Domain Science & Engineering*

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

September 5, 2012: 11:29

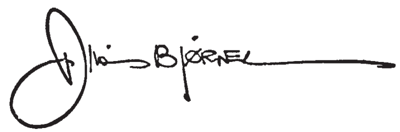
1
2
Abstract

This paper covers a **new science & engineering of domains** as well as a **new foun-**
dation for software development. We treat the latter first. Instead of commencing with 3
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 4
time, 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 descrip-
tions are void of any reference to requirements and encompass the modelling of domain
phenomena without regard to their being computable. 5
(ii) **Another facet** is **the pursuit of domain descriptions as a free-standing activity.**
In this paper 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 paper we shall investigate a method 6
for analysing domains, for constructing domain descriptions and some emerging scientific
bases. 7
Our contribution to **domain analysis** is that we view domain analysis as a variant of
formal concept analysis [38], a contribution which can be formulated by the "catch phrase"
domain entitities and their qualities form Galois connections, and further contribute
with a methodology of necessary corresponding principles and techniques of domain anal-
ysis. Those corresponding principles and techniques hinge on our view of domains as 8
having the following **ontology**. There are the entities that we can describe and then there
is "the rest" which we leave un-described. We analyse entities into endurant entities and
perdurant entities , that is, parts and materials as endurant entities and discrete actions,
discrete events and behaviours as perdurant entities , respectively. Another way of looking
at entities is as discrete entities , or as continuous entities. We also contribute to the analysis 9
of discrete endurants in terms of the following notions: part types and material types, part
unique identifiers, part mereology and part attributes and material attributes and **material**
laws. Of the above we point to the introduction, into computing science and software
engineering of the notions of materials and continuous behaviours **as novel.** 10
The example formalisations are expressed in `RAISE` [40] (with [8, 9, 10] being a rather
comprehensive monograph cum textbook), but could as well have been expressed in, for
example, `Alloy` [50], `Event B` [1] , `VDM` [18, 19, 35] or `Z` [105].

*

- **Administrative Notes:**
  - This document serves as a basis for my full day tutorial at the FM 2012 International Symposium (http://fm2012.cnam.fr/), August 28, at Conservatoire National des Arts et Métiers, 292 rue Saint-Martin, F-75141 Paris, France.
  - My 31 December 2011 Tutorial Proposal, so kindly accepted by the relevant FM 2012 committees can be found at **http://www2.imm.dtu.dk/~dibj/fm2012/31-12-2011-tutorial-bjorner.pdf**.
  - The FM 2012 Tutorial Programme organisers have kindly prepared a volume of the tutorial lecture notes.
    - The lecture notes for the tutorial related to the present document are (most likely) dated July 14, 2012 (or could be as early as July 3, 2012) at **http://-www2.imm.dtu.dk/~dibj/fm2012/Bjorner-FM2012-Notes-july14.pdf**.
    - The present document, and the tutorial that will be presented is a rather complete rewrite, restructuring, re-editing and, I shall claim, rather significant improvement over earlier attempts.
    - This work took place between July 14 and August 22, 2012.
- **Editorial Notes:**
  - In the present document you will notice some margin numerals. They refer to slide numbers for the of slides that correspond to this document.
  - You will find a 4:1 reduced set of these slides at **http://www2.imm.dtu.dk/~dibj/4-dsae-f.pdf**.
- **Thanks:** DTU Informatics have kindly let print and bind a set of these lecture notes.

Special thanks are due to Mr. Finn Kuno Christensen, DTU Informatik.



Contents

1 Introduction 11

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

This is primarily a methodology paper. By a method$_\delta$ 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$_\delta$ we shall understand the study and knowledge about methods. 12

This paper 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 13

By a domain$_\delta$ 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. 14

Example: 1 Some Domains Some examples are: 20

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

1.1.1 Domain Analysis 15

By domain analysis$_\delta$ we shall understand an inquiry into the domain, its entities and their properties. 16

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*. 23

1.1.2 Domain Descriptions 17

By a domain description$_\delta$ 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. 18

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**. ■

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 19

By domain engineering$_\delta$ 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. The size (usually, say a hundred pages), structure (usually a finely sectioned document of may subsub···subsections) and complexity (having many cross-references between subsub···subsections) 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.

1.1.4 Domain Science 21

By domain science$_\delta$ 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.

