **??** notice vehicle vi movements along link in formula lines 53a — no movement (stopped or parked), 53(c)i — incremental movement along link, and 53(c)iiB — movement from link into hub. The corresponding example $\mathcal{M}_{\mathsf{P,M}}$ might then be modelling these movements and no movements requiring access to such attributes as link length, vehicle position, vehicle velocity, vehicle acceleration, etcetera. This model would need to abstract the non-deterministic behaviour of the driver: accelerating, decelerating or steady velocity. Example **??**'s model of vehicles' link

[46] In showing 'no movement' we claim to abstract that the vehicle remains within the hub, either moving or in a stopped or parked state. To model, in Example **??**, actual "delta" movements within hub would require some, albeit, modelling of hub topology.

position in terms of a fragment (δ) can be expected to appear in $\mathcal{M}_{\mathsf{P,M}}$ as an x, viewing the link as an x-axis. ■

Example: 44 A Pipeline Behaviour Consolidation. We continue the line of exemplifying formalisations of pipelines, cf. Examples 16 (Page 25) and 23–25 (Pages 29–31) and especially Examples 26–27 (Pages 31–32). Let the $\mathcal{D}_{\mathsf{P,M}}$ model be focused on the flows and leaks of pipeline units, cf. Examples 26 and 27. The $\mathcal{M}_{\mathsf{P,M}}$ model would then $\mathcal{M}$athematically model the fluid dynamics of the pipeline material per pipeline unit: flow and part actions and reactions for any of the corresponding $\mathcal{D}$omain models: *well* intakes, $\mathcal{D}^{\mathsf{well}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{well}}_{\mathsf{U,O}}$, *pipes*, $\mathcal{D}^{\mathsf{pipe}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{pipe}}_{\mathsf{U,O}}$, *pumps* (of various, relevant kinds), $\mathcal{D}^{\mathsf{pump}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{pump}}_{\mathsf{U,O}}$, *valves* (of various, relevant kinds), $\mathcal{D}^{\mathsf{valve}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{valve}}_{\mathsf{U,O}}$, *forks*, $\mathcal{D}^{\mathsf{fork}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{fork}}_{\mathsf{U,O}}$, *joins*, $\mathcal{D}^{\mathsf{join}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{join}}_{\mathsf{U,O}}$, and *sink* outlets $\mathcal{D}^{\mathsf{sink}}_{\mathsf{U,O}} \rightarrow \mathcal{M}^{\mathsf{sink}}_{\mathsf{U,O}}$. ■

Some more model annotations, reflecting the match between $\mathcal{D}_{\mathsf{P,M}}$ and $\mathcal{M}_{\mathsf{P,M}}$, seem relevant. Thus we further subscript $\mathcal{D}_{\mathsf{P,M}}$ optionally with a unique identifier variable, π, and the properties $p_i, p_j, ..., p_k$ where p_i is a property name of part type P or of material type M, and where these property names typically are the distinct attribute names of P and/or M, to arrive at $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,...,p_k}}$. Here π is a variable name for p:P, i.e., π is uid_P(p). Do not confuse property names, p_i etc., with part names, p.

And we likewise adorn $\mathcal{M}_{\mathsf{P,M}}$ optionally with superscripts $p_i, p_j, ..., p_k$ and subscripts $x_i, x_j, ..., x_k$ where $p_i, p_j, ..., p_k$ are as for $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,...,p_k}}$ and $x_i, x_j, ..., x_k$ are the names of the variables occurring in $\mathcal{M}_{\mathsf{P,M}}$ possibly in its partial differential equations, possibly in its difference equations, possibly in its other mathematical expressions of the $\mathcal{M}_{\mathsf{P,M}}$ model. to arrive at $\mathcal{M}^{\pi}_{\mathsf{P,M}^{p_i,p_j,...,p_k}_{x_i,x_j,...,x_k}}$

The "adornments" are the result of an analysis which identifies the variables of $\mathcal{M}_{\mathsf{P,M}}$ with the properties of $\mathcal{D}_{\mathsf{P,M}}$.

Common to all conventional mathematical models is that they all operate with a very simple type concept: **Real**s, **Int**egers, arrays (vectors, matrices, and tensors), sets of the above and sets. Common to all domain model descriptions is that they all operate with a rather sophisticated type concept: abstract types and concrete types, union ($\mathsf{T}_i|\mathsf{T}_j...$) of these, sets, Cartesians, lists, maps, and partial functions and total functions over these, etcetera.

9.4.3 Model Instantiation

The above models, $\mathcal{D}_{\mathsf{P,M}}$ and $\mathcal{M}_{\mathsf{P,M}}$, differ as follows. The $\mathcal{D}_{\mathsf{P,M}}$ models (are claimed to) hold for indefinite sets of domains "of the same kind": The axioms and invariants, cf. Example 13 on page 23, Examples 26–27 (Pages 31–32) and Example 29 on page 33, are universally quantified over all transport nets. The $\mathcal{M}_{\mathsf{P,M}}$ models express no such logic. The above difference can, however, be ameliorated. For a given, that is, an instantiated domain, we can "compile" the $\mathcal{D}_{\mathsf{P,M}}$ models into a set of models, one per part of that domain; similarly, with the binding of model $\mathcal{M}_{\mathsf{P,M}}$ variables to instantiated model $\mathcal{D}_{\mathsf{P,M}}$ attributes, we can "compile" the $\mathcal{M}_{\mathsf{P,M}}$ models into as set of — instantiated $\mathcal{M}_{\mathsf{P,M}}$ models, one per part of that domain.

[1] Model Instantiation – in Principle Since this partial evaluation compilation can be (almost) automated, there is really no reason to actually perform it; all necessary theorems should be derivable from the annotated models. $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,...,p_k}}$ and $\mathcal{M}^{\pi}_{\mathsf{P,M}^{p_i,p_j,...,p_k}_{x_i,x_j,...,x_k}}$. That is, as far as a domain understanding concerns we might, with continuous mathematical modelling and mostly discrete domain modelling very well have achieved all we can possibly, today, achieve.

[2] Model Instantiation – in Practice We continue Example 39 (Pages 50–52). The definition of pipeline_system function (Page 51) indicates the basis for an instantiation.

Example: 45 An Instantiated Pipeline System. Figure 2 on the next page indicates an instantiation. That pipeline system gives rise to the following instantiation.

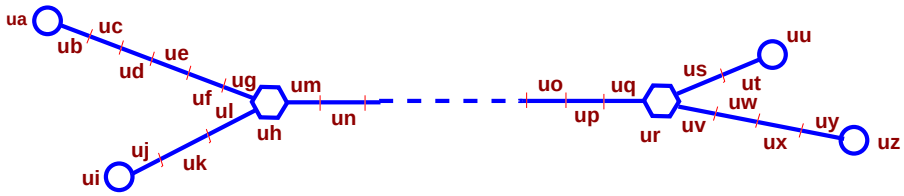


Figure 2: A specific pipeline

scada(pro)‖
unit(ua)‖unit(ub)‖unit(uc)‖unit(ud)‖unit(ue)‖unit(uf)‖unit(ug)‖
unit(uh)‖
unit(ui)‖unit(uj)‖unit(uk)‖unit(ul)‖
unit(um)‖unit(un)‖...‖unit(uo)‖unit(up)‖unit(uq)‖
unit(ur)‖
unit(us)‖unit(ut)‖unit(uu)‖
unit(uv)‖unit(uw)‖unit(ux)‖unit(uy)‖unit(uz)

We leave further details, that is, the $\mathcal{D}^{\mathsf{uid\_U(u)}}_{\mathrm{U,O}}$, on the unit, u behaviours to the reader. It is in the scada behaviour, that each of the $\mathcal{M}^{\mathsf{uid\_U(u)}}_{\mathrm{U,O}}$ models are 'instantiated'. The above instantiated model is not a domain model of a generic pipeline system but is a requirements model for the monitoring & control of a specific pipeline system. ■

9.5 An Aside on Time 273

An important aspect of domain modelling is the description of time phenomena: absolute time (or just time) and time intervals. We shall, regrettably, not cover this facet in this tutorial, but refer to a number of specifications expressed in combined uses of the `RAISE` [45] combined with the `DC: Duration Calculus` [100]. We could also express these specifications using `TLA+` [65]: Lamport's `Temporal Logic of Actions`. We otherwise refer to [8] (Chap. 15.).

9.6 A Research Agenda 274

This section opens two main lines of research problems; methodology problems cum computing science problems and computer science cum mathematics problems.

9.6.1 Computing Science cum Programming Methodology Problems 275

Some of the methodology problems are techniques for developing continuous mathematics models — which we leave to the relevant fields of physics and control theory to "deliver"; contained in this are more detailed techniques for matching $\mathcal{D}_{\mathsf{D,M}}$ and $\mathcal{M}_{\mathsf{D,M}}$ models, that is, for identifying and pairing the p_is and x_is in $\mathcal{D}^{\pi}_{\mathsf{P,M}_{p_i,p_j,\ldots,p_k}}$ and $\mathcal{M}^{\pi}_{\mathsf{P,M}^{p_i,p_j,\ldots,p_k}_{x_i,x_j,\ldots,x_k}}$ and for instantiating these.
A problem of current programming methodology in that it has for most of its "existence" relied 276
on discrete mathematics and not sufficiently educated and trained its candidates in continuous mathematics.

9.6.2 Mathematical Modelling Problems 277

Some of the open mathematics problems are the lack of well-understood interfaces between discrete mathematics models and continuous mathematics models; and the lack of proof systems across the two modes of expression. 278

By well-understood interfaces between the two modes of expression, the discrete mathematics models and the continuous mathematics models; we mean that the semantics models of the discrete mathematics formal specification languages and the continuous mathematics specification notations, at this time, August 10, 2012, are not commensurate, that is, do not "carry over": a variable, a of some, even abstract type, say A, cannot easily be related to what it has to be related to, namely a variable, x of some concrete, mathematical type, say **Real** or **Int**eger, or arrays of these, etc.

Lack of proof systems across the two modes of expression. the discrete mathematics models and the continuous mathematics models; we mean, firstly, that the former problem of lack of clear a↔x relations is taken to prevent such proof systems, secondly, that mathematics essentially does not embody a "formal language". But nobody is really looking into, that is, researching possible "solutions" to these problems.

10 Discussion of Entities

We have examined the concepts of entities, endurant and perdurant.

We have not examined those "things" (of a domain) which "fall outside" this categorisation. That would lead to a rather lengthy discourse. In the interest of "really understanding" what can be described such a computer science study should be made. Philosophers have clarified the issues in centuries of studies. Their interest is in identifying the issues and clarifying the questions. Computer scientists are interested in answers.

We see entities as either endurants or perdurants or as either discrete or continuous. We analyse discrete endurants into atomic and composite parts with observers, unique identifiers, mereology and attributes. And we analyse perdurants into actions, events and behaviours.

This domain ontology is entirely a pragmatic one: it appears to work; it has been used in the description of numerous cases; it leads to descriptions which in a straightforward manner lend themselves to the "derivation" of significant fragments of requirements; and appears not to stand in the way of obtaining remaining requirements.

Most convincingly to us is that the concepts of our approach endurants and perdurants, atomic and composite parts, mereology and attributes, actions, events and behaviours fit it with major categories of philosophically analyses.

Part III
Calculus

11 Towards a Calculus of Domain Discoverers

The 'towards' term is significant. We are not presenting a "ready to serve" comprehensive, tested and tried calculus. We hope that the one we show you is interesting. It is, we think, the first time such a calculus is presented.

By a domain description calculus or, as we shall also call it, either a domain discovery calculus or a calculus of domain discoverers we shall understand an algebra, that is, a set of meta-operations and a pair of a fixed domain and a varying repository.

The meta-operations will be outlined in this section. The fixed domain is of the kind of domains alluded to in the previous section. The varying repository contains fragments of a description of the fixed domain.

The meta-operators are referred to as either domain analysis meta-functions or domain discovery meta-functions. The former are carried out by the domain analyser when inquiring (the domain) as to its properties. The latter are carried out by the domain describer when deciding upon which descriptions "to go for" ! The two persons can be the same one domain engineer.

The operators are referred to as meta-functions, or meta-linguistic functions, since they are applied and calculated by humans, i.e., the domain describers. They are directives, here set down

on paper, which can be referred to by the domain describers while carrying out their analytic and creative work.

11.1 Introductory Notions 289

In order to present the operators of the calculus we must clear a few concepts.

11.1.1 Discovery

By a domain discovery calculus we shall understand a set of operations (the domain discoverers), which when applied to a domain by a human agent, the domain describer, and, "bit-by-bit", yield domain description texts. 290

The domain discoverers are applied "mentally". That is, not in a mechanisable way. It is not like when procedure calls invoke computations of a computer. But they are applied by the domain describer. That person is to follow the ideas laid down for these domain discoverers (as they were in Sect. 2). They serve to guide the domain engineer to discoverer the desired domain entities and their properties.

In this section we shall review an ensemble of (so far) nine domain discoverers and (so far) four domain analysers. 291

We list the nine domain discoverers.

- [Sect. 11.3.2 Page 64] PART_SORTS,
  [Sect. 11.3.1 Page 63] MATEREIAL_SORTS,
  [Sect. 11.3.3 Page 64] PART_TYPES,
  [Sect. 11.3.4 Page 65] UNIQUE_ID,
  [Sect. 11.3.5 Page 65] MEREOLOGY,
  [Sect. 11.3.6 Page 66] ATTRIBUTES,
  [Sect. 11.3.7 Page 68] ACTION_SIGNATURES,
  [Sect. 11.3.8 Page 68] EVENT_SIGNATURES and
  [Sect. 11.3.9 Page 69] BEHAVIOUR_SIGNATURES.

11.1.2 Analysis 292

In order to "apply" these domain discoverers certain conditions must be satisfied. Some of these condition inquiries can be represented by (so far) four domain analysers.

- [Sect. 11.2.1 Page 61] IS_MATERIALS_BASED,
  [Sect. 11.2.2 Page 62] IS_ATOM,
  [Sect. 11.2.2 Page 62] IS_COMPOSITE and
  [Sect. 11.2.3 Page 62] HAS_A_CONCRETE_TYPE.

11.1.3 Domain Indexes 293

In order to discover, the domain describer must decide on *"where & what in the domain"* to analyse and describe. One can, for this purpose, think of the domain as semi-lattice-structured. The root of the lattice is then labelled Δ. Let us refer to the domain as Δ. We say that it has index $\langle\Delta\rangle$. Initially we analyse the usually composite Δ domain to consist of one or more distinctly typed parts $\mathsf{p}_1{:}\mathsf{t}_1$, $\mathsf{p}_2{:}\mathsf{t}_2$, ..., $\mathsf{p}_m{:}\mathsf{t}_m$. Each of these have indexes $\langle\Delta, t_i\rangle$. So we view Δ, in the semi-lattice, to be the join of m sub-semi-lattices whose roots we shall label with t_1, t_2, ..., t_m. And so forth 294 for any composite part type t_i, etcetera. It may be that any two or more such sub-semi-lattice root types, t_{i_j}, t_{i_j}, ..., t_{i_k} designate the same, shared type t_{i_x}, that is $\mathsf{t}_{i_j} = \mathsf{t}_{i_j} = \ldots = \mathsf{t}_{i_k} = \mathsf{t}_{i_x}$. If so then the k sub-semi-lattices are "collapsed" into one sub-semi-lattice. The building of the semi-lattice terminates when one can no longer analyse part types into further sub-semi-lattices, that is, when these part types are atomic. 295

That is, the roots of the sub-trees of the Δ tree are labelled with type names. Every point in 296 the semi-lattice can be identified by a domain index. The root is defined to have index $\langle\Delta\rangle$.

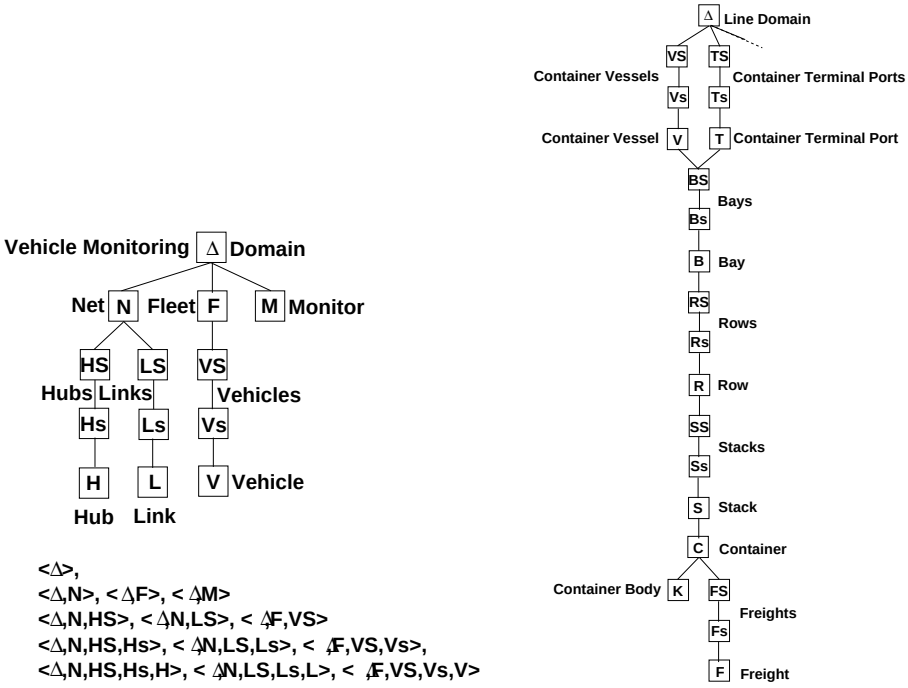


Figure 3: Domain indices

The immediate sub-semi-lattices of Δ have domain indexes $\langle\Delta,t_1\rangle$, $\langle\Delta,t_2\rangle$, ..., $\langle\Delta,t_m\rangle$. And so forth. If $\ell^\frown\langle t\rangle$ is a prefix of another domain index, say $\ell^\frown\langle t,t'\rangle$, then t designates a composite type. For every domain index, $\ell^\frown\langle t\rangle$, that index designates the type t domain type texts. These texts consists of several sub-texts. There are the texts directly related to the parts, p:P: the observer functions, obs_$\cdots$, if type t is composite, the unique identifier functions, uid_P, the mereology function, mereo_P, and the attribute functions, attr_$\cdots$. To the above "add" possible auxiliary types and auxiliary functions as well as possible axioms. Then there are the texts related to actions, events, and behaviours "based" (primarily) on parts p:P. These texts consists of function signatures (for actions, events, and behaviours), function definitions for these, and channel declarations and channel message type definitions for behaviours. We shall soon see examples of the above.

But not all can be "discovered" by just examining the domain from the point of view of a sub-semi-lattice type. Many interesting action, event and behaviour signatures depend on domain type texts designated by "roots" of disjoint sub-trees of the semi-lattice. Each such root has its own domain index. Together a meet of the semi-lattice is defined by the set of disjoint domain indices: $\{\ell_i,\ell_j,\cdots,\ell_k\}$. It is thus that we arrive at a proper semi-lattice structure relating the various entities of the domain rooted in Δ.

The domain discoverers are therefore provided with arguments: either a single domain index, DOMAIN_FUNCTION(ℓ), or a pair, DOMAIN_FUNCTION(ℓ)($\{\ell_i,\ell_j,\cdots,\ell_k\}$), the single domain index ℓ and a set of domain indices, $\{\ell_i,\ell_j,\cdots,\ell_k\}$ where DOMAIN_FUNCTION is any of the domain discoverers or domain analysers listed in Sects. 11.1.1–11.1.2.

11.1.4 The ℜepository

We have yet to give the full signature of the domain discoverers and domain analysers. One argument of these meta-functions was parts of the actual domain as designated by the domain indices. Another argument is to be the ℜepository of description texts being inspected (together with the sub-domain) when analysing that sub-domain and being updated when "generating" the "discovered" description texts. We can assume, without loss of generality, that the ℜepository of

description texts is the description texts discovered so far. The result of domain analysis is either undefined or a truth value. We can assume, without any loss of generality that that result is not recorded. The result of domain discovery is either undefined or is a description text consisting of two well-defined fragments: a narrative text, and a formal text. Those well-defined texts are "added" to the text of the ℜepository of description texts. For pragmatic reasons, when we explain the positive effect of domain discovery, then we show just this "addition" to the ℜepository. 303

98. The proper type of the discover functions is therefore:

    98. DISCOVER_FUNCTION: Index→Index-**set**→ℜ$\xrightarrow{\sim}$ℜ

In the following we shall omit the ℜepository argument and result.

99. So, instead of showing the discovery function invocation and result as:

    99. DISCOVER_FUNCTION(ℓ)(ℓset)(ρ) = ρ'

where ρ' incorporates a pair of texts and RSL formulas,

100. we shall show the discover function signature, the invocation and the result as:

    100. DISCOVER_FUNCTION: Index→Index-**set**$\xrightarrow{\sim}$(Narr_Text×RSL_Text)

    100. DISCOVER_FUNCTION(ℓ)(ℓset): (narr_text,RSL_text)

11.2 Domain Analysers 304

Currently we identify four analysis functions. As the discovery calculus evolves (through further practice and research) we expect further analysis functions to be identified.

11.2.1 IS_MATERIALS_BASED 305

We refer to Sects. 3.2 and 5.

You are reminded of the *Continuous Endurant Modelling* frame on Page 32.

IS_MATERIALS_BASED

An early decision has to be made as to whether a domain is significantly based on materials or not:

101. IS_MATERIALS_BASED($\langle\Delta_{\text{Name}}\rangle$).

If Item 101 holds of a domain Δ_{Name} then the domain describer can apply MATERIAL_SORTS (Item 103 on page 63).

306

Example: 46 Pipelines and Transports: Materials or Parts.

- IS_MATERIALS_BASED($\langle\Delta_{\text{Pipeline}}\rangle$) = **true**.
- IS_MATERIALS_BASED($\langle\Delta_{\text{Transport}}\rangle$)= **false**.

11.2.2 IS_ATOM, IS_COMPOSITE

307

We refer to Sects. 4.2.2.2–4.2.3.3 (Pages 19–20).

During the discovery process discrete part types arise (i.e., the names are yielded) and these may either denote atomic or composite parts. The domain describer must now decide as to whether a named, discrete type is atomic or is composite.

308

IS_ATOM

The IS_ATOM analyser serves that purpose:

value
IS_ATOM: Index $\xrightarrow{\sim}$ **Bool**
IS_ATOM($\ell^\frown\langle$t$\rangle$) ≡ **true** | **false** | **chaos**

The analysis is undefined for ill-formed indices.

Example: 47 Transport Nets: Atomic Parts (II). We refer to Example 3 (Page 10).

IS_ATOM(⟨Δ,N,HS,Hs,H⟩), IS_ATOM(⟨Δ,N,LS,Ls,L⟩)
∼IS_ATOM(⟨Δ,N,HS,Hs⟩), ∼IS_ATOM(⟨Δ,N,LS,Ls⟩) ■

309

IS_COMPOSITE

The IS_COMPOSITE analyser is similarly applied by the domain describer to a part type t to help decide whether t is a composite type.

value
IS_COMPOSITE: Index $\xrightarrow{\sim}$ **Bool**
IS_COMPOSITE($\ell^\frown\langle$t$\rangle$) ≡ **true** | **false** | **chaos**

310

Example: 48 Transport Nets: Composite Parts. We refer to Example 3 (Page 10)

IS_COMPOSITE(⟨Δ⟩), IS_COMPOSITE(⟨Δ,N⟩)
IS_COMPOSITE(⟨Δ,N,HS,Hs⟩), IS_COMPOSITE(⟨Δ,N,LS,Ls⟩)
∼IS_COMPOSITE(⟨Δ,N,HS,Hs,H⟩), ∼IS_COMPOSITE(⟨Δ,N,LS,Ls,L⟩) ■

11.2.3 HAS_A_CONCRETE_TYPE

311

We refer to Sect. 4.2.4.4 (Page 20).

Sometimes we find it expedient to endow a "discovered" sort with a concrete type expression, that is, "turn" a sort definition into a concrete type definition.

HAS_A_CONCRETE_TYPE

102. Thus we introduce the analyser:

102 HAS_A_CONCRETE_TYPE: Index $\xrightarrow{\sim}$ **Bool**
102 HAS_A_CONCRETE_TYPE($\ell^\frown\langle$t$\rangle$): **true** | **false** | **chaos**

312

Example: 49 Transport Nets: Concrete Types . We refer to Example 3 (Page 10) while exemplifying four cases:

HAS_A_CONCRETE_TYPE(⟨Δ,N,HS,Hs⟩)
HAS_A_CONCRETE_TYPE(⟨Δ,N,LS,Ls⟩)
∼ HAS_A_CONCRETE_TYPE(⟨Δ,N,HS,Hs,H⟩)
∼ HAS_A_CONCRETE_TYPE(⟨Δ,N,LS,Ls,L⟩) ■

313

We remind the reader that it is a decision made by the domain describer as to whether a part type is to be considered a sort or be given a concrete type.

Sect. 11.3.3 covers a domain discoverer related to the positive outcome of the above inquiry.

11.3 Domain Discoverers

314

A domain discoverer is a mental tool. It takes a written form shown earlier. It is to be "applied" by a human, the domain describer. The domain describer applies the domain discoverer to a fragment of the domain, as it is: "out there" ! 'Application' means the following. The domain describer 315
examines the domain as directed by the explanation given for the domain discoverer — as here, in Sects. 2–11. As the brain of the domain describer views, examines, analyses, a domain index-designated fragment of the domain, ideas as to which domain concepts to capture arise and these take the form of pairs of narrative and formal texts. This section will outline these 'texts'.

11.3.1 MATERIAL_SORTS

316

We refer to Sects. 3.2 and 5.

MATERIAL_SORTS

103. The MATERIAL_SORTS discovery function applies to a domain, usually designated by ⟨Δ_{Name}⟩ where Name is a pragmatic hinting at the domain by name.
104. The result of the domain discoverer applying this meta-function is some narrative text
105. and the **type**s of the discovered materials
106. usually affixed a comment
    a which lists the "somehow related" part types
    b and their related materials observers.

317

103. MATERIAL_SORTS: ⟨Δ⟩ → (**Text** × RSL)
103. MATERIAL_SORTS(⟨Δ_{Name}⟩):
104. [narrative text ;
105. **type** M_a, M_b, ..., M_c **materials**
106. **comment**: related part **type**s: P_i, P_j, ..., P_k
106. obs_M_n : P_m → M_n, ...]
101. **pre**: IS_MATERIALS_BASED(⟨Δ_{Name}⟩)

318

Example: 50 Pipelines: Material.

- MATERIAL_SORTS(⟨$\Delta_{\mathsf{Oil\ Pipeline\ System}}$⟩):
  [The oil pipeline system is focused on oil ;
  type O **material**
  comment related part type: U, obs_O: U → O]

11.3.2 PART_SORTS

319

We refer to Sect. 4.2 on page 18.

PART_SORTS

107. The part type discoverer PART_SORTS
    a applies to a simply indexed domain, $\ell^\frown\langle \mathsf{t}\rangle$,
    b where t denotes a composite type, and yields a pair
        i. of narrative text[47] and
        ii. formal text which itself consists of a pair:
            A. a set of type names
            B. each paired with a part (sort) observer.

320

value
107. PART_SORTS: Index $\xrightarrow{\sim}$ (**Text**×RSL)
107a. PART_SORTS($\ell^\frown\langle \mathsf{t}\rangle$):
107(b)i. [narrative, possibly enumerated texts ;
107(b)iiA. **type** t_1,t_2,...,t_m,
107(b)iiB. **value** obs_t_1:t→t_1,obs_t_2:t→t_2,...,obs_t_m:t→t_m
107b. **pre**: IS_COMPOSITE($\ell^\frown\langle \mathsf{t}\rangle$)]

321

Example: 51 Transport: Part Sorts. We apply a concrete version of the above sort discoverer to the road traffic system domain Δ. See Example 37.

- PART_SORTS($\langle\Delta\rangle$):
  [the vehicle monitoring domain contains three sub-parts: net, fleet and monitor ;
  type N, F, M, **value** obs_N: Δ → N, obs_F: Δ → F, obs_M: Δ → M]
- PART_SORTS($\langle\Delta$,N$\rangle$):
  [the net domain contains two sub-parts: sets of hubs and sets of link ;
  322 **type** HS, LS, **value** obs_HS: N → HS, obs_LS: N → LS]
- PART_SORTS($\langle\Delta$,F$\rangle$):
  [the fleet domain consists of one sub-domain: set of vehicles;
  type VS, **value** obs_VS: F → VS] ■

11.3.3 PART_TYPES

323

We refer to Sect. 4.2.4.4 (Page 20).

PART_TYPES

108. The PART_TYPES discoverer applies to a composite sort, t, and yields a pair
    a of narrative, possibly enumerated texts [omitted], and
    b some formal text:
        i. a type definition, $\mathsf{t}_c = \mathsf{te}$,
        ii. together with the sort definitions of so far undefined type names of te.
        iii. An observer function observes t_c from t.

[47] In this tutorial we omit the narratives.

iv. The PART_TYPES discoverer is not defined if the designated sort is judged to not warrant a concrete type definition.

```
108.   PART_TYPES: Index ~→ (Text×RSL)
108.   PART_TYPES(ℓ^⟨t⟩):
108a.      [ narrative, possibly enumerated texts ;
108(b)i.     type t_c = te,
108(b)ii.          t_α, t_β, ..., t_γ,
108(b)iii.   value obs_t_c: t → t_c
108(b)iv.    pre: HAS_CONCRETE_TYPE(ℓ^⟨t⟩) ]
108(b)ii.    where: type expression te contains
108(b)ii.              type names t_α, t_β, ..., t_γ
```

Example: 52 Transport: Concrete Part Types. Continuing Examples ??–51 and Example 3 – we omit narrative informal texts.

```
PART_TYPES(⟨Δ,F,VS⟩):
    type V, Vs=V-set, value obs_Vs: VS→Vs
PART_TYPES(⟨Δ,N,HS⟩):
    type H, Hs=H-set, value obs_Hs: HS→Hs
PART_TYPES(⟨Δ,N,LS⟩):
    type L, Ls=L-set, value obs_Ls: LS→Ls
```

■

11.3.4 UNIQUE_ID

We refer to Sect. 4.3.1.1 (Page 22).

We associate with every part type t, a unique identity type ti.

UNIQUE_ID

109. For every part type t we postulate a unique identity analyser function uid_t.

```
value
109.  UNIQUE_ID: Index → (Text×RSL)
109.  UNIQUE_ID(ℓ^⟨t⟩):
109.  [ narrative, possibly enumerated text ;
109.     type ti
109.     value uid_t: t → ti ]
```

Example: 53 Transport Nets: Unique Identifiers. Continuing Example 3:

```
UNIQUE_ID(⟨Δ,HS,Hs,H⟩): type H, HI, value uid_H→HI
UNIQUE_ID(⟨Δ,LS,Ls,L⟩): type L, LI, value uid_L→LI
```

■

11.3.5 MEREOLOGY

We refer to Sect. 4.3.2.2 (Pages 22–25).

Given a part, p, of type t, the mereology, MEREOLOGY, of that part is the set of all the unique identifiers of the other parts to which part p is part-ship-related as "revealed" by the mereo_ti$_i$ functions applied to p. Henceforth we omit the otherwise necessary narrative texts.

MEREOLOGY

110. Let type names t_1, t_2, ..., t_n denote the types of all parts of a domain.
111. Let type names ti_1, ti_2, ..., ti_n[48], be the corresponding type names of the unique identifiers of all parts of that domain.
112. The mereology analyser MEREOLOGY is a generic function which applies to a pair of an index and an index set and yields some structure of unique identifiers. We suggest two possibilities, but otherwise leave it to the domain analyser to formulate the mereology function.
113. Together with the "discovery" of the mereology function there usually follows some axioms.

329

type
110. t_1, t_2, ..., t_n
111. $\mathsf{t}_{idx} = \mathsf{ti}_1 \mid \mathsf{ti}_2 \mid ... \mid \mathsf{ti}_n$

112. MEREOLOGY: Index $\xrightarrow{\sim}$ Index-**set** $\xrightarrow{\sim}$ (**Text**×RSL)
112. MEREOLOGY($\ell^\frown\langle\mathsf{t}\rangle$)($\{\ell_i{}^\frown\langle\mathsf{t}_j\rangle,...,\ell_k{}^\frown\langle\mathsf{t}_l\rangle\}$):
112. [narrative, possibly enumerated texts ;
112. **either:** {}
112. **or:** **value** mereo_t: t → ti_x
112. **or:** **value** mereo_t: t → ti_x-**set** × ti_y-**set** × ... × ti_x-**set**
113. **axiom** $\mathcal{P}$redicate over values of t′ and t_{idx}]

where none of the ti_x, ti_y, ..., ti_z are equal to ti.

330

Example: 54 Transport Net Mereology. Examples:

- MEREOLOGY(⟨Δ,N,HS,Hs,H⟩)({⟨Δ,N,LS,Ls,L⟩}):
  value mereo_H→LI-**set**
- MEREOLOGY(⟨Δ,N,LS,Ls,L⟩)({⟨Δ,N,HS,Hs,H⟩}):
  value mereo_L→HI-**set**
  axiom see Example 12 Page 23.

11.3.6 ATTRIBUTES

331

We refer to Sect. 4.3.3.3 (Pages 26–27).

A general attribute analyser analyses parts beyond their unique identities and possible mereologies. Part attributes have names. We consider these names to also abstractly name the corresponding attribute types.

332

ATTRIBUTES

114. Attributes have types. We assume attribute type names to be distict from part type names.
115. ATTRIBUTES applies to parts of type t and yields a pair of
    a. narrative text and
    b. formal text, here in the form of a pair
        i. a set of one or more attribute types, and
        ii. a set of corresponding attribute observer functions attr_at, one for each attribute

[48] We here assume that all parts have unique identifications.

```
                sort at of t.
type
114.  at = at1 | at2 | ... | atn
value
115.  ATTRIBUTES: Index → (Text×RSL)
115.  ATTRIBUTES(ℓ^⟨t⟩):
115a.       [ narrative, possibly enumerated texts ;
115(b)i.      type at1, at2, ..., atm
115(b)ii.     value attr_at1:t→at1,attr_at2:t→at2,...,attr_atm:t→atm  ]

where m≤n
```

Example: 55 Transport Nets: Part Attributes. We exemplify attributes of composite and of atomic parts — omitting narrative texts:

```
ATTRIBUTES(⟨Δ⟩):
  type Domain_Name, ...
  value attr_Domain_Name: Δ → Domain_Name, ...
```

where Domain_Name could include *State Roads* or *Rail Net*. etcetera.

```
ATTRIBUTES(⟨Δ,N⟩):
  type
    Sub_Domain_Name      ex.: State Roads
    Sub_Domain_Location  ex.: Denmark
    Sub_Domain_Owner     ex.: The Danish Road Directorate
    ...
    Length               ex.: 3.786 Kms.
  value
    attr_Sub_Domain_Name: N → Sub_Domain_Name
    attr_Sub_Domain_Location: N → Sub_Domain_Location
    attr_Sub_Domain_Owner: N → Sub_Domain_Owner
    ...
    attr_Length: N → Length
```

```
ATTRIBUTES(⟨Δ,N,LS,Ls,L⟩):
  type LOC, LEN, ...
  value attr_LOC: L → LOC, attr_LEN: L → LEN, ...

ATTRIBUTES(⟨Δ,N,LS,Ls,L⟩)({,⟨Δ,N,HS,Hs,H⟩}):
  type
    LΣ=HI-set
    LΩ=LΣ-set
  value
    attr_LΣ:L→LΣ
    attr_LΩ:L→LΩ
```

where LOC might reveal some Bézier curve[49] representation of the possibly curved three dimensional location of the link in question, LEN might designate length in meters, LΣ designates the state of the link, LΩ designates the space of all allowed states of the link. ■

[49] http://en.wikipedia.org/wiki/Bézier_curve

11.3.7 ACTION_SIGNATURES

340

We refer to Sect. 8.2.2.2 (Page 35).