1.2 The `Triptych` of Software Development 22

We suggest a "dogma": before software can be designed one must understand[1] the requirements; and before requirements can be expressed one must understand[2] 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[3] ($\mathcal{D}$) a domain is analysed, described and "theorised", that is, the beginnings of a specific domain theory is established. In the requirements engineering phase[4] ($\mathcal{R}$) a requirements prescription is constructed — significant fragments of which are "derived", systematically, from the domain description. In the software design phase[5] ($\mathcal{S}$)

[1] Or maybe just: have a reasonably firm grasp of
[2] See previous footnote!
[3] See Sects. 4–6
[4] See Sect. 8
[5] We do not illustrate the software design phase in this paper.

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: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$. 24

By a machine$_\delta$ we shall understand the hardware and software[6] of a target, i.e., a required IT system.

In [11, 17, 14] 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 25 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. 26

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. 8.

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. 8. When you review this section ('A Triptych of 27 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 [15]. You may also characterise domain-specific developments as representing another 'definition' of domain engineering.

1.3 Issues of Domain Science & Engineering 28

We specifically focus on the following issues of domain science &[7] engineering: (i) which are the "things" to be described[8], (ii) how to analyse these "things" into constituent description structures[9], (iii) how to describe these "things" informally and formally, (iv) how to further structure descriptions[10], and a further study of (v) mereology[11].

[6] By software$_\delta$ 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.

[7] When we put '&' between two terms that the compound term forms a whole concept.

[8] endurants [manifest entities henceforth called parts and materials] and perdurants [actions, events, behaviours]

[9] atomic and composite, unique identifiers, mereology, attributes

[10] *intrinsics, support technology, rules & regulations, organisation & management, human behaviour* etc.

[11] the study and knowledge of parts and relations of parts to other parts and a "whole".

1.4 Structure of Paper 29

First, Sect. 1, we introduce the problem. And that was done above.

Then, in Sects. 4–6 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 paper 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 30 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 31 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 32 practices in domain analysis.

In Sect. 7 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 per- 33 son 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 34 number of laws that the domain description process ought satisfy.

In Sect. 8 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 35 from domain descriptions.

• • •

The formal descriptions will here be expressed in the `RAISE` [40] Specification Language, `RSL`. We otherwise refer to [8]. 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` [50], `Event B` [1] , `VDM` [18, 19, 35] and `Z` [105].

2 The Main Example: Road Traffic System 36

Example: 4 The Main Example. The main example presents a terse narrative and formalisation of a road traffic domain. Since the example description conceptually covers also major aspects of railroad nets, shipping nets, and air traffic nets, we shall use such terms as hubs and links to stand for road (or street) intersection and road (or street) segments, train stations and rail lines, harbours and shipping lanes, and airports and air lanes.

2.1 Parts 37

2.1.1 Root Sorts

The domain, the stepwise unfolding of whose description is to be exemplified, is that of a composite traffic system (i) with a road net, (ii) with a fleet of vehicles (iii) of whose individual position on the road net we can speak, that is, monitor. 38

1. We analyse the composite traffic system into
   a a composite road net,
   b a composite fleet (of vehicles), and
   c an atomic monitor.

type
1. Δ
1a. N
1b. F
1c. M
value
1a. **obs_**N: $\Delta \rightarrow$ N
1b. **obs_**F: $\Delta \rightarrow$ F
1c. **obs_**M: $\Delta \rightarrow$ M

2.1.2 Sub-domain Sorts and Types 39

2. From the road net we can observe
   a a composite part, HS, of road (i.e., street) intersections (hubs) and
   b an composite part, LS, of road (i.e., street) segments (links).

type
2. HS, LS
value
2a. **obs_**HS: N $\rightarrow$ HS
2b. **obs_**LS: N $\rightarrow$ LS

40

3. From the fleet sub-domain, F, we observe a composite part, VS, of vehicles

type
3. VS
value
3. **obs_**VS: F $\rightarrow$ VS

41

4. From the composite sub-domain VS we observe
   a the composite part Vs, which we concretise as a set of vehicles
   b where vehicles, V, are considered atomic.

type
4a. Vs = V-**set**
4b. V
value
4a. **obs_**Vs: VS $\rightarrow$ V-**set**

42

The "monitor" is considered atomic; it is an abstraction of the fact that we can speak of the positions of each and every vehicle on the net without assuming that we can indeed pin point these positions by means of for example sensors.

2.1.3 Further Sub-domain Sorts and Types 43

We now analyse the sub-domains of HS and LS.