We really should discover actions, but actually analyse function definitions. And we focus, in this tutorial, on just "discovering" the function signatures of these actions. By a function signature, to repeat, we understand a functions name, say fct, and a function type expression (te), say dte$\xrightarrow{\sim}$rte where dte defines the type of the function's definition set and rte defines the type
341 of the function's image, or range set. We use the term 'functions' to cover actions, events and behaviours. We shall in general find that the signatures of actions, events and behaviours depend on types of more than one domain. Hence the schematic index set $\{\ell_1\hat{}\langle t_1\rangle,\ell_2\hat{}\langle t_2\rangle,...,\ell_n\hat{}\langle t_n\rangle\}$ is
342 used in all action, event and behaviour discoverers.

ACTION_SIGNATURES

116. The ACTION_SIGNATURES meta-function, besides narrative texts, yields
   a a set of auxiliary sort or concrete type definitions and
   b a set of action signatures each consisting of an action name and a pair of definition set and range type expressions where
   c the type names that occur in these type expressions are defined by in the domains indexed by the index set.

343

```
116   ACTION_SIGNATURES: Index → Index-set ~→ (Text×RSL)
116   ACTION_SIGNATURES(ℓ^⟨t⟩)({ℓ1^⟨t1⟩,ℓ2^⟨t2⟩,...,ℓn^⟨tn⟩}):
116    [ narrative, possibly enumerated texts ;
116      type ta,tb,... tc,
116      value
116b       act_i:te_{i_d} ~→ te_{i_r},act_j:te_{j_d} ~→ te_{j_r},...,act_k:te_{k_d} ~→ te_{k_r}
116c     where:
116c       type names in  te_{(i|j|...|k)_d}  and in  te_{(i|j|...|k)_r}  are either
116c       type names ta, tb, ... tc or are type names defined by the
116c       indices which are prefixes of  ℓm^⟨Tm⟩  and where Tm is
116c       in some signature act_{i|j|...|k} ]
```

344

Example: 56 Transport Nets: Action Signatures.

- ACTION_SIGNATURES(⟨Δ,N,HS,Hs,H⟩)({⟨Δ,N,LS,Ls,L⟩}):
  insert_H: N → H ~→ N
  remove_H: N → HI ~→ N
  ...
- ACTION_SIGNATURES(⟨Δ,N,LS,Ls,L⟩)({⟨Δ,N,HS,Hs,H⟩}):
  insert_L: N → L ~→ N
  remove_L: N → LI ~→ N
  ...

where ··· refer to the possibility of discovering further action signatures "rooted" in ⟨Δ,N,HS,Hs,H⟩, respectively ⟨Δ,N,LS,Ls,L⟩. ■

11.3.8 EVENT_SIGNATURES

345

We refer to Sect. 8.3.2.2 (Page 38).

EVENT_SIGNATURES

117. The EVENT_SIGNATURES meta-function, besides narrative texts, yields
   a a set of auxiliary event sorts or concrete type definitions and
   b a set of event signatures each consisting of an event name and a pair of definition set and range type expressions where
   c the type names that occur in these type expressions are defined either in the domains indexed by the indices or by the auxiliary event sorts or types.

346

```
117   EVENT_SIGNATURES: Index → Index-set ~→ (Text×RSL)
117   EVENT_SIGNATURES(ℓ^⟨t⟩)({ℓ1^⟨t1⟩,ℓ2^⟨t2⟩,...,ℓn^⟨tn⟩}):
117a    [ narrative, possibly enumerated texts omitted ;
117a     type  ta,tb,... tc,
117b     value
117b       evt_predi: ted_i × ter_i → Bool
117b       evt_predj: ted_j × ter_j → Bool
117b       ...
117b       evt_predk: ted_k × ter_k → Bool ]
```

117c **where:** t is any of t_a,t_b,...,t_c or type names listed in in indices; type names of the '*d*'efinition set and '*r*'ange set type expressions te_d and te_r are type names listed in domain indices or are in t_a,t_b,...,t_c, the auxiliary discovered event types. ■

347

Example: 57 Transport Nets: Event Signatures. We refer to Example 36 on page 38. The omitted narrative text would, if included, as it should, be a subset of the Items 23–26 texts on Page 38.

- EVENT_SIGNATURES(⟨Δ,N,LS,Ls,L⟩)({⟨Δ,N,HS,Hs,H⟩⟩}):
  value
    link_disappearance: N × N ~→ **Bool**
    link_disappearance(n,n′) ≡
      ∃ ℓ:L • l ∈ obs_Ls(n) ⇒ pre_cond(n,ℓ) ∧ post_cond(n,ℓ,n′)

  ... [possibly further, discovered event]
  ... [signatures "rooted" in ⟨Δ,N,LS,Ls,L⟩] ■

The undefined pre_ and post_conditions were "fully discovered" on Pages 39 and 39.

11.3.9 BEHAVIOUR_SIGNATURES

348

We refer to Sect. 8.4. (Pages 40–53).

We choose, in this tutorial, to model behaviours in CSP[50]. This means that we model (synchronisation and) communication between behaviours by means of messages m of type M, CSP channels (**channel** ch:M) and CSP

- output: ch!e [offer to deliver value of expression e on channel ch], and
- input: ch? [offer to accept a value on channel ch].

349

We allow for the declaration of single channels as well as of one, two, ..., *n* dimensional arrays of channels with indexes ranging over channel index types

- **type** Idx, CIdx, RIdx ...:
- **channel** ch:M, { ch_v[vi]:M′|vi:Idx }, { ch_m[ci,ri]:M″|ci:CIdx,ri:RIdx }, ...

etcetera. We assume some familiarity with CSP [53] (or even RSL/CSP [7] [Chapter 21]). 350

A behaviour usually involves two or more distinct sub-domains.

Example: 58 Vehicle Behaviour. Let us illustrate that behaviours usually involve two or more distinct sub-domains. A vehicle behaviour, for example, involves the vehicle sub-domain, the hub sub-domain (as vehicles pass through hubs), the link sub-domain (as vehicles pass along links) and, for the road pricing system, also the monitor sub-domain. ■ 351

BEHAVIOUR_SIGNATURES

118. The BEHAVIOUR_SIGNATURES meta-function, besides narrative texts, yields

119. It applies to a set of indices and results in a pair,

    a a narrative text and

    b a formal text:

        i. a set of one or more message types,
        ii. a set of zero, one or more channel index types,
        iii. a set of one or more channel declarations,
        iv. a set of one or more process signatures with each signature containing a behaviour name, an argument type expression, a result type expression, usually just **Unit**, and
        v. an input/output clause which refers to channels over which the signatured behaviour may interact with its environment.

352

```
118.  BEHAVIOUR_SIGNATURES: Index→ Index-set ~→ (Text×RSL)
118.  BEHAVIOUR_SIGNATURES(ℓ^⟨t⟩)({ℓ1^⟨t1⟩,ℓ2^⟨t2⟩,...,ℓn^⟨tn⟩}):
119a.       [ narrative, possibly enumerated texts ;
119(b)i.     type    m = m1 | m 2 | ... | mμ, μ≥1
119(b)ii.              i = i1 | i2 | ... | in, n≥0
119(b)iii.   channel c:m, {vc[x]|x:ia}:m, {mc[x,y]|x:ib,y:ic}:m,...
119(b)iv.    value
119(b)iv.      bhv1: ate1 → inout1 rte1,
119(b)iv.      ... ,
119(b)iv.      bhvm: atem → inoutm rtem. ]
119(b)iv.  where  type expressions atei and rtei for all i involve at least
119(b)iv.         two types t'i, t''j of respective indexes ℓi^⟨ti⟩, ℓj^⟨tj⟩,
119(b)v.   where  Unit may appear in either atei or rtej or both.
119(b)v.   where  inouti: in k | out k | in,out k
119(b)v.   where  k: c or vc[x] or {vc[x]|x:ia•x ∈ xs} or
119(b)v.            {mc[x,y]|x:ib,y:ic • x ∈ xs ∧ y ∈ ys} or ...
```

353

Example: 59 Vehicle Transport: Behaviour Signatures. We refer to Example 37.

```
BEHAVIOUR_SIGNATURES(⟨Δ,F,VS,Vs,V⟩)({⟨Δ,M⟩}):
  [ With each vehicle we associate behaviour with the following
        arguments: the vehicle identifier, the vehicle parts, and
        the vehicle position.  The vehicle communicates with
        the monitor process over a vehicle to monitor array of
```

[50] Other behaviour modelling languages are Petri Nets, MSCs: Message Sequence Charts, Statechart etc. We invite the reader to suggest corresponding 'discovery' techniques and tools.

```
        channels, one for each vehicle ... ;
    type
        VP
    channel
        {vm[ vi ]|vi:VI • vi ∈ vis}:VP
    value
        veh: vi:VI → v:V → vp:VP → out vm[ vi ]  Unit ]
```

354

```
BEHAVIOUR_SIGNATURES(⟨Δ,M⟩)({⟨Δ,F,VS,Vs,V⟩}):
  [ With the monitor part  we associate a behaviour with the monitor
        part as only argument. The monitor accepts communications
        from vehicle behaviours ... ;
    value
        mon: M → in {vm[ vi ]|vi:VI • vi ∈ vis},clkm_ch  Unit ]
```

The "discovery" of vehicle positions into positions on a link, some fraction down that link, or at a hub, that "discovery", is left for further analysis. We refer to Page 43 (Items 47a–47(a)iii). ■

11.4 Order of Analysis and "Discovery"

355

Analysis and "discovery", that is, the "application" of the analysis meta-functions of Sect. 11.1.2 and the "discovery" meta-functions of Sect. 11.1.1 has to follow some order: starts at the "root", that is with index $\langle\Delta\rangle$, and proceeds with indices appending part domain type names already discovered.

11.5 Analysis and "Discovery" of "Leftovers"

356

The analysis and discovery meta-functions focus on types, that is, the types of abstract parts, i.e., sorts, of concrete parts, i.e., concrete types, of unique identifiers, of mereologies, and of attributes – where the latter has been largely left as sorts. In this tutorial we do not suggest any meta- 357
functions for such analyses that may lead to concrete types from non-part sorts, or to action, event and behaviour definitions say in terms of pre/post-conditions, etcetera. So, for the time, we suggest, as a remedy for the absence of such "helpers", good "old-fashioned" domain engineer ingenuity.

11.6 Laws of Domain Descriptions

358

By a domain description law we shall understand some desirable property that we expect (the 'human') results of the (the 'human') use of the domain description calculus to satisfy. We may think of these laws as axioms which an ideal domain description ought satisfy, something that domain describers should strive for. 359

Notational Shorthands:

- $(f;g;h)(\Re) = h(g(f(\Re)))$
- $(f_1;f_2;\ldots;f_m)(\Re) \simeq (g_1;g_2;\ldots;g_n)(\Re)$
  means that the two "end" states are equivalent modulo appropriate renamings of types, functions, predicates, channels and behaviours.
- $[f;g;\ldots;h;\alpha]$
  stands for the Boolean value yielded by α (in state $\Re$).

11.6.1 1st Law of Commutativity

We make a number of assumptions: the following two are well-formed indices of a domain: ι': $\langle\Delta\rangle\hat{}\ell'\hat{}\langle\mathsf{A}\rangle$, ι'': $\langle\Delta\rangle\hat{}\ell''\hat{}\langle\mathsf{B}\rangle$, where ℓ' and ℓ'' may be different or empty ($\langle\rangle$) and A and B are distinct; that $\mathcal{F}$ and $\mathcal{G}$ are two, not necessarily distinct discovery functions; and that the domain at ι' and at ι'' have not yet been explored.

We wish to express, as a desirable property of domain description development that exploring domain Δ at either ι' first and then ι'' or at ι'' first and then ι', the one right after the other (hence the ";"), ought yield the same partial description fragment:

120. $(\mathcal{G}(\iota'')\ ;\ (\mathcal{F}(\iota')))(\Re) \simeq (\mathcal{F}(\iota')\ ;\ (\mathcal{G}(\iota'')))(\Re)$

When a domain description development satisfies Law 120., under the above assumptions, then we say that the development — modulo type, action, event and behaviour name "assignments" — satisfies a mild form of commutativity.

11.6.2 2nd Law of Commutativity

Let us assume that we are exploring the sub-domain at index ι: $\langle\Delta\rangle\hat{}\ell\hat{}\langle\mathsf{A}\rangle$. Whether we first "discover" $\mathcal{A}$ttributes and then $\mathcal{M}$ereology (including $\mathcal{U}$nique identifiers) or first "discover" $\mathcal{M}$ereology (including $\mathcal{U}$nique identifiers) and then $\mathcal{A}$ttributes should not matter. We make some abbreviations: $\mathcal{A}$ stand for the ATTRIBUTES, $\mathcal{U}$ stand for the UNIQUE_IDENTIFIER, $\mathcal{M}$ stand for the MEREOLOGY, ι for index $\langle\Delta\rangle\hat{}\ell\hat{}\langle\mathsf{A}\rangle$, and $\iota\mathsf{s}$ for a suitable set of indices. Thus we wish the following law to hold:

121. $(\mathcal{A}(\iota);\mathcal{U}(\iota);\mathcal{M}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{U}(\iota);\mathcal{M}(\iota)(\iota s);\mathcal{A}(\iota))(\Re) \simeq$
$(\mathcal{U}(\iota);\mathcal{A}(\iota);\mathcal{M}(\iota)(\iota s))(\Re)$.

here modulo attribute and unique identifier type name renaming.

11.6.3 3rd Law of Commutativity

Let us again assume that we are exploring the sub-domain at index ι: $\langle\Delta\rangle\hat{}\ell\hat{}\langle\mathsf{A}\rangle$ where $\iota\mathsf{s}$ is a suitable set of indices. Whether we are exploring actions, events or behaviours at that domain index in that order, or some other order ought be immaterial. Hence with $\mathcal{A}$ now standing for the ACTION_SIGNATURES, $\mathcal{E}$ standing for the EVENT_SIGNATURES, $\mathcal{B}$ standing for the BEHAVIOUR_SIGNATURES, discoverers, we wish the following law to hold:

122. $(\mathcal{A}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{A}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{E}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{E}(\iota)(\iota s);\mathcal{B}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{B}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s))(\Re) \simeq$
$(\mathcal{B}(\iota)(\iota s);\mathcal{E}(\iota)(\iota s);\mathcal{A}(\iota)(\iota s))(\Re)$.

here modulo action function, event predicate, channel, message type and behaviour (and all associated, auxiliary type) renamings.

11.6.4 1st Law of Stability

Re-performing the same discovery function over the same sub-domain, that is with identical indices, one or more times, ought not produce any new description texts. That is:

123. $(\mathcal{D}(\iota)(\iota\mathsf{s});\mathcal{A}\_\mathsf{and}\_\mathcal{D}\_\mathsf{seq})(\Re) \simeq (\mathcal{D}(\iota)(\iota\mathsf{s});\mathcal{A}\_\mathsf{and}\_\mathcal{D}\_\mathsf{seq};\mathcal{D}(\iota)(\iota\mathsf{s}))(\Re)$

where $\mathcal{D}$ is any discovery function, $\mathcal{A}\_\mathsf{and}\_\mathcal{D}\_\mathsf{seq}$ is any specific sequence of intermediate analyses and discoveries, and where ι and $\iota\mathsf{s}$ are suitable indices, respectively sets of indices.

11.6.5 2nd Law of Stability

367

Re-performing the same analysis functions over the same sub-domain, that is with identical indices, one or more times, ought not produce any new analysis results. That is:

124. $[\mathcal{A}(\iota)] = [\mathcal{A}(\iota);\ldots;\mathcal{A}(\iota)]$

where $\mathcal{A}$ is any analysis function, "..." is any sequence of intermediate analyses and discoveries, and where ι is any suitable index.

11.6.6 Law of Non-interference

368

When performing a discovery meta-operation, $\mathcal{D}$ on any index, ι, and possibly index set, $\iota\mathsf{s}$, and on a repository state, $\Re$, then using the $[\mathcal{D}(\iota)(\iota\mathsf{s})]$ notation expresses a pair of a narrative text and some formulas, [txt,rsl], whereas using the $(\mathcal{D}(\iota)(\iota\mathsf{s}))(\Re)$ notation expresses a next repository state, $\Re'$. What is the "difference" ? Informally and simplifying we can say that the relation between the two expressions is:

125. $[\mathcal{D}(\iota)(\iota\mathsf{s})]$: [txt,rsl]
$(\mathcal{D}(\iota)(\iota\mathsf{s}))(\Re) = \Re'$
where $\Re' = \Re \cup \{[\text{txt,rsl}]\}$

369

We say that when 125. is satisfied for any discovery meta-function $\mathcal{D}$, for any indices ι and $\iota\mathsf{s}$ and for any repository state $\Re$, then the repository is not interfered with, that is, *"what you see is what you get:"* and therefore that the discovery process satisfies the law on non-interference.

11.7 Discussion

370

The above is just a hint at domain development laws that we might wish orderly developments to satisfy. We invite the reader to suggest other laws.

The laws of the analysis and discovery calculus forms an ideal set of expectations that we have of not only one domain describer but from a domain describer team of two or more domain describers whom we expect to work, i.e., loosely collaborate, based on "near"-identical domain development principles.

371

These are quite some expectations. But the whole point of a highest-level academic scientific education and engineering training is that one should expect commensurate development results.

372

Now, since the ingenuity and creativity in the analysis and discovery process does differ between domain developers we expect that a daily process of *"buddy checking"*, where individual team members present their findings and where these are discussed by the team will result in adherence to the laws of the calculus.

The laws of the analysis and discovery calculus expressed some properties that we wish the repository to exhibit. We have deliberately abstained from "over-defining" the structure of repositories and the "hidden" operations (i.e., 'update', etc.) repositories. We expect further research into, development of, possible changes to and use of the calculus to yield such insight as to lead to a firmer understanding of the nature of repositories.

373

374

In the analysis and discovery calculus such as we have presented it we have emphasised the types of parts, sorts and immediate part concrete types, and the signatures of actions, events and behaviours — as these predominantly featured type expressions. We have therefore, in this tutorial, not investigated, for example, pre/post conditions of action function, form of event predicates, or behaviour process expressions. We leave that, substantially more demanding issue, for future explorative and experimental research.

375

376

377

Part IV

Requirements Engineering

12 Requirements Engineering 378

We shall present a terse overview of how one can "derive" essential fragments of requirements prescriptions from a domain description. First we give, in the next section, a summary of the net domain, N, as developed in Sects. 2 and 11. We refer to Examples 8, 13, 29, 47– 53.

12.1 The Transport Domain — a Resumé 379

12.1.1 Nets, Hubs and Links

126. From a transport net one can observe sets of hubs and links.

type
126. N, HS, Hs = H-**set**, H, LS, Ls = L-**set**, L
127. HI, LI
15. LΣ = HI-**set**, HΣ = (LI×LI)-**set**
16. LΩ = LΣ-**set**, HΩ = HΣ-**set**

value
126. obs_HS: N → HS, obs_LS: N → LS
126. obs_Hs: N → H-**set**, obs_Ls: N → L-**set**
15. attr_LΣ: L → LΣ, attr_HΣ: H → HΣ
16. attr_LΩ: L → LΩ, attr_HΩ: H → HΩ

12.1.2 Mereology 380

127. From hubs and links one can observe their unique hub, respectively link identifiers and their respective mereologies.

128. The mereology of a link identifies exactly two distinct hubs.

129. The mereologies of hubs and links must identify actual links and hubs of the net.

381

value
127. uid_H: H → HI, uid_L: L → LI
127. mereo_H: H → LI-**set**, mereo_L: L → HI-**set**

axiom
128. ∀ l:L•**card** mereo_L(l)=2
129. ∀ n:N,l:L•l ∈ obs_Ls(n) ⇒
129. ∧ ∀ hi:HI•hi ∈ mereo_L(l)
129. ⇒ ∃ h:h•h ∈ obs_Hs(n)∧uid_H(h)=hi
129. ∧ ∀ h:H•h ∈ obs_Hs(n) ⇒
129. ∀ li:LI•li ∈ mereo_H(h)
129. ⇒ ∃ l:L•l ∈ obs_Ls(n)∧uid_L(l)=li

12.2 A Requirements "Derivation" 382

12.2.1 Definition of Requirements

IEEE Definition of 'Requirements'

By a requirements we understand (cf. IEEE Standard 610.12 [56]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*.

12.2.2 The Machine = Hardware + Software 383

By 'the machine' we shall understand the software to be developed and hardware (equipment + base software) to be configured for the domain application.

12.2.3 Requirements Prescription 384

The core part of the requirements engineering of a computing application is the requirements prescription. A requirements prescription tells us which parts of the domain are to be supported by 'the machine'. A requirements is to satisfy some goals. Usually the goals cannot be prescribed in such a manner that they can serve directly as a basis for software design. Instead we derive the requirements from the domain descriptions and then argue (incl. prove) that the goals satisfy the requirements. In this paper we shall not show the latter but shall show the former.

12.2.4 Some Requirements Principles 385

The "Golden Rule" of Requirements Engineering

Prescribe only such requirements that can be objectively shown to hold for the designed software.

An "Ideal Rule" of Requirements Engineering

When prescribing (including formalising) requirements, also formulate tests (theorems, properties for model checking) whose actualisation should show adherence to the requirements.

We shall not show adherence to the above rules.

12.2.5 A Decomposition of Requirements Prescription 386

We consider three forms of requirements prescription: the domain requirements, the interface requirements and the machine requirements. Recall that the machine is the hardware and software (to be required). Domain requirements are those whose technical terms are from the domain only. Machine requirements are those whose technical terms are from the machine only. Interface requirements are those whose technical terms are from both.

12.2.6 An Aside on Our Example 387

We shall continue our "ongoing" example. Our requirements is for a tollway system. By a requirements goal we mean *"an objective the system under consideration should achieve"* [94]. The goals of having a tollway system are: to decrease transport times between selected hubs of a general net; and to decrease traffic accidents and fatalities while moving on the tollway net as compared to comparable movements on the general net. The tollway net, however, must be paid for by its 388
users. Therefore tollway net entries and exits occur at tollway plazas with these plazas containing entry and exit toll collectors where tickets can be issued, respectively collected and travel paid for. We shall very briefly touch upon these toll collectors, in the Extension part (as from Page 79) of the next section, Sect. 12.3 on the next page. So all the other parts of the next section serve to build up to the Extension section, Sect. 12.3.4 on page 79.

12.3 Domain Requirements

Domain requirements cover all those aspects of the domain — parts and materials, actions, events and behaviours — which are to be supported by 'the machine'. Thus domain requirements are developed by systematically "revising" cum "editing" the domain description: which parts are to be **projected:** left in or out; which general descriptions are to be **instantiated** into more specific ones; which non-deterministic properties are to be made more **determinate**; and which parts are to be **extended** with such computable domain description parts which are not feasible without IT.

Thus projection, instantiation, determination and extension are the basic engineering tasks of domain requirements engineering. An example may best illustrate what is at stake. The example is that of a tollway system — in contrast to the general nets covered by description Items 126–129 (Pages 74–74). See Fig. 4.

The links of the general net of Fig. 4 are all two-way links, so are the plaza-to-tollway links of the tollway net of Fig. 4. The tollway links are all one-way links. The hubs of the general net of Fig. 4 are assumed to all allow traffic to move in from any link and onto any link. The plaza hubs do not show links to "an outside" — but they are assumed. Vehicles enter the tollway system from the outside and leave to the outside. The tollway hubs allow traffic to move in from the plaza-to-tollway link and back onto that or onto the one or two tollway links emanating from that hub, as well as from tollway links incident upon that hub onto tollway links emanating from that hub or onto the tollway-to-plaza link.

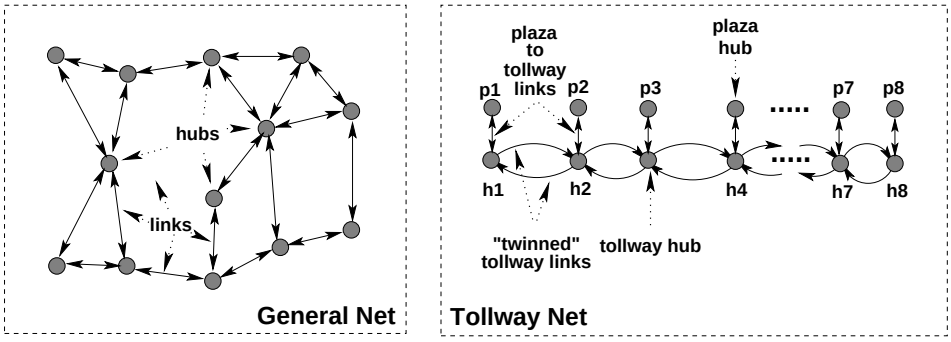


Figure 4: General and Tollway Nets

12.3.1 Projection

We keep what is needed to prescribe the tollway system and leave out the rest.

130. We keep the description, narrative and formalisation,
    a nets, hubs, links,
    b hub and link identifiers,
    c hub and link states,
131. as well as related observer functions.

type
130a. N, H, L
130b. HI, LI
130c. HΣ, LΣ
value
131. obs_Hs,obs_Ls,obs_HI,obs_LI,
131. obs_HIs,obs_LIs,obs_HΣ,obs_L Σ

We omit bringing the composite part concepts of HS, LS, Hs and Ls into the requirements.

12.3.2 Instantiation

From the general net model of earlier formalisations we instantiate, that is, make more concrete, the tollway net model now described.

132. The net is now concretely modelled as a pair of sequences.
133. One sequence models the plaza hubs, their plaza-to-tollway link and the connected tollway hub.
134. The other sequence models the pairs of "twinned" tollway links.
135. From plaza hubs one can observe their hubs and the identifiers of these hubs.
136. The former sequence is of m such plaza "complexes" where $m \geq 2$; the latter sequence is of $m-1$ "twinned" links.
137. From a tollway net one can abstract a proper net.
138. One can show that the posited abstraction function yields well-formed nets, i.e., nets which satisfy previously stated axioms.

397

```
type
132. TWN = PC* × TL*
133. PC = PH × L × H
134. TL = L × L
value
133. obs_H: PH → H, obs_HI: PH → HI
axiom
136. ∀ (pcl,tll):TWN •
136.   2≤len pcl∧len pcl=len tll+1
value
137. abs_N: TWN → N
137. abs_N(pcl,tll) as n
137.   pre: wf_TWN(pcl,tll)
137.   post:
137.     obs_Hs(n) =
137.       {h,h′|(h,_,h′):PC
137.       •(h,_,h′)∈ elems pcl}
137.   ∧ obs_Ls(n) =
137.       {l|(_,l,_):PC
137.         •(_,l,_)∈ elems pcl} ∪
137.       {l,l′|(l,l′):TL•(l,l′)∈ elems tll}

theorem:
138. ∀ twn:TWN • wf_TWN(twn)
138.     ⇒ wf_N(abs_N(twn))
```

398

[1] Model Well-formedness wrt. Instantiation: Instantiation restricts general nets to tollway nets. Well-formedness deals with proper mereology: that observed identifier references are proper. The well-formedness of instantiation of the tollway system model can be defined as follows:

139. The i'plaza complex, (p_i, l_i, h_i), is instantiation-well-formed if
    a. link l_i identifies hubs p_i and h_i, and
    b. hub p_i and hub h_i both identifies link l_i; and if
140. the i'th pair of twinned links, tl_i, tl'_i,
    a. has these links identify the tollway hubs of the i'th and $i+1$'st plaza complexes ((p_i, l_i, h_i) respectively $(p_{i+1}, l_{i+1}, h_{i_1})$).

399

```
value
  Instantiation_wf_TWN: TWN → Bool
  Instantiation_wf_TWN(pcl,tll) ≡
139.   ∀ i:Nat • i ∈ inds pcl⇒
139.     let (pi,li,hi)=pcl(i) in
139a.      obs_LIs(li)={obs_HI(pi),obs_HI(hi)}
139b.    ∧ obs_LI(li)∈ obs_LIs(pi)∩ obs_LIs(hi)
140.   ∧ let (li′,li″) = tll(i) in
140.       i < len pcl ⇒
140.         let (pi′,li‴,hi′) = pcl(i+1) in
140a.          obs_HIs(li) = obs_HIs(li′)
140a.            = {obs_HI(hi),obs_HI(hi′)}
      end end end
```

12.3.3 Determination

Determination, in this example, fixes states of hubs and links. The state sets contain only one set. Twinned tollway links allow traffic only in opposite directions. Plaza to tollway hubs allow traffic in both directions. tollway hubs allow traffic to flow freely from plaza to tollway links and from incoming tollway links to outgoing tollway links and tollway to plaza links. The determination-well-formedness of the tollway system model can be defined as follows[51]:

[1] Model Well-formedness wrt. Determination: We need define well-formedness wrt. determination. Please study Fig. 5.

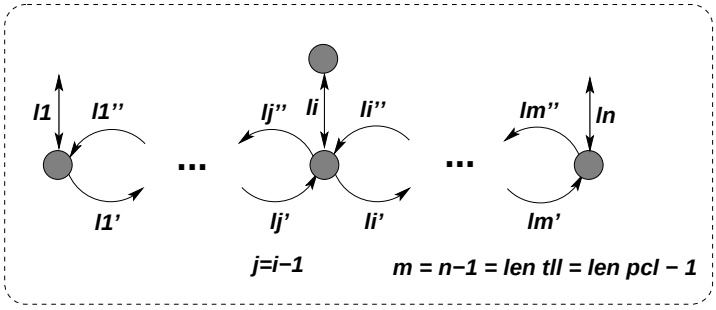


Figure 5: Hubs and Links

141. All hub and link state spaces contain just one hub, respectively link state.
142. The i'th plaza complex, pcl(i):(p_i, l_i, h_i) is determination-well-formed if
    a l_i is open for traffic in both directions and
    b p_i allows traffic from h_i to "revert"; and if
143. the i'th pair of twinned links (li', li'') (in the context of the i+1st plaza complex, pcl(i+1):$(p_{i+1}, l_{i+1}, h_{i+1})$) are determination-well-formed if
    a link l'_i is open only from h_i to h_{i+1} and
    b link l''_i is open only from h_{i+1} to h_i; and if
144. the jth tollway hub, h_j (for $1 \leq j \leq \mathbf{len}$ pcl) is determination-well-formed if, depending on whether j is the first, or the last, or any "in-between" plaza complex positions,
    a [the first:] hub $i = 1$ allows traffic in from l_1 and l''_1, and onto l_1 and l'_1.
    b [the last:] hub $j = i + 1 = \mathbf{len}$ pcl allows traffic in from $l_{\mathbf{len}\,\mathsf{tll}}$ and $l''_{\mathbf{len}\,\mathsf{tll}-1}$, and onto $l_{\mathbf{len}\,\mathsf{tll}}$ and $l'_{\mathbf{len}\,\mathsf{tll}-1}$.
    c [in-between:] hub $j = i$ allows traffic in from l_i, l''_i and l'_i and onto l_i, l'_{i-1} and l''_i.

```
value
142. Determination_wf_TWN: TWN → Bool
142. Determination_wf_TWN(pcl,tll) ≡
142.   ∀ i:Nat• i ∈ inds tll ⇒
142.     let (pi,li,hi) = pcl(i),
142.         (npi,nli,nhi) = pcl(i+1), in
142.         (li′,li″) = tll(i) in
141.       obs_HΩ(pi)={obs_HΣ(pi)}∧obs_HΩ(hi)={obs_HΣ(hi)}
141.     ∧ obs_LΩ(li)={obs_LΣ(li)}∧obs_LΩ(li′)={obs_LΣ(li′)}
```

[51] i ranges over the length of the sequences of twinned tollway links, that is, one less than the length of the sequences of plaza complexes. This "discrepancy" is reflected in out having to basically repeat formalisation of both Items 142a and 142b.

```
141.   ∧ obs_LΩ(li″)={obs_LΣ(li″)}
142a.  ∧ obs_LΣ(li)
142a.      = {(obs_HI(pi),obs_HI(hi)),(obs_HI(hi),obs_HI(pi))}
142a.  ∧ obs_LΣ(nli)
142a.      = {(obs_HI(npi),obs_HI(nhi)),(obs_HI(nhi),obs_HI(npi))}
142b.  ∧ {(obs_LI(li),obs_LI(li))}⊆obs_HΣ(pi)
142b.  ∧ {(obs_LI(nli),obs_LI(nli))}⊆obs_HΣ(npi)
143a.  ∧ obs_LΣ(li′)={(obs_HI(hi),obs_HI(nhi))}
143b.  ∧ obs_LΣ(li″)={(obs_HI(nhi),obs_HI(hi))}
144.   ∧ case i+1 of
144a.     2 → obs_HΣ(h_1)=
144a.           {(obs_LΣ(l_1),obs_LΣ(l_1)), (obs_LΣ(l_1),obs_LΣ(l_1″)),
144a.            (obs_LΣ(l″_1),obs_LΣ(l_1)), (obs_LΣ(l″_1),obs_LΣ(l′_1))},
144b.     len pcl → obs_HΣ(h_i+1)=
144b.           {(obs_LΣ(l_len pcl),obs_LΣ(l_len pcl)),
144b.            (obs_LΣ(l_len pcl),obs_LΣ(l′_len tll)),
144b.            (obs_LΣ(l″_len tll),obs_LΣ(l_len pcl)),
144b.            (obs_LΣ(l″_len tll),obs_LΣ(l′_len tll))},
144c.     _ → obs_HΣ(h_i)=
144c.           {(obs_LΣ(l_i),obs_LΣ(l_i)), (obs_LΣ(l_i),obs_LΣ(l′_i)),
144c.            (obs_LΣ(l_i),obs_LΣ(l″_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i)),
144c.            (obs_LΣ(l″_i),obs_LΣ(l′_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i))}
142.   end end
```

12.3.4 Extension

404

By domain extension we understand the *introduction of domain entities, actions, events and behaviours that were not feasible in the original domain, but for which, with computing and communication, there is the possibility of feasible implementations, and such that what is introduced become part of the emerging domain requirements prescription.* 405

[1] Narrative: The domain extension is that of the controlled access of vehicles to and departure from the tollway net: the entry to (and departure from) tollgates from (respectively to) an `"an external"` net — which we do not describe; the new entities of tollgates with all their machinery; the user/machine functions: upon entry: driver pressing entry button, tollgate delivering ticket; upon exit: driver presenting ticket, tollgate requesting payment, driver providing payment, etc. 406

One added (extended) domain requirements: as vehicles are allowed to cruise the entire net payment is a function of the totality of links traversed, possibly multiple times. This requires, in our case, that tickets be made such as to be sensed somewhat remotely, and that hubs be equipped with sensors which can record and transmit information about vehicle hub crossings. (When exiting, the tollgate machine can then access the exiting vehicles' sequence of hub crossings — based on which a payment fee calculation can be done.) All this to be described in detail — including all the things that can go wrong (in the domain) and how drivers and tollgates are expected to react. 407 408

We omit details of narration and formalisation. In this case the extension description would entail a number of formalisations: An initial one which relies significantly on the use of `RSL/CSP` [55, 7]. It basically models tollbooth and vehicle behaviours. A "derived" one which models temporal properties. It is expressed, for example, in the `Duration Calculus, DC` [100]. And finally a timed-automata [2, 71] model which "implements" the `DC` model.