5. From the hubs aggregate we decide to observe
   a the concrete type of a set of hubs,
   b where hubs are considered atomic; and
6. from the links aggregate we decide to observe
   a the concrete type of a set of links,
   b where links are considered atomic;

44

type
5a. Hs = H-**set**
6a. Ls = L-**set**
5b. H
6b. L
value
5. **obs_**Hs: HS $\rightarrow$ H-**set**
6. **obs_**Ls: LS $\rightarrow$ L-**set**

45

We have no composite parts left to further analyse into parts whether they be again composite or atomic. That is, at various, what we shall refer to as, domain indexes[12] we have discovered the following part types:

[12] We shall take up the notion of domain index in Sect. 7.1.3 on Page 78.

- $\langle\Delta\rangle$: N, F, M
- $\langle\Delta, N\rangle$: HS, LS
- $\langle\Delta, F\rangle$: VS
- $\langle\Delta, HS\rangle$: Hs, H
- $\langle\Delta, LS\rangle$: Ls, L
- $\langle\Delta, VS\rangle$: Vs, V

Thus we have ended up with atomic parts.

2.2 Properties 46 49

Parts are distinguished by their properties: the types and the values of these. We consider three kinds of properties: unique identifiers, mereology and attributes.

2.2.1 Unique Identifications 47

There is, for any traffic system, exactly one composite aggregation, HS, of hubs, exactly one composite aggregation, Hs, of hubs, exactly one composite aggregation, LS, of links, exactly one composite aggregation, Ls, of links, exactly one composite aggregation, VS, of vehicles and exactly one composite aggregation, Vs, of vehicles, Therefore we shall not need to associate unique identifiers with any of these.

7. We decide the following: 50
   a each hub has a unique hub identifier,
   b each link has a unique link identifier and
   c each vehicle has a unique vehicle identifier.

```
type
7a.  HI
7b.  LI
7c.  VI                                    52
value
7a.  uid_H: H → HI
7b.  uid_L: L → LI
7c.  uid_V: V → VI
```

2.2.2 Mereology 48

[1] Road Net Mereology: By *mereology* we mean the study, knowledge and practice of understanding parts and part relations.

The relations between, that is, the mereology of, the composite parts of the road net, n:N, are simple: there is one HS part of n:N; there is one Hs part of the only HS part of n:N; there is one LS part of n:N; and there is one Ls part of the only LS part of n:N. Therefore we shall not associate any special mereology based on unique identifiers which we therefore also decided to not express for these composite parts. 53

8. Each link is connected to exactly two hubs, that is,

   a from each link we can observe its mereology, that is, the identities of these two distinct hubs,
   b and these hubs must be of the net of the link;

9. and each hub is connected to zero, one or more links, that is,
   a from each hub we can observe its mereology, that is, the identities of these links,
   b and these links must be of the net of the hub.

```
value
8a.  mereo_L: L → HI-set,   axiom ∀ l:L•card mereo_L(l)=2
axiom
8b.  ∀ n:N,l:L,hi:HI • l ∈ obs_Ls(obs_LS(n)) ∧ hi ∈ mereo_L(l)
8b.       ⇒ ∃ h:H•h ∈ obs_Hs(obs_HS(n))∧uid_H(h)=hi
value
9a.  mereo_H: H → LI-set
axiom
9b.  ∀ n:N,h:H,li:LI • h ∈ obs_Hs(obs_HS(n)) ∧ li ∈ mereo_H(h)
9b.       ⇒ ∃ l:L•l ∈ obs_Ls(obs_LS(n))∧uid_L(l)=li
```

[2] Fleet of Vehicles Mereology: In the traffic system that we are building up there are no relations to be expressed between vehicles, only between vehicles and the (single and only) monitor. Thus there is no mereology needed for vehicles.

2.2.3 Attributes 51

We shall model attributes of links, hubs and vehicles. The composite parts, aggregations of hubs, HS and Hs, aggregations of links, LS and Ls and aggregations of vehicles, VS and Vs, also have attributes, but we shall omit modelling them here.

[1] Attributes of Links:

10. The following are attributes of links.
   a Link states, lσ:LΣ, which we model as possibly empty sets of pairs of distinct identifiers of the connected hubs. A link state expresses the directions that are open to traffic across a link.
   b Link state spaces, lω:LΩ which we model as the set of link states. A link state space expresses the states that a link may attain across time.
   c Further link attributes are length, location, etcetera.

Link states are usually dynamic attributes whereas link state spaces, link length and link location (usually some curvature rendition) are considered static attributes.

```
type
10a.  LΣ = (HI × HI)-set
axiom
```

10a. $\forall$ lσ:LΣ • $0 \leq$ **card** lσ ≤ 2
value
10a. <u>**attr_**</u>LΣ: L $\rightarrow$ LΣ
axiom
10a. $\forall$ l:L • **let** {hi,hi′}=<u>**mereo_**</u>L(l) **in** <u>**attr_**</u>LΣ(l)$\subseteq${(hi,hi′),(hi′,hi)} **end**
type
10b. LΩ = LΣ-**set**
value
10b. <u>**attr_**</u>LΩ: L $\rightarrow$ LΩ
axiom
10b. $\forall$ l:L • **let** {hi,hi′}=<u>**mereo_**</u>L(l) **in** <u>**attr_**</u>LΣ(l)$\in$ <u>**attr_**</u>LΩ(l) **end**
type
10c. LOC, LEN, ...
value
10c. <u>**attr_**</u>LOC: L $\rightarrow$ LOC, <u>**attr_**</u>LEN: L $\rightarrow$ LEN, ...

54

[2] Attributes of Hubs:

11. The following are attributes of hubs:
    a Hub states, hσ:HΣ, which we model as possibly empty sets of pairs of identifiers of the connected links. A hub state expresses the directions that are open to traffic across a hub.
    b Hub state spaces, hω:HΩ which we model as the set of hub states. A hub state space expresses the states that a hub may attain across time.
    c Further hub attributes are location, etcetera.

Hub states are usually dynamic attributes whereas hub state spaces and hub location are considered static attributes. 55

type
11a. HΣ = (LI $\times$ LI)-**set**
value
11a. <u>**attr_**</u>HΣ: H $\rightarrow$ HΣ
axiom
11a. $\forall$ h:H • <u>**attr_**</u>HΣ(h)$\subseteq${(li,li′)|li,li′:LI•{li,li′}$\subseteq$<u>**mereo_**</u>H(h)}
type
11b. HΩ = HΣ-**set**
value
11b. <u>**attr_**</u>HΩ: H $\rightarrow$ HΩ
axiom
11b. $\forall$ h:H • <u>**attr_**</u>HΣ(h) $\in$ <u>**attr_**</u>HΩ(h)
type
11c. LOC, ...
value
11c. <u>**attr_**</u>LOC: L $\rightarrow$ LOC, ...

56

[3] Attributes of Vehicles:

12. 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.
13. All these vehicle attributes can be observed.

57

type
12a. VP = atH | onL
12(a)i. atH :: fli:LI $\times$ hi:HI $\times$ tli:LI
12(a)ii. onL :: fhi:HI $\times$ li:LI $\times$ frac:FRAC $\times$ thi:HI
12(a)iii. FRAC = **Real**, **axiom** $\forall$ frac:FRAC • $0 \leq$ frac ≤ 1
12b. VEL, ACC, ...
value
13. <u>**attr_**</u>VP:V$\rightarrow$VP, <u>**attr_**</u>onL:V$\rightarrow$onL, <u>**attr_**</u>atH:V$\rightarrow$atH
13. <u>**attr_**</u>VEL:V$\rightarrow$VEL, <u>**attr_**</u>ACC:V$\rightarrow$ACC

58

[4] Vehicle Positions:

14. Given a net, n:N, we can define the possibly infinite set of potential vehicle positions on that net, vps(n).
    a vps(n) is expressed in terms of the links and hubs of the net.
    b vps(n) is the
    c union of two sets:
        i. the potentially[13] infinite set of "on link" positions
        ii. for all links of the net

    and
        i. the finite set of "at hub" positions
        ii. for all hubs in the net.

59

[13] The 'potentiality' arises from the nature of FRAC. If fractions are chosen as, for example, 1/5'th, 2/5'th, ..., 4/5'th, then there are only a finite number of "on link" vehicle positions. If instead fraction are arbitrary infinitesimal quantities, then there are infinitely many such.

```
value
14.     vps: N → VP-infset
14b.    vps(n) ≡
14a.        let ls=obs_Ls(obs_LS(n)), hs=obs_Hs(obs_HS(n)) in
14(c)i.       { onL(fhi,uid(l),f,thi) | fhi,thi:HI,l:L,f:FRAC •
14(c)ii.            l ∈ ls ∧ {fhi,thi}=mereo_L(l) }
14c.          ∪
14(c)i.       { atH(fli,uid_H(h),tli) | fli,tli:LI,h:H •
14(c)ii.            h ∈ hs ∧ {fli,tli}⊆mereo_H(h) }
14a.        end
```