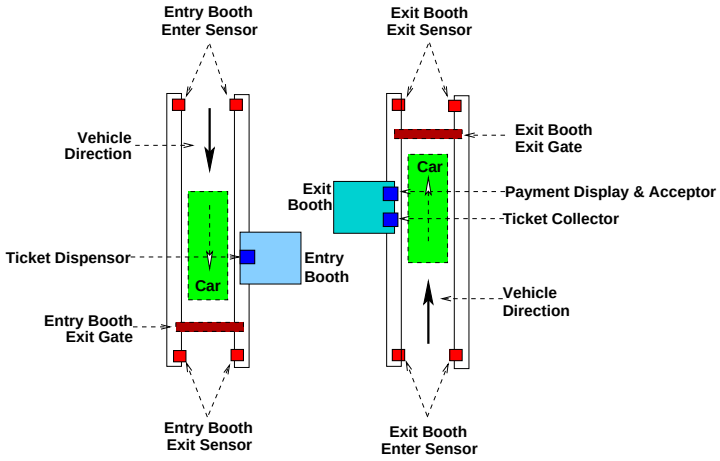


Figure 6: Entry and Exit Tollbooths

12.4 Interface Requirements Prescription

409

A systematic reading of the domain requirements shall result in an identification of all shared parts and materials, actions, events and behaviours. An entity is said to be a shared entity if it is mentioned in both the domain description and the requirements prescription. That is, if the entity is present in the domain and is to be present in the machine.

410

Each such shared phenomenon shall then be individually dealt with: **part** and **materials sharing** shall lead to interface requirements for **data initialisation and refreshment; action sharing** shall lead to interface requirements for **interactive dialogues between the machine and its environment; event sharing** shall lead to interface requirements for **how events are communicated between the environment of the machine and the machine. behaviour sharing** shall lead to interface requirements for **action and event dialogues between the machine and its environment.**

• • •

We shall now illustrate these domain interface requirements development steps with respect to our ongoing example.

12.4.1 Shared Parts

411

The main shared parts of the main example of this section are the net, hence the hubs and the links. As domain parts they repeatedly undergo changes with respect to the values of a great number of attributes and otherwise possess attributes — most of which have not been mentioned so far: length, cadestral information, namings, wear and tear (where-ever applicable), last/next scheduled maintenance (where-ever applicable), state and state space, and many others.

412

We "split" our interface requirements development into two separate steps: the development of $d_{r.net}$ (the common domain requirements for the shared hubs and links), and the co-development of $d_{r.db:i/f}$ (the common domain requirements for the interface between $d_{r.net}$ and DB_{rel} — under the assumption of an available relational database system DB_{rel} When planning the common domain requirements for the net, i.e., the hubs and links, we enlarge our scope of requirements concerns beyond the two so far treated ($d_{r.toll}$, $d_{r.maint.}$) in order to make sure that the shared relational database of nets, their hubs and links, may be useful beyond those requirements. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modelling

413

414

effort it must be secured that "standard" actions on nets, hubs and links can be supported by the chosen relational database system DB_{rel}. 415

[1] Data Initialisation: As part of $d_{r.net}$ one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes (names) and their types and, for example, two for the input of hub and link attribute values. Interaction prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

Essentially these prescriptions concretise the insert link action. 416

[2] Data Refreshment: As part of $d_{r.net}$ one must also prescribe data refreshment: an interactive user interface dialogue with a set of proper display screens one for updating net, hub or link attributes (names) and their types and, for example, two for the update of hub and link attribute values. Interaction prompts may be prescribed: next update, on-line vetting and display of revised net, etc. These and many other aspects may therefore need prescriptions.

These prescriptions concretise remove and insert link actions.

12.4.2 Shared Actions 417

The main shared actions are related to the entry of a vehicle into the tollway system and the exit of a vehicle from the tollway system.

[1] Interactive Action Execution: As part of $d_{r.toll}$ we must therefore prescribe the varieties of successful and less successful sequences of interactions between vehicles (or their drivers) and the toll gate machines.

The prescription of the above necessitates determination of a number of external events, see below.

(Again, this is an area of embedded, real-time safety-critical system prescription.)

12.4.3 Shared Events 418

The main shared external events are related to the entry of a vehicle into the tollway system, the crossing of a vehicle through a tollway hub and the exit of a vehicle from the tollway system.

As part of $d_{r.toll}$ we must therefore prescribe the varieties of these events, the failure of all appropriate sensors and the failure of related controllers: gate opener and closer (with sensors and actuators), ticket "emitter" and "reader" (with sensors and actuators), etcetera.

The prescription of the above necessitates extensive fault analysis.

12.4.4 Shared Behaviours 419

The main shared behaviours are therefore related to the journey of a vehicle through the tollway system and the functioning of a toll gate machine during "its lifetime". Others can be thought of, but are omitted here.

In consequence of considering, for example, the journey of a vehicle behaviour, we may "add" some further, extended requirements: (a) requirements for a vehicle statistics "package"; (b) requirements for tracing supposedly "lost" vehicles; (c) requirements limiting tollway system access in case of traffic congestion; etcetera.

12.5 Machine Requirements 420

The machine requirements make hardly any concrete reference to the domain description; so we omit its treatment altogether.

12.6 Discussion of Requirements "Derivation" 421

We have indicated how the domain engineer and the requirements engineer can work together to 422 "derive" significant fragments of a requirements prescription. This puts requirements engineering in a new light. Without a previously existing domain descriptions the requirements engineer has to do double work: both domain engineering and requirements engineering but without the principles of domain description, as laid down in this tutorial that job would not be so straightforward as we 423 now suggest. 424

Part V
Closing

13 Conclusion 425

This document, meant as the basis for my tutorial at FM 2012 (CNAM, Paris, August 28), "grew" from a paper being written for possible journal publication. Sections 2–11 possibly represent two 426 publishable journal papers. Section 12 has been "added" to the 'tutorial' notes. The style of the two tutorial "parts", Sects. 2–11 and Sect. 12 are, necessarily, different: Sects. 2–11 are in the form of research notes, whereas Sect. 12 is in the form of "lecture notes" on methodology. Be that as it may. Just so that you are properly notified !

13.1 Comparison to Other Work 427

In this section we shall only compare our contribution to domain engineering as presented in Sect. 2 to that found in the broader literature with respect to the software engineering term 'domain'. We shall not compare[52] our contribution to requirements engineering as surveyed in Sect. 12. to that, also, found in the broader requirements engineering literature. Finally we shall also not compare our work on a description calculus as we find no comparable literature!

13.1.1 Ontological Engineering: 428

Ontological engineering is described mostly on the Internet, see however [6]. Ontology engineers build ontologies. And ontologies are, in the tradition of ontological engineering, *"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 groups building upon one-anothers work and processed by various 429 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 or makes reference to its reliance of an upper ontology. Instead the ontology databases appear to be used for the 430 computerised discovery and analysis of relations between ontologies.

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.

[52] As these tutorial notes primarily focus on domain science & engineering we shall postpone a proper literature comparison till we have factored Sect. 12 out from these notes and into a proper paper in which a proper comparison must be made.

13.1.2 Knowledge and Knowledge Engineering:

The concept of knowledge has occupied philosophers since Plato. No common agreement on what 'knowledge' is has been reached. From Wikipedia we may learn that *knowledge is a familiarity with someone or something; it can include facts, information, descriptions, or skills acquired through experience or education; it can refer to the theoretical or practical understanding of a subject; knowledge is produced by socio-cognitive aggregates (mainly humans) and is structured according to our understanding of how human reasoning and logic works.*

The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [40]: knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. Knowledge engineering focuses 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 [26].

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.

Further references to seminal exposés of knowledge engineering are [89, 64].

13.1.3 Domain Analysis:

There are different "schools of domain analysis". Domain analysis, or product line analysis (see below), as it was first 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-Dïaz's approach [77, 78, 79]. Firstly, the two meanings of domain analysis basically coincide. Secondly, in, for example, [77], Prieto-Dïaz's domain analysis is focused on the very important stages that precede the kind of domain modelling that we have described:major concerns areselection 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-Dïaz. 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-Dïaz'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-Dïaz 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 the `TripTych` approach to domain analysis we provide a full ontology — cf. Sects. 2–10 and suggest a domain description calculus. In our approach we do not consider requirements, let alone software components, as do Prieto-Dïaz, but we find that that is not an important issue.

13.1.4 Software Product Line Engineering:

439

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
440 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 [16] puts the ideas of software product lines and model-oriented software development in the context of the TripTych approach. Notable sources on software product line engineering are [5, 96, 3, 90, 51, 84, 29, 31, 36, 31, 74].

13.1.5 Problem Frames:

441

The concept of problem frames is covered in [61]. Jackson's prescription for software development focuses 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
442 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", well, iterating between these rôles repeatedly. So, perhaps belabouring the point, domain engineering is done only to the extent needed by the prescription of requirements
443 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 [59] 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 the TripTych, "more general", approach. There are a number of aspects of software development that we have not treated in this tutorial. They have to do with software verification and validation. These aspects are covered in [49, 59].

13.1.6 Domain Specific Software Architectures (DSSA):

444

It seems that the concept of DSSA was formulated by a group of ARPA[53] 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 [92]. The [92] definition of domain engineering is *"the process of creating a DSSA: domain analysis and domain modelling followed by*
445 *creating a software architecture and populating it with software components."* This definition is basically followed also by [70, 85, 69]. Defined and pursued this way, DSSA appears, notably in these latter references, to start with the with the analysis of software components, "per domain", to identify commonalities within application software, and to then base the idea of software ar-
446 chitecture 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,
447 whether already existing, or being subsequently developed. Of course, given a domain descriptions 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 ! So that puts a "damper" of my "enthusiasm" for DSSA.

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

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.

13.1.7 Domain Driven Design (DDD)

Domain-driven design (DDD)[54] *"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."*[55] We have studied some of the DDD literature, mostly only accessible on The Internet, but see also [52], 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.

13.1.8 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 [63] 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 [34]. Since FODA "leans" quite heavily on 'Software Product Line Engineering' our remarks in that section, above, apply equally well here.

13.1.9 Unified Modelling Language (UML)

Three books representative of UML are [28, 82, 62]. 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 [3,4,6,7,8], with either software design (as it most often is), or requirements prescription. Certainly, in UML, in [28, 82, 62] 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.

There is, however, some important notions of UML and that is the notions of class diagrams, objects, etc. How these notions relate to the discovery of part types, unique part identifiers, mereology and attributes, as well as action, event and behaviour signatures and channels, as discovered at a particular domain index, is not yet clear to me. That there must be some relation seems obvious. We leave that as an interesting, but not too difficult, research topic.

13.1.10 Requirements Engineering:

There are in-numerous books and published papers on requirements engineering. A seminal one is [95]. I, myself, find [66] full of very useful, non-trivial insight. [38] is seminal in that it brings a number or early contributions and views on requirements engineering. Conventional text books, notably [73, 76, 87] all have their "mandatory", yet conventional coverage of requirements engineering. None of them "derive" requirements from domain descriptions, yes, OK, from domains,

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

but since their description is not mandated it is unclear what "the domain" is. Most of them repeatedly refer to domain analysis but since a written record of that domain analysis is not mandated it is unclear what "domain analysis" really amounts to. Axel van Laamsweerde's book [95] is remarkable. Although also it does not mandate descriptions of domains it is quite precise as to the relationships between domains and requirements. Besides, it has a fine treatment of the distinction between goals and requirements, also formally. Most of the advices given in [66] can beneficially be followed also in TripTych requirements development. Neither [95] nor [66] preempts TripTych requirements development.

13.1.11 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, [49, 59], and the present document, being a good starting point.

13.2 What Have We Achieved and Future Work

Sect. 13.1 has already touched upon, or implied, a number of 'achievement' points and issues for future work. Here is a summary of 'achievement' and future work items.

We claim that there are three major contributions being reported upon: (i) the separation of domain engineering from requirements engineering, (ii) the separate treatment of domain science & engineering: as "free-standing" with respect, ultimately, to computer science, and endowed with quite a number of domain analysis principles and domain description principles; and (iii) the identification of a number of techniques for "deriving" significant fragments of requirements prescriptions from domain descriptions — where we consider this whole relation between domain engineering and requirements engineering to be novel.

Yes, we really do consider the possibility of a systematic 'derivation' of significant fragments of requirements prescriptions from domain descriptions to cast a different light on requirements engineering.

What we have not shown in this tutorial is the concept of domain facets; this concept is dealt with in [12] — but more work has to be done to give a firm theoretical understanding of domain facets of domain intrinsics, domain support technology, domain scripts, domain rules and regulations, domain management and organisation, and human domainbehaviour.

13.3 General Remarks

Perhaps belaboring the point: one can pursue creating and studying domain descriptions without subsequently aiming at requirements development, let alone software design. That is, domain descriptions can be seen as "free-standing", of their "own right", useful in simply just understanding domains in which humans act. Just like it is deemed useful that we study "Mother Nature", the physical world around us, given before humans "arrived"; so we think that there should be concerted efforts to study and create domain models, for use in studying "our man-made domains of discourses"; possibly proving laws about these domains; teaching, from early on, in middle-school, the domains in which the middle-school students are to be surrounded by; etcetera

How far must one formalise such domain descriptions ? Well, enough, so that possible laws can be mathematically proved. Recall that domain descriptions usually will or must be developed by domain researchers — not necessarily domain engineers — in research centres, say universities, where one also studies physics. And, when we base requirements development on domain descriptions, as we indeed advocate, then the requirements engineers must understand the formal domain descriptions, that is, be able to perform formal domain projection, domain instantiation, domain determination, domain extension, etcetera. This is similar to the situation in classical engineering which rely on the sciences of physics, and where, for example, *Bernoulli's equations*, *Navier-Stokes equations*,

Maxwell's equations, etcetera were developed by physicists and mathematicians, but are used, daily, by engineers: read and understood, massaged into further differential equations, etcetera, in order to calculate (predict, determine values), etc. Nobody would hire non-skilled labour for the engineering development of airplane designs unless that "labourer" was skilled in *Navier-Stokes equations*, or for the design of mobile telephony transmission towers unless that person was skilled in *Maxwell's equations*.

So we must expect a future, we predict, where a subset of the software engineering candidates from universities are highly skilled in the development of formal domain descriptions formal requirements prescriptions in at least one domain, such as *transportation*, for example, air traffic, railway systems, road traffic and shipping; or *manufacturing*, *services* (health care, public administration, etc.), *financial industries*, or the like.

13.4 Acknowledgements

I thank the tutorial organisers of the FM 2012 event for accepting my Dec. 31. 2011 tutorial proposal. I thank that part of participants who first met up for this tutorial this morning (Tuesday 28 August, 2012) to have remained in this room for most, if not all of the time. I thank colleagues and PhD students around Europe for having listened to previous, somewhat less polished versions of this tutorial. I in particular thank Drs. **Magne Haveraaen** and **Marc Bezem** of the University of Bergen for providing an important step in the development of the present material. And I thank my wife for her patience during the spring and summer of 2012 where I ought to have been tending to the garden, etc. !

14 Bibliographical Notes

14.1 The Notes

14.1.1 Domains: Methodology and Theory Papers

There are now a number of published papers on domain science & engineering from the point of view put forward in this report. In the reverse order of publication (latest first) we begin with a bracketed pair: [#,`file.pdf`] where # refers to the References below and `file.pdf` is to be prefixed by `http://www.imm.dtu.dk/~dibj/` in order to create an appropriate access `URL`.

- [17, `urbino-p.pdf`] examines the relationship between the philosophical and logic notion of mereology and model-oriented treatments of parts (as here: pipeline nets and units).
- [18, `human-p.pdf`] examines possible bases for the research into and development of IT systems that, when developed from sound models of domains, may meet a number of 'humanity'-related criteria.
- [16, `maurer-bjorner.pdf`] examines domain models as the basis for the development of domain demos, domain simulators, and domain monitors & controllers. The present paper, in its quest for domain demos etc., relies on the observation made in [16].
- [13, 15, `kiev-p1.pdf, kiev-p2.pdf`] In the first of these references we methodically unravel a domain description, showing most facets of domain engineering. In the second of these papers we report on an emerging theory of domain descriptions.
- [22, `db-psi09-paper.pdf`] This paper posits that requirements engineering pursued without a prior phase of domain engineering from which significant parts of the requirements can be systematically "derived" may be the wrong way to develop software, cf. the next reference.
- [11, `montanari.pdf`] This paper outlines some of the stages and steps of domain engineering and some of the stages and steps of requirements engineering — all around a common example.

- [14, bsm.pdf] This paper reviews the practical software engineering management notions of process assessment and process improvement as these notions are to be interpreted in the context of domain **and** requirements engineering.
- [23, wpdr.pdf] This paper, like [17], provides some insight into possible theoretical foundations for domain descriptions.
- [10, ictac-paper.pdf] This paper reviews salient aspects of domain engineering and posits a number of research topics suggested for further study.
- [12, facs-domain.pdf] This paper overviews stages and steps of domain engineering.

14.1.2 Experimental and Explorative Domain Descriptions

We list a number of URLs to reports that tackle individual domain descriptions. Except for the last entry below, the prefix http://www.imm.dtu.dk/~dibj/ should be put in front of the teletype identifiers given below in order to retrieve these reports from the Internet.

- Financial Service Industry (fsi.pdf) This 170+ page, incomplete draft report sketches various aspects of banking and securities trading.
- Stock Exchanges (tse-1.pdf) This 74 page report overlaps with the above but specifically models the Tokyo Stock Exchange Trading Rules.
- Container Line Industry (container-paper.pdf) This almost 100 page draft report models a comprehensive set of a container line (like, f.ex. the Danish Maersk Lines, http://www.-maersk.com).
- Logistics (logistics.pdf) This draft document models some of the aspects of logistics.
- The Market: Consumers, Retailers, Wholesaler and Producers (themarket.pdf) This published paper models "a supply-chain market".
- Window Systems, Web and Transaction Protocols (wfdftp.pdf) This incomplete report models what it (might) mean[s] to be a window system (of recursively defined windows [icons may reveal windows]), an extended SQL system, and how window editing may involve a number of [say Internet] shared "window"-like databases being updated according to the Two-phase Commit Protocol.
- License Languages (license-languages.pdf) This paper (appearing first as [19] in [21]) models, as script languages, the commands required for (i) the downloading, editing, sublicensing, etc. of electronic media; (ii) the creating, editing, copying, reading and trashing of public administration documents; and (iii) the computerised support of patient hospitalisaton whose actions involve anamnese, analysis, diagnostics, creation of treatment plans, treatments, etc.
- IT Security (it-security.pdf) This paper (appearing first as [20] in [21]) suggests a model for the concept of IT Security Management as described in ISO Standard ISO/IEC 17799.
- The Railway Domain (http://www.railwaydomain.org/) This Web page was established around 2003. It was intended as a focal point for academic R&D efforts in one or another aspect of formal software development related to one or another kind of railway and train software. Interest in this "focal point", unfortunately, never really "took off".

  We refer to Toward a TRain Book (http://www.railwaydomain.org/PDF/tb.pdf) as illustrating fragments of a domain model for railways systems.

14.1.3 Lecture-oriented Notes on Domain Engineering

There are a number of Internet-based sets of lecture note slides on Domain Science & Engineering. We refer to:

- 2012: Towards a Theory of Domain Descriptions – a gentle introduction: `tadt-s.pdf`
- 2009: From Domains to Requirement. The Triptych Approach to Software Engineering: `de-re--s.pdf`

14.1.4 Textbooks on RAISE/RSL and Software Engineering

- [44] The definitive reference to RSL, the RAISE Specification Language,
- [45] The first comprehensive treatment of the RAISE Method by the creators of RAISE.
- [7, 8, 9] A comprehensive treatment of abstraction and modelling, languages and systems, domain and requirements engineering and of software design. [7] (additionally) covers RSL, [8] (additionally) covers various forms of Petri Nets[56] [81], MSC (Message Sequence Charts) [57], Statecharts [50], and DC: Duration Calculus [100].

14.1.5 Other Textbooks on Formal Methods

The ideas of formal domain descriptions are independent of the specific formal specification languages being used. Below we list some other model-oriented formal (specification and development) methods.

- [58] A delightful introduction to a delightful 'Abstraction, Modelling & Verification' using the Alloy specification language.
- [1] Seminal introductions to B and Event B by the creator of these two languages.
- [100] The standard reference work on the continuous time, interval temporal Duration Calculus logic by two of its most prominent researchers and developers.
- [65] The standard reference work on Temporal Logic of Actions by the creator of TLA+.
- [99] A standard reference to the Z formal specification language[57] and its use.

14.2 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] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. (Preliminary versions appeared in Proc. 17th ICALP, LNCS 443, 1990, and Real Time: Theory in Practice, LNCS 600, 1991).

[3] M. Ardis, N. Daley, D. Hoffman, H. Siy, and D. Weiss. Software product lines: a case study. *Software: Practice and Experience*, 2000.

[4] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. The Cambridge University Press, (1967) 2005.

[5] J. Bayer, J.-M. DeBaud, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, and T. Widen. Pulse: Amethodology to develop software product lines. In *Symposium on Software Reusability*, volume SSR'99, pages 122–131, May 1999.

[56] Basic Petri Nets, Place-Transition Nets, Coloured Petri Nets

[57] Z was created by the creator also of B and Event B: Jean-Raymond Abrial.

[6] V. Benjamins and D. Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems , note =, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany.

[7] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .

[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. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[15] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[16] D. Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[17] D. Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, 2012.

[18] D. Bjørner. *Domain Science and Engineering as a Foundation for Computation for Humanity*. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC [Francis & Taylor], 2012. (eds.: Justyna Zander and Pieter J. Mosterman).

[19] D. Bjørner. *[21] Chap. 10: Towards a Family of Script Languages – – Licenses and Contracts – Incomplete Sketch*, pages 283–328. JAIST Press, March 2009.

[20] D. Bjørner. *[21] Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.

[21] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph (# 4), March 2009. .

[22] D. Bjørner. The Role of Domain Engineering in Software Development. Why Current Requirements Engineering Seems Flawed! In *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 2–34, Heidelberg, Wednesday, January 27, 2010. Springer.

[23] D. Bjørner and A. Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

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

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

[26] D. Bjørner and J. F. Nilsson.

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

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

[29] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.

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

[31] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[32] N. Cocchiarella. Formal Ontology. In H. Burkhardt and B. Smith, editors, *Handbook in Metaphysics and Ontology*, pages 640–647. Philosophia Verlag, Munich, Germany, 1991.

[33] CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.

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

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

[36] R. de Almeida Falbo, G. Guizzardi, and K. C. Duarte. An Ontological Approach to Domain Engineering. *International Conference on Software Engineering and Knowledge Engineering, SEKE'02, Ischia, Italy*, 2002.

[37] H. Delugach. Common Logic Standard. ISO Standard ISO/IEC IS 24707:2007, ISO, 2007.

[38] M. Dorfman and R. H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.

[39] E. Evans and B. Coulbeck, editors. *Pipeline Systems*. Number 7 in Fluid Mechanics and Its Applications. Kluwer Academic Publishers, 2012.

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

[41] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, Second edition, 2009.

[42] K. Futatsugi, A. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211, NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.

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

[44] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[45] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[46] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993.

[47] N. Guarino. Formal Ontology, Conceptual Analysis and Knowledge Representation. *Intl. Journal of Human–Computer Studies*, 43:625–640, 1995.

[48] N. Guarino. Some Organising Principles for a Unified Top–level Ontology. Int.rept., Italian National Research Council (CNR), LADSEB–CNR, Corso Stati Uniti 4, I–35127 Padova, Italy. guarino@ladseb.pd.cnr.it, 1997.

[49] 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.

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

[51] M. Harsu. A survey on domain engineering. , note =, Institute of Software Systems, Tampere University of Technology.

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

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

[54] T. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.

[55] T. Hoare. Communicating Sequential Processes. Published electronically: `http://www.-usingcsp.com/cspbook.pdf`, 2004. Second edition of [54]. See also `http://www.usingcsp.-com/`.

[56] IEEE Computer Society. IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical report, IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614, 1990.

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

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

[59] 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.

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

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

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

[63] 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, note =, Software Engineering Institute, Carnegie Mellon University.

[64] S. Kendal and M. Green. *An introduction to knowledge engineering*. Springer, London, 2007.

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

[66] S. Lauesen. *Software Requirements - Styles and Techniques*. Addison-Wesley, UK, 2002.

[67] J. McCarthy and et al. *LISP 1.5, Programmer's Manual*. The MIT Press, Cambridge, Mass., USA, 1962.

[68] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [75].

[69] 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.

[70] E. Mettala and M. H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, month =, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213.

[71] E.-R. Olderog and H. Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, UK, 2008.

[72] OWL Working Group W3C. OWL 2 Web Ontology Language Document Overview. W3C Recommendation http://www.w3.org/TR/owl2-overview/, month =, The World Wide Web Consortium.

[73] S. L. Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.

[74] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.

[75] R. L. Poidevin and M. MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[76] R. S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.

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

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

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

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

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

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

[83] B. Russell. *The Principles of Mathematics*. Cambridge University Press, 1903. (Allen & Unwin, London, 1937).

[84] K. Schmid. Scoping Software Product Lines. In *Software Product Lines: Experience and Research Directions*. Kluwer Academic Press, 2000.

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

[86] B. Smith. Ontology and the Logistic Analysis of Reality. In N. Guarino and R. Poli, editors, *International Workshop on Formal Ontology in Conceptual Analysis and Knowledge Representation*. Institute for Systems Theory and Biomedical Engineering of the Italian National Research Council, Padua, Italy, 1993. Revised version in G. Haefliger and P. M. Simons (eds.), Analytic Phenomenology, Dordrecht/Boston/London: Kluwer.

[87] I. Sommerville. *Software Engineering*. Pearson, 8th edition, 2006.

[88] S. Staab and R. Stuber, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, Heidelberg, 2004.

[89] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge Engineering: Principles and Methods. *Data & Knowledge Engineering*, 25:161–197, 1998.

[90] S. Thiel and F. Peruzzi. Starting a product line approach for an envisioned market. In *Software Product Lines, Experience and Research Directions*. Kluwer Academic Press, 2000.

[91] A. T. D. Thorley. *Fluid Transients in Pipeline Systems*. The American Society of Mechanical Engineers. ASME Press, New Yorm NY, USA, 2nd edition edition, (1991) 2004.

[92] W. Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 1994.

[93] 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.

[94] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *5th IEEE International Symposium of Requirements Engineering*, volume RE'01, pages 249–263, Toronto, Canada, August 2001. IEEE CS Press.

[95] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[96] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software-Development Process*. Addison–Wesley, 1999.

[97] J. F. Wendt, editor. *Computational Fluid Dynamics, An Introduction*. von Karman Institute Books. Springer, 3rd edition edition, (1992) 2009.

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

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

[100] 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.

Part VI
Appendices

A A TripTych Ontology

A Domain Description Ontology

B On A Theory of Transport Nets 472

This section is under development. The idea of this section is not so much to present a transport domain description, but rather to present fragments, "bits and pieces", of a theory of such a domain. The purpose of having a theory is to "draw" upon the 'bits and pieces' when expressing properties of endurants and definitions of actions, events and behaviours. Again: this section is very much in embryo.

B.1 Some Pictures 473

Nets can either be rail nets, road nets, shipping lanes, or air traffic nets. The following pictures illustrate some of these nets.

A rail net; a traffic light

The traffic light (above, on the right) appears to be able to let any combination of incoming links 474 pass any combination of outgoing links.

A freeway hub

From any incoming link one can pass through this hub to any outgoing link without red traffic 475 lights.

Another freeway hub

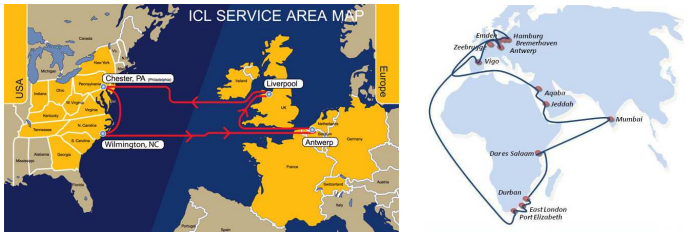
476

The left side of the road roundabout below is rather special. Its traffic lights are also located in the inner circle of the roundabout. One drives in, at green light, and may be guided by striping, depending on where one is driving, either directly to an outgoing link, or is queued up against a 477 red light awaiting permission to continue.

A roundabout

478

The map below left is for a container line serving one route between Liverpool (UK), Chester (PA, USA), Wilmington (NC, USA) and Antwerp (Belgium), an so forth, circularly. The map below right is an "around Africa" Mitsui O.S.K. Line.



Two shipping line nets

B.2 Parts

479

B.2.1 Nets, Hubs and Links

148. From a transport net one can observe sets of hubs and links.

type
148. N, H, L
value
148. obs_Hs: N → H-**set**, obs_Ls: N → L-**set**

B.2.2 Mereology

480

149. From hubs and links one can observe their unique hub, respectively link identifiers and their respective mereologies.

150. The mereology of a link identifies exactly two distinct hubs.

151. The mereologies of hubs and links must identify actual links and hubs of the net.

type
149. HI, LI
value
149. uid_H: H → HI, uid_L: L → LI
149. mereo_H: H → LI-**set**, mereo_L: L → HI-**set**
axiom
150. ∀ l:L•**card** mereo_L(l)=2
151. ∀ n:N,l:L•l ∈ obs_Ls(n) ⇒
151. ∧ ∀ hi:HI•hi ∈ mereo_L(l)
151. ⇒ ∃ h:h•h ∈ obs_Hs(n)∧uid_H(h)=hi

151. ∧ ∀ h:H•h ∈ obs_Hs(n) ⇒
151. ∀ li:LI•li ∈ mereo_H(h)
151. ⇒ ∃ l:L•l ∈ obs_Ls(n)∧uid_L(l)=li

B.2.3 An Auxiliary Function

481

152. For every net we can define functions which
    a extracts all its link identifiers,
    b and all its hub identifiers.

value
152a. xtr_HIs: N → HI-**set**
152a. xtr_HIs(n) ≡ {uid_H(h)|h:H•h ∈ obs_Hs(n)}
152b. xtr_LIs: N → LI-**set**
152b. xtr_LIs(n) ≡ {uid_L(l)|l:L•l ∈ obs_Ls(n)}

B.2.4 Retrieving Hubs and Links

482

153. We can also define functions which
    a given a net and a hub identifier obtains the designated hub, respectively
    b given a net and a link identifier obtains the designated link.

value
153a. get_H: N → HI $\xrightarrow{\sim}$ H
153a. get_H(n)(hi) **as** h
153a. **pre** hi ∈ xtr_HIs(n)
153a. **post** h ∈ obs_Hs(n)∧hi=uid_H(h)
153b. get_L: N → LI $\xrightarrow{\sim}$ L
153b. **pre** li ∈ xtr_LIs(n)
153b. **post** l ∈ obs_Ls(n)∧li=uid_L(l)

B.2.5 Invariants over Link and Hub States and State Spaces

483

154. Links include two attributes:
    a Link states. These are sets of pairs of the identifiers of the hubs to which the links are connected.
    b Link state spaces. These are the sets of link states that a link may attain.
155. The link states must mention only those hub identifiers of the two hubs to which the link is connected.
156. The link state spaces must likewise mention only such link states as are defined in Items 154a and 155.

484

type
154a. LΣ = (HI×HI)-**set axiom** ∀ lσ:LΣ•**card** lσ≤2
154b. LΩ = LΣ-**set**
value
154a. attr_LΣ: L → LΣ

```
154b. attr_LΩ: L → LΩ
axiom
155. ∀ l:L, lσ′:LΣ • lσ′ ∈ attr_LΩ(l)
155.       ⇒ lσ′ ⊆ {(hi,hi′)|hi,hi′:HI•{hi,hi′}⊆mereo_L(l)}
155.         ∧ attr_LΣ(l) ∈ attr_LΩ(l)
```

485

157. Hubs include two attributes:
    a Hub states. These are sets of pairs of identifiers of the links to which the hubs are connected.
    b Hub state spaces. These are the sets of hub states that a hub may attain.
158. The hub states must mention only those link identifiers of the links to which the hub is connected.
159. The hub state spaces must likewise mention only such hub states as are defined in Items 157a and 158.

486

```
type
157a. HΣ = (LI×LI)-set
157b. HΩ = HΣ-set
value
157a. attr_HΣ: H → HΣ
157b. attr_HΩ: H → HΩ
axiom
158. ∀ h:H, hσ′:HΣ • hσ′ ∈ attr_HΩ(h)
158.       ⇒ hσ′ ⊆ {(li,li′)|li,li′:LI•{li,li′}⊆mereo_H(h)}
158.         ∧ attr_HΣ(h) ∈ attr_HΩ(h)
```

B.2.6 Maps

487

A map is an abstraction of a net. The map just shows the hub and link identifiers of the net, and hence its mereology.

```
type
    Map′ = HI ~m~> (LI ~m~> HI)
    Map = {|m:Map′•wf_Map(m)|}
value
    wf_Map: Map′ → Bool
    wf_Map(m) ≡ dom m = ∪ { rng lhm | lhm:(LI ~m~> HI) • lhm ∈ rng m }
```

488

Let m be a map. The *definition set* of the map is domṁ. Let hi be in the definition set of map m. Then m(hi) is the *image* of hi in m. Let li be in the image of m(hi), that is, liISINdom (m(hi)), then hi'=(m(hi))(li) is the *target* of li in m(hi).

489

Given a net which satisfies the axiom concerning mereology one can extract from that net a corresponding map.

```
value
    xtr_Map: N → Map
    xtr_Map(n) ≡
        [ hi ↦ [ li ↦ uid_H(retr_H(n)(hi)(li))
                 | li:LI • li ∈ mere_H(get_H(n)(hi)) ]
          | h:H,hi:HI • h ∈ obs_Hs(n) ∧ hi = uid_H(h) ]
```

490

The retrieve hub function retrieve the "second" hub, i.e., "at the other end", of a link wrt. a "first" hub.

```
retr_H: N → HI → LI → H
retr_H(n)(hi)(li) ≡
    let h = get_H(n)(hi) in
    let l = get_L(n)(li) in
    let {hi‴} = mereo_L(l)\{hi} in
    get_H(n)(hi‴) end end end
    pre: hi ∈ mereo_L(get_L(n)(li))

xtr_LIs: Map → LI-set
xtr_LIs(m) = ∪ {dom(m(hi))|hi:HI • hi ∈ dom m}
```

B.2.7 Routes

491

160. A route is an alternating sequence of hub and link identifiers.

```
160. R′ = (HI|LI)^ω, R = {|r:R′•wf_R(r)|}
value
160. wf_R: R′ → Bool
160. wf_R(r) ≡
160.     ∀ i:Nat • {i,i+1}⊆inds r ⇒
160.         is_HI(r(i))∧is_LI(r(i+1)) ∨ is_LI(r(i))∧is_HI(r(i+1))
```

492

161. A route of a map, m, is a route as follows:
    a An empty sequence is a route.
    b A sequence of just a single hub identifier or of hubs of the map is a route.
    c A sequence of just a single link identifier of links of the map is a route.
    d If $r^\langle hi\rangle$ and $\langle li\rangle^r'$ are routes of the map and li is in the definition set of m(hi) then $r^\langle hi,li\rangle^r'$ is a route of the map.
    e If $r^\langle li\rangle$ and $\langle hi\rangle^r'$ are routes of the map and hi is the target of (m(hi′))(li) then $r^\langle li,hi\rangle^r'$ is a route of the map.
    f Only such routes are routes of a net if they result from a finite [possibly infinite] set of uses of Items 161a-161e.

493

```
type
type
161. MR′ = R, MR = {r:MR′•∃ m:Map • r ∈ routes(m)|}
value
161. routes: N → MR-infset
161. routes(n) ≡ routes(xtr_Map(n))
161. routes: Map → MR-infset
161. routes(m) ≡
161a.    let rs = {⟨⟩}
161b.      ∪ ∪ {⟨hi⟩|hi:HI • hi ∈ dom m}
161c.      ∪ ∪ {⟨li⟩|li:LI,hi:HI • li ∈ xtr_LIs(m)}
161d.      ∪ ∪ {r^⟨hi,li⟩^r′|r,r′:MR,hi:HI,li:LI•{r,r′}⊆rs∧li ∈ dom m(hi)}
161e.      ∪ ∪ {r^⟨li,hi⟩^tl r′|r,r′:MR,li:LI,hi:HI•{r,r′}⊆rs∧is_target(m)(hi)(li)}
```

161f. **in** rs **end**

161e. is_target: Map → HI × LI
161e. is_target(m)(hi)(li) ≡
161e. ∃ h″:HI•h″∈ **dom** m∧li ∈ **dom** m(hi″)∧hi=(m(hi″))(li)

B.2.8 Special Routes

494

[1] Acyclic Routes

162. A route of a map is acyclic if no hub identifier appears twice or more.

value
162. is_Acyclic: MR → Map $\xrightarrow{\sim}$ **Bool**
162. is_Acyclic(mr)(m) ≡ ∼∃ hi:HI,i,j:**Nat**•{i,j}⊆**inds** mr ∧ i≠j ⇒ mr(i)=hi=mr(j)
162. **pre** mr ∈ routes(m)

[2] Direct Routes

163. A route, r, of a map (from hub hi or linkli to hub hi′ or linkli′) is a direct route if r is acyclic.

163. direct_route: MR → Map $\xrightarrow{\sim}$ **Bool**
163. direct_route(mr) ≡ is_Acyclic(mr)
163. **pre** mr ∈ routes(m)

495

[3] Routes Between Hubs

164. Let there be given two distinct hub identifiers of a route map. Find the set of acyclic routes between them, including zero if no routes.

value
164. find_MR: Map → (HI×HI) $\xrightarrow{\sim}$ MR-**set**
164. find_MR(m)(hi,hi′) ≡
164. **let** rs = routes(m) **in**
164. {mr | mr,mr′:MR • mr ∈ rs
164. ∧ mr ∈ mr = ⟨hi⟩⌢mr′⟨hi′⟩ ∧ is_Acyclic(mr)(m) }
164. **end**
164. **pre**: {hi,hi′}⊆**dom** m

B.2.9 Special Maps

496

[1] Isolated Hubs

165. A net, n, consists of two or more isolated hubs

a if there exists two hub identifiers, hi_1,hi_2, of the map of the net
b such that there is no route from hi_1 to hi_2.

value
165. are_isolated_hubs: Map → **Bool**
165. are_isolated_hubs(m) ≡
165a. ∃ hi_1,hi_2:HI • {hi_1,hi_2}⊆**dom** m ⇒
165b. ∼∃ mr,mr_i:MR • mr ∈ routes(m) ⇒ mr = ⟨hi_1⟩⌢mr_i⌢⟨hi_2⟩

[2] Isolated Maps

166. If there are isolated hubs in a net then the net can be seen as two or more isolated nets.

value
166. are_isolated_nets: Map → **Bool**
166. are_isolated_nets(m) ≡ are_isolated_hubs(m)

497

[3] Sub_Maps

167. Given a map one can identify the set of all sub_maps which which contains a given hub identifier.

168. Given a map one can identify the sub_map which contains a given hub identifier.

value
167. sub_maps: Map → Map-**set**
167. sub_maps(m) **as** ms
167. { xtr_Map(m)(hi) | hi:HI • hi ∈ **dom** m }

168. sub_Map: Map → HI $\xrightarrow{\sim}$ Map
168. sub_Map(m)(hi) ≡
168. **let** his = { hi′ | hi′:HI ∧ hi′ ∈ **dom** m ∧ find_MRs(m)(hi,hi′)≠{} } **in**
168. [hi″ ↦ m(hi″) | hi″ ∈ his] **end**

theorem: are_isolated_nets(m) ⇒ sub_maps(m) ≠ { m }

B.3 Actions

498

B.3.1 Insert Hub

169. The insert action
    a applies to a net and a hub and conditionally yields an updated net.
    b The condition is that there must not be a hub in the initial net with the same unique hub identifier as that of the hub to be inserted and
    c the hub to be inserted does not initially designate links with which it is to be connected.
    d The updated net contains all the hubs of the initial net "plus" the new hub.
    e and the same links.

499

value
169. insert_H: N → H $\xrightarrow{\sim}$ N
169a. insert_H(n)(h) **as** n′
169a. **pre**: pre_insert_H(n)(h)
169a. **post**: post_insert_H(n)(h)(n′)

169b. pre_insert_H(n)(h) ≡
169b. ∼∃ h′:H • h′ ∈ obs_Hs(n) ∧ uid_H(h)=uid_H(h′)
169c. ∧ mereo_H(h) = {}

169d. post_insert_H(n)(h)(n′) ≡
169d. obs_Hs(n) ∪ {h} = obs_Hs(n′)
169e. ∧ obs_Ls(n) = obs_Ls(n′)

B.3.2 Insert Link

500

170. The insert link action
    a is given a "fresh" link, that is, one not in the net (before the action)
    b but where the two distinct hub identifiers of the mereology of the inserted link are of hubs in the net.
    c The link is inserted.
    d These two hubs
    e have their mereologies updated to reflect the new link
    f and nothing else; all other links and hubs of the net are unchanged.

```
value
170. insert_L: N → L ~→ N
170. insert_L(n)(l) as n′
170.     ∃ l:L • pre_insert_L(n)(l) ⇒ pre_insert_L(n)(l) ∧ post_insert_L(n,n′)(l)
```

501

```
170. pre_insert_L: N → L → Bool
170. pre_insert_L(n)(l) ≡
170a.     uid_L(l) ∉ xtr_Lls(n)
170b.   ∧ mereo_L(l)⊆xtr_Hls(n)

170. post_insert_L: N × N → L → Bool
170. post_insert_L(n,n′)(l) ≡
170c.     obs_Ls(n) ∪ {l} = obs_Ls(n′)
170d.   ∧ let {hi1,hi2} = mereo_L(l) in
170d.     let (h1,h2) = (get_H(n)(hi1),get_H(n)(hi2)),
170d.         (h1′,h2′) = (get_H(n′)(hi1),get_H(n′)(hi2)) in
170e.     mereo_H(h)∪{uid_L(l)}=mereo_H(h′)
170f.   ∧ obs_Hs(n)\{h1,h2} = obs_Hs(n′)\{h1′,h2′}
170f.   ∧ [ all other properties of h1 and h2 unchanged ]
170f.     [ that is, same as h1′ and h2′                ]
170.    end end
```

502

The insert link post-condition has too many lines. I will instead compose the post-condition from the conjunction of a number of invocations of predicates with "telling" names. For these action function definitions such "small" predicates amount to building a nicer theory.

B.3.3 Remove Hub

503

171. remove hub
    a where a hub, known by its hub identifier, is given,
    b where the [to be] removed hub is indeed in the net (before the action),
    c where the removed hub's mereology is empty (that is, the [to be] removed hub) is not connected to any links in the net (before the action)).
    d All other links and hubs of the net are unchanged.

```
value
171.  remove_H: N → HI ~→ N
171a. remove_H(n)(hi) as n′
171b.     ∃ h:H • uid_H(h)=hi ∧ h ∈ obs_Hs(n) ⇒
171c.       pre_remove_H(n)(hi) ∧ post_remove_H(n,n′)(hi)
```

We leave the definitions of the pre/post conditions of this and the next action function to the reader.

B.3.4 Remove Link 504

172. remove link

a where a link, known by its link identifier, is given,

b where that link is indeed in the net (before the action),

c where hubs to which the link is connected after the action has the only change to their mereologies changed be that they do not list the [to be] removed link.

d All other links and hubs of the net are unchanged.

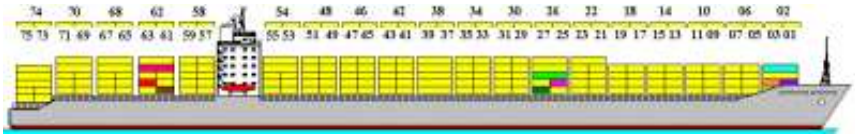
value

172. remove_L: N → LI $\xrightarrow{\sim}$ N
172a. remove_L(n)(li) **as** n′
172b. ∃ l:L • uid_L(l)=li ∧ l ∈ obs_Ls(n) ⇒
172c. pre_remove_L(n)(li) ∧ post_remove_L(n,n′)(li)

C On A Theory of Container Stowage 505

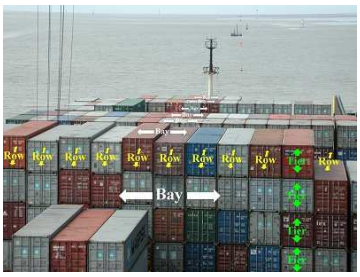
This section is under development. The idea of this section is not so much to present a container domain description, but rather to present fragments, "bits and pieces", of a theory of such a domain. The purpose of having a theory is to "draw" upon the 'bits and pieces' when expressing properties of endurants and definitions of actions, events and behaviours. Again: this section is very much in embryo.

C.1 Some Pictures 506

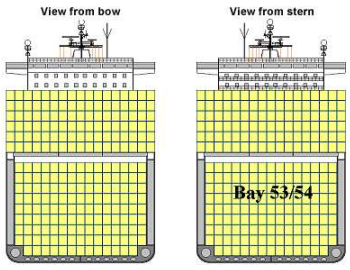


A container vessel with 'bay' numbering

Container vessels ply the seven seas and in-numerous other waters. They carry containers from port to port. The history of containers[58] goes back to the late 1930s. The first container vessels made their first transports in 1956. Malcolm P. McLean is credited to have invented the container. To prove the concept of container transport he founded the container line `Sea-Land Inc.` which was sold to `Maersk Lines` at the end of the 1990s. 507



Bay numbers.



Ship stowage cross section

Down along the vessel, horisontally, from front to aft, containers are grouped, in numbered bays. 508

Row and tier numbers

Bays are composed from rows, horisontally, across the vessel. Rows are composed from stacks, horisontally, along the vessel. And stacks are composed, vertically, from [tiers of] containers

C.2 Parts 509

C.2.1 A Basis

173. From a container vessel (cv:CV) and from a container terminal port (ctp:CTP) one can observe their bays (bays:BAYS).

[58] http://www.containerhandbuch.de/chb_e/stra/index.html?/chb_e/stra/stra_01_01_00.html

type
173. CV, CTP, BAYS
value
173. obs_BAYS: (CV|CTP) → BAYS

510

174. The bays, bs:BS, (of a container vessel or a container terminal port) are mereologically structured as an (BId) indexed set of individual bays (b:B).

type
174. BId, B
174. BS = BId $\xrightarrow[m]{}$ B
value
174. obs_BS: BAYS → BS (i.e., BId $\xrightarrow[m]{}$ B)

511

175. From a bay, b:B, one can observe its rows, rs:ROWS.
176. The rows, rs:RS, (of a bay) are mereologically structured as an (RId) indexed set of individual rows (r:R).

type
175. ROWS, RId, R
176. RS = RId $\xrightarrow[m]{}$ R
value
175. obs_ROWS: B → ROWS
176. obs_RS: ROWS → RS (i.e., RId $\xrightarrow[m]{}$ R)

512

177. From a row, r:R, one can observe its stacks, STACKS.
178. The stacks, ss:SS (of a row) are mereologically structured as an (SId) indexed set of individual stacks (s:S).

type
177. STACKS, SId, S
178. SS = SId $\xrightarrow[m]{}$ S
value
177. obs_STACKS: R → STACKS
178. obs_SS: STACKS → SS (i.e., SId $\xrightarrow[m]{}$ S)

513

179. A stack (s:S) is mereologically structured as a linear sequence of containers (c:C).

type
179. C
179. S = C*

The containers of the same stack index across stacks are called the tier at that index, cf. photo on
514 Page 107..

180. A container is here considered a composite part
    a of the container box, k:K
    b and freight, f:F.
181. Freight is considered composite

a and consists of zero, one or more colli (package, indivisible unit of freight),
b each having a unique colli identifier (over all colli of the entire world!).
c Container boxes likewise have unique container identifiers.

515

type
180. C, K, F, P
value
180a. obs_K: C → K
180b. obs_F: C → F
181a. obs_Ps: F → P-**set**
type
181b. PI
181c. CI
value
181b. uid_P: P → PI
181c. uid_C: C → CI

C.2.2 Mereological Constraints

516

182. For any bay of a vessel the index sets of its rows are identical.

183. For a bay of a vessel the index sets of its stacks are identical.

axiom
182. ∀ cv:CV •
182. ∀ b:B•b ∈ **rng** obs_BS(obs_BAYS(cv))⇒
182. **let** rws=obs_ROWS(b) **in**
182. ∀ r,r′:R•{r,r′}⊆**rng** obs_RS(b)⇒**dom** r=**dom** r′
183. ∧ **dom** obs_SS(r) = **dom** obs_SS(r′) **end**

C.2.3 Stack Indexes

517

184. A container stack (and a container) is designated by an index triple: a bay index, a row index and a stack index.

185. A container index triple is valid, for a vessel, if its indices are valid indices.

type
184. StackId = BId×RId×SId
value
185. valid_address: BS → StackId → **Bool**
185. valid_address(bs)(bid,rid,sid) ≡
185. bid ∈ **dom** bs
185. ∧ rid ∈ **dom** (obs_RS(bs))(bid)
185. ∧ sid ∈ **dom** (obs_SS((obs_RS(bs))(bid)))(rid)

518

The above can be defined in terms of the below.

type
BayId = BId
RowId = BId×RId
value
185. valid_BayId: V → BayId → **Bool**

185. valid_BayId(v)(bid) ≡ bid ∈ **dom** obs_BS(obs_BAYS(v))

185. get_B: V → BayId $\xrightarrow{\sim}$ B
185. get_B(v)(bid) ≡ (get_B(bs))(bid) **pre**: valid_BId(v)(bid)

185. get_B: BS → BayId $\xrightarrow{\sim}$ B
185. get_B(bs)(bid) ≡ (obs_BS(obs_BAYS(v)))(bid) **pre**: bid ∈ **dom** bs

519

185. valid_RowId: V → RowId → **Bool**
185. valid_RowId(v)(bid,rid) ≡ rid ∈ **dom** obs_RS(get_B(v)(bid))
185. **pre**: valid_BayId(v)(bid)

185. get_R: V → RowId $\xrightarrow{\sim}$ R
185. get_R(v)(bid,rid) ≡ get_R(obs_BS(v))(bid,rid) **pre**: valid_RowId(v)(bid,rid)

185. get_R: BS → RowId $\xrightarrow{\sim}$ R
185. get_R(bs)(bid,rid) ≡ (obs_RS(get_RS(bs(bid))))(rid)
185. **pre**: valid_RowId(v)(bid,rid)

520

185. get_S: V → StackId $\xrightarrow{\sim}$ S
185. get_S(v)(bid,rid,sid) ≡ (obs_SS(get_R(get_B(v)(bid,rid))))(sid)
185. **pre**: valid_address(v)(bid,rid,sid)

521

185. get_C: V → StackId $\xrightarrow{\sim}$ C
185. get_C(v)(stid) ≡ get_C(obs_BS(v))(stid) **pre**: get_S(v)(bid,rid,sid) ≠ ⟨⟩

185. get_C: BS → StackId $\xrightarrow{\sim}$ C
185. get_C(bs)(bid,rid,sid) ≡ **hd**(obs_SS(get_R((bs(bid))(rid))))(sid)
185. **pre**: get_S(bs)(bid,rid,sid) ≠ ⟨⟩

185. valid_addresses: V → StackId-**set**
185. valid_addresses(v) ≡ {adr|adr:StackId•valid_address(adr)(v)}

522

186. The predicate non_empty_designated_stack checks whether the designated stack is non-empty.

186. non_empty_designated_stack: V → StackId → **Bool**
186. non_empty_designated_stack(v)(bid,rid,sid) ≡ get_S(v)(bid,rid,sid) ≠ ⟨⟩

523

187. Two vessels have the same mereology if they have the same set of valid-addresses.

value
187. unchanged_mereology: BS × BS → **Bool**
187. unchanged_mereology(bs,bs′) ≡ valid_addresses(bs) = valid_addresses(bs′)

524

188. The designated stack, s′, of a vessel, v′ is popped with respect the "same designated" stack, s, of a vessel, v
    a. if the ordered sequence of the containers of s′ are identical to the ordered sequence of containers of all but the first container of s.

188. popped_designated_stack: BS × BS → StackId → **Bool**
188. popped_designated_stack(bs,bs′)(stid) ≡
188a. **tl** get_S(v)(stid) = get_S(bs′)(stid)

525

189. For a given stack index, valid for two bays (bs, bs′) of two vessels or two container terminal ports, and say stid, these two bays enjoy the unchanged_non_designated_stacks(bs,bs′)(stid) property

    a if the stacks (of the two bays) not identified by stid are identical.

189. unchanged_non_designated_stacks: BS × BS → StackId → **Bool**
189. unchanged_non_designated_stacks(bs,bs′)(stid) ≡
189a. ∀ adr:StackId•adr ∈ valid_addresses(v)\{stid}⇒
189a. get_S(bs)(adr) = get_S(bs′)(adr)
189. **pre**: unchanged_mereology(bs,bs′)

C.2.4 Stowage Schemas

526

190. By a stowage schema of a vessel we understand a "table"

    a which for every bay identifier of that vessel records a bay schema
    b which for every row identifier of an identified bay records a row schema
    c which for every stack identifier of an identified row records a stack schema
    d which for every identified stack records its tier schema.
    e A stack schema records for every tier index (which is a natural number) the type of container (contents) that may be stowed at that position.
    f The tier indexes of a stack schema form a set of natural numbers from one to the maximum number in the index set.[59]

527

value
190. obs_StoSchema: V → StoSchema
type
190a. StoSchema = BId $\overrightarrow{m}$ BaySchema
190b. BaySchema = RId $\overrightarrow{m}$ RowSchema
190c. RowSchema = SId $\overrightarrow{m}$ StaSchema
190d. StaSchema = **Nat** $\overrightarrow{m}$ C_Type
190e. C_Type
axiom
190f. ∀ stsc:StaSchema • **dom** stsc = {1..**max dom** stsc}

528

191. One can define a function which from an actual vessel "derives" its "current stowage schema".

191. cur_sto_schema: V → StoSchema
191. cur_sto_schema(v) ≡
191. **let** bs = obs_BS(obs_BAYS(v)) **in**
191. [bid ↦ **let** rws = obs_RS(obs_ROWS(bs(bid))) **in**
191. [rid ↦ **let** ss = obs_SS(obs_STACKS(rws)(rid)) **in**
191. [sid ↦ ⟨ analyse_container(ss(i))|i:**Nat**•i ∈ **inds** ss ⟩

[59] That maximum number designates the maximum height of the stack at that stack position. For any actual stack the height is between zero and the maximum height, inclusive.

```
191.                     | sid:SId•sid ∈ ss ] end
191.               | rid:RId•rid ∈ dom rws ] end
191.      | bid:BId•bid ∈ dom ds ] end

191. analyse_container: C → C_Type
```

529

192. Given a stowage schema and a current stowage schema one can check the latter for conformance wrt. the former.

```
192. conformance: StoSchema × StoSchema → Bool
192. conformance(stosch,cur_stosch) ≡
192.   dom cur_stosch = dom stosch
192. ∧ ∀ bid:BId • bid ∈ dom stosch ⇒
192.     dom cur_stosch(bid) = dom stosch(bid)
192.   ∧ ∀ rid:RId • rid ∈ dom(stosch(bid))(rid) ⇒
192.       dom(cur_stosch(bid))(rid) = dom(stosch(bid))(rid)
192.     ∧ ∀ sid:SId • sid ∈ dom(cur_stosch(bid))(rid)
192.         ∀ i:Nat • i ∈ inds((cur_stosch(bid))(rid))(sid) ⇒
192.           conform((((cur_stosch(bid))(rid))(sid))(i),
192.                   (((stosch(bid))(rid))(sid))(i))

192. conform: C_Type × C_Type → Bool
```

530

193. From a vessel one can observe its mandated stowage schema.

194. The current stowage schema of a vessel must always conform to its mandated stowage schema.

```
value
193.  obs_StoSchema: V → StoSchema

194.  stowage_conformance: V → Bool
194.  stowage_conformance(v) ≡
194.    let mandated = obs_StoSchema(v),
194.        current = cur_sto_schema(v) in
194.    conformance(mandated,current) end
```

C.3 Actions

531

C.3.1 Remove Container from Vessel

20. The remove_Container_from_Vessel action applies to a vessel and a stack address and conditionally yields an updated vessel and a container.

    20a. We express the 'remove from vessel' function primarily by means of an auxiliary function remove_C_from_BS, remove_C_from_BS(obs_BS(v))(stid), and some further post-condition on the before and after vessel states (cf. Item 20d).

    20b. The remove_C_from_BS function yields a pair: an updated set of bays and a container.

    20c. When obs_erving the BayS from the updated vessel, v′, and pairing that with what is assumed to be a vessel, then one shall obtain the result of remove_C_from_BS(obs_BS(v))(stid).

    20d. Updating, by means of remove_C_from_BS(obs_BS(v))(stid), the bays of a vessel must leave all other properties of the vessel unchanged.

532

21. The pre-condition for remove_C_from_BS(bs)(stid) is
    21a. that stid is a valid_address in bs, and
    21b. that the stack in bs designated by stid is non_empty.
22. The post-condition for remove_C_from_BS(bs)(stid) wrt. the updated bays, bs′, is
    22a. that the yielded container, i.e., c, is obtained, get_C(bs)(stid), from the top of the non-empty, designated stack,
    22b. that the mereology of bs′ is unchanged, unchanged_mereology(bs,bs′). wrt. bs. ,
    22c. that the stack designated by stid in the "input" state, bs, is popped, popped_designated_stack(bs,bs′)(stid), and
    22d. that all other stacks are unchanged in bs′ wrt. bs, unchanged_non_designated_stacks(bs,bs′)(stid).

533

```
value
20.  remove_C_from_V: V → StackId ~→ (V×C)
20.  remove_C_from_V(v)(stid) as (v′,c)
20c.    (obs_BS(v′),c) = remove_C_from_BS(obs_BS(v))(stid)
20d.   ∧ props(v)=props(v″)

20b.  remove_C_from_BS: BS → StackId → (BS×C)
20a.  remove_C_from_BS(bs)(stid) as (bs′,c)
21a.        pre:  valid_address(bs)(stid)
21b.         ∧ non_empty_designated_stack(bs)(stid)
22a.        post: c = get_C(bs)(stid)
22b.         ∧ unchanged_mereology(bs,bs′)
22c.         ∧ popped_designated_stack(bs,bs′)(stid)
22d.         ∧ unchanged_non_designated_stacks(bs,bs′)(stid)
```

The props function was introduced in Sect. 7 on page 34.

C.3.2 Remove Container from CTP

534

We define a remove action similar to that of Sect. C.3.1 on the preceding page.

195. Instead of vessel bays we are now dealing with the bays of container terminal ports.

We omit the narrative — which is very much like that of narrative Items 20c and 20d.

```
value
195.  remove_C_from_CTP: CTP → StackId ~→ (CTP×C)
195.  remove_C_from_CTP(ctp)(stid) as (ctp′,c)
20c.     (obs_BS(ctp′),c) = remove_C_from_BS(obs_BS(ctp))(stid)
20d.    ∧ props(ctp)=props(ctp″)
```

C.3.3 Stack Container on Vessel

535

196. Stacking a container at a vessel bay stack location
    a
    b
    c

value
196. stack_C_on_vessel: BS → StackId $\xrightarrow{\sim}$ C $\xrightarrow{\sim}$ BS
196a. stack_C_on_vessel(bs)(stid)(c) **as** bs′
196a. **comment:** bs is bays of a v:V, i.e., bs = obs_BS(v)
196b. **pre**:
196c. **post**:

C.3.4 Stack Container in CTP 536

197.

198.

199.

200.

value
197. stack_C_in_CTP: CTP → StackId → C $\xrightarrow{\sim}$ CTP
198. stack_C_in_CTP(ctp)(stid)(c) **as** ctp′
199. **pre**:
200. **post**:

C.3.5 Transfer Container from Vessel to CTP 537

201.

202.

203.

204.

value
201. transfer_C_from_V_to_CTP: V→StackId$\xrightarrow{\sim}$CTP→StackId$\xrightarrow{\sim}$(V×CTP)
202. transfer_C_from_V_to_CTP(v)(v_stid)(ctp)(ctp_stid) ≡
203. **let** (c,v′) = remove_C_from_V(v)(v_stid) **in**
203. (v′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

C.3.6 Transfer Container from CTP to Vessel 538

205.

206.

207.

value
205. transfer_C_from_CTP_to_V: CTP→StackId$\xrightarrow{\sim}$V→StackId$\xrightarrow{\sim}$(CTP×V)
206. transfer_C_from_CTP_to_V(ctp)(ctp_stid)(v)(v_stid) ≡
207. **let** (c,ctp′) = remove_C_from_CTP(ctp)(ctp_stid) **in**
207. (ctp′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

D RSL: The Raise Specification Language 539

D.1 Type Expressions

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

D.1.1 Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts. 540

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

D.1.2 Composite Types 541

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully "taken apart". There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions. 542

[1] Concrete Composite Types From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

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

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).

5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A*)**-set**, or (A**-set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.
16. The postulated disjoint union of types A, B, ..., and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

543

[2] Sorts and Observer Functions

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

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

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

D.2 Type Definitions

544

D.2.1 Concrete Types

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

type
A = Type_expr

Some schematic type definitions are:

```
[1] Type_name = Type_expr /∗ without |s or subtypes ∗/
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
              mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
              ... |
              mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • P(v) |}
```

545

where a form of [2–3] is provided by combining the types:

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

axiom
∀ a1:A_1, a2:A_2, ..., ai:Ai •
  s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
  ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
∀ a:A • **let** mk_id_1(a1′,a2′,...,ai′) = a **in**
  a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) **end**

D.2.2 Subtypes

546

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

type
A = {| b:B • $\mathcal{P}$(b) |}

D.2.3 Sorts — Abstract Types

547

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

type
A, B, ..., C

D.3 The RSL Predicate Calculus 548

D.3.1 Propositional Expressions

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

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

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

D.3.2 Simple Predicate Expressions 549

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

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

are simple predicate expressions.

D.3.3 Quantified Expressions 550

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

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

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

D.4 Concrete RSL Types: Values and Operations 551

D.4.1 Arithmetic

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

D.4.2 Set Expressions 552

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

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

553

[2] Set Comprehension The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

```
type
  A, B
  P = A → Bool
  Q = A ~→ B
value
  comprehend: A-infset × P × Q → B-infset
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

D.4.3 Cartesian Expressions

554

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

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

D.4.4 List Expressions

555

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

```
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

556

[2] List Comprehension The last line below expresses list comprehension.

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

D.4.5 Map Expressions

557

[1] Map Enumerations Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

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

558

[2] Map Comprehension The last line below expresses map comprehension:

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

D.4.6 Set Operations

559

[1] Set Operator Signatures

```
value
  19 ∈: A × A-infset → Bool
  20 ∉: A × A-infset → Bool
  21 ∪: A-infset × A-infset → A-infset
  22 ∪: (A-infset)-infset → A-infset
  23 ∩: A-infset × A-infset → A-infset
  24 ∩: (A-infset)-infset → A-infset
  25 \: A-infset × A-infset → A-infset
  26 ⊂: A-infset × A-infset → Bool
  27 ⊆: A-infset × A-infset → Bool
  28 =: A-infset × A-infset → Bool
  29 ≠: A-infset × A-infset → Bool
  30 card: A-infset ~~> Nat
```

560

[2] Set Examples

```
examples
  a ∈ {a,b,c}
  a ∉ {}, a ∉ {b,c}
  {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
  ∪{{a},{a,b},{a,d}} = {a,b,d}
  {a,b,c} ∩ {c,d,e} = {c}
  ∩{{a},{a,b},{a,d}} = {a}
  {a,b,c} \ {c,d} = {a,b}
  {a,b} ⊂ {a,b,c}
  {a,b,c} ⊆ {a,b,c}
  {a,b,c} = {a,b,c}
  {a,b,c} ≠ {a,b}
  card {} = 0, card {a,b,c} = 3
```

561

[3] Informal Explication

19. ∈: The membership operator expresses that an element is a member of a set.
20. ∉: The nonmembership operator expresses that an element is not a member of a set.

21. ∪: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22. ∪: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23. ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24. ∩: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets. 562
25. \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26. ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. ⊂: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28. =: The equal operator expresses that the two operand sets are identical.
29. ≠: The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set. 563

[4] Set Operator Definitions The operations can be defined as follows (≡ is the definition symbol):

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

D.4.7 Cartesian Operations 564

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

```
value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3
```

decomposition expressions
let (a1,b1,c1) = g0,
(a1′,b1′,c1′) = g1 **in** .. **end**
let ((a2,b2),c2) = g2 **in** .. **end**
let (a3,(b3,c3)) = g3 **in** .. **end**

D.4.8 List Operations

565

[1] List Operator Signatures

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

[2] List Operation Examples

examples
hd⟨a1,a2,...,am⟩=a1
tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
len⟨a1,a2,...,am⟩=m
inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
566 ⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩
567

[3] Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of
568 elements larger than or equal to i, gives the ith element of the list.
- ⌢: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

569 The operations can also be defined as follows:

[4] List Operator Definitions

```
value
  is_finite_list: A^ω → Bool

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

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

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

570

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

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

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

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

D.4.9 Map Operations

571

[1] Map Operator Signatures and Map Operation Examples

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

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

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

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

572

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

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

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

=,≠: M × M → **Bool**

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

573

[2] Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction
574 of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

575

[3] Map Operation Redefinitions The map operations can also be defined as follows:

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

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

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

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

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

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

D.5 λ-Calculus + Functions 576

D.5.1 The λ-Calculus Syntax

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

D.5.2 Free and Bound Variables 577

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

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

D.5.3 Substitution 578

In RSL, the following rules for substitution apply:

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

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

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

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

D.5.4 α-Renaming and β-Reduction 579

- α-renaming: λx•M

  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.

- β-reduction: (λx•M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

D.5.5 Function Signatures 580

For sorts we may want to postulate some functions:

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

D.5.6 Function Definitions 581

Functions can be defined explicitly:

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

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

582

Or functions can be defined implicitly:

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

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

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

D.6 Other Applicative Expressions 583

D.6.1 Simple let Expressions

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

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

is an "expanded" form of:

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

D.6.2 Recursive let Expressions 584

Recursive **let** expressions are written as:

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

is "the same" as:

let f = **YF** **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

D.6.3 Predicative let Expressions 585

Predicative **let** expressions:

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

D.6.4 Pattern and "Wild Card" let Expressions 586

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩^ℓ = list **in** ... **end**
let ⟨a,_,b⟩^ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

D.6.5 Conditionals 587

Various kinds of conditional expressions are offered by RSL:

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
   if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
   choice_pattern_1 → expr_1,
   choice_pattern_2 → expr_2,
   ...
   choice_pattern_n_or_wild_card → expr_n
end
```

D.6.6 Operator/Operand Expressions 588

```
⟨Expr⟩ ::=
          ⟨Prefix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Suffix_Op⟩
        | ...
⟨Prefix_Op⟩ ::=
        − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
         = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
        | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | °
⟨Suffix_Op⟩ ::= !
```

D.7 Imperative Constructs 589

D.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

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

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).

- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

D.7.2 Variables and Assignment 590

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

D.7.3 Statement Sequences and skip 591

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

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

D.7.4 Imperative Conditionals 592

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

D.7.5 Iterative Conditionals 593

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

D.7.6 Iterative Sequencing 594

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

D.8 Process Constructs 595

D.8.1 Process Channels

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

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

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

D.8.2 Process Composition 596

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

```
P ‖ Q    Parallel composition
P [] Q   Nondeterministic external choice (either/or)
P ⌈⌉ Q   Nondeterministic internal choice (either/or)
P ‡ Q    Interlock parallel composition
```

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

D.8.3 Input/Output Events 597

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

```
c ?, k[i] ?      Input
c ! e, k[i] ! e  Output
```

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

D.8.4 Process Definitions 598

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

```
value
  P: Unit → in c out k[i]
  Unit
  Q: i:KIdx →  out c in k[i] Unit

  P() ≡ ... c ? ... k[i] ! e ...
  Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

D.9 Simple RSL Specifications 599

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

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

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[60]

[60] For schemes, classes and objects we refer to [8, Chap. 10]

E Indexes 600

RSL Index

Definition Index

Example Index

Concept Index

Language, Method and Technology Index

Selected Author Index