Given a net and a finite set of vehicles we can distribute these over the net, i.e., assign initial vehicle positions, so that no two vehicles "occupy" the same position, i.e., are "crashed"! Let us call the non-deterministic assignment function, i.e., a relation, for vpr. 60

15. vpm:VPM is a bijective map from vehicle identifiers to (distinct) vehicle positions.
16. vpr has the obvious signature.
17. vpr(vs)(n) is defined in terms of
18. a non-deterministic selection, vpa, of vehicle positions, and
19. a non-deterministic assignment of these vehicle positions to vehicle identifiers —
20. being the resulting distribution.

61

```
type
15.   VPM′ = VI →m VP
15.   VPM = {| vpm:VPM′ • card dom vpm = card rng vpm |}
value
16.   vpr: V-set × N → VMP
17.   vpr(vs)(n) ≡
18.      let vpa:VP-set • vpa ⊆ vps(vs)(n) ∧ card vpa = vard vs in
19.      let vpm:VPM • dom vpm = vps ∧ rng vpm = vpa in
20.      vpm end end
```

2.3 Definitions of Auxiliary Functions 62

21. From a net we can extract all its link identifiers.
22. From a net we can extract all its hub identifiers.

```
value
21.   xtr_LIs: N → LI-set
21.   xtr_LIs(n) ≡ {uid_L(l)|l:L•l ∈ obs_Ls(obs_LS(n))}
22.   xtr_HIs: N → HI-set
22.   xtr_HIs(n) ≡ {uid_H(l)|h:H•h ∈ obs_Hs(obs_HS(n))}
```


23. Given a link identifier and a net get the link with that identifier in the net.
24. Given a hub identifier and a net get the hub with that identifier in the net.

63

```
value
26.    get_H: HI → N ~→ H
26.    get_H(hi)(n) ≡ ι h:H•h ∈ obs_Hs(obs_HS(n))∧uid_H(h)=hi
26.       pre: hi ∈ xtr_HIs(n)
26a.   get_L: LI → N ~→ L
26a.   get_L(li)(n) ≡ ι l:L•l ∈ obs_Ls(obs_LS(n))∧uid_L(l)=li
26a.      pre: hl ∈ xtr_LIs(n)
```

The ι a:A•$\mathcal{P}$(a) expression yields the unique value a:A which satisfies the predicate $\mathcal{P}$(a). If none, or more than one exists then the function is undefined.

2.4 Some Derived Traffic System Concepts 64

2.4.1 Maps

25. A road map is an abstraction of a road net. We define one model of maps below.
    a A road map, RM, is a finite definition set function, M, (a specification language map) from
      - hub identifiers (the source hub)
      - to (such finite definition set) functions from link identifiers
      - to hub identifiers (the target hub).

```
type
25a.   RM′ = HI →m (LI →m HI)
```

65 If a hub identifier in the source or an rm:RM maps into the empty map then the "corresponding" hub is "isolated": has no links emanating from it.

26. These road maps are subject to a well-formedness criterion.
    a The target hubs must be defined also as source hubs.
    b If a link is defined from source hub (referred to by its identifier) shi via link li to a target hub thi, then, vice versa, link li is also defined from source thi to target shi.

```
type
26.    RM = {| rm:RM′ • wf_RM(rm) |}
value
26.    wf_RM: RM′ → Bool
26.    wf_RM(rm) ≡
26a.       ∪ { rng(rm(hi))|hi:HI•hi ∈ dom rm } ⊆ dom rm
26b.     ∧ ∀ shi:HI•shi ∈ dom rm ⇒
26b.          ∀ li:LI • li ∈ dom rm(shi) ⇒
26b.             li ∈ dom rm((rm(shi))(li)) ∧ (rm((rm(shi))(li)))(li)=shi
```

66

27. Given a road net, n, one can derive “its” road map.
    a Let hs and ls be the hubs and links, respectively of the net n.
    b Every hub with no links emanating from it is mapped into the empty map.
    c For every link identifier uid_L(l) of links, l, of ls and every hub identifier, hi, in the mereology of l
    d hi is mapped into a map from uid_L(l) into hi'
    e where hi' is the other hub identifier of the mereology of l.

67

```
value
27.   derive_RM: N → RM
27.   derive_RM(n) ≡
27a.      let hs = obs_Hs(obs_HS(n)), ls = obs_Ls(obs_LS(n)) in
27b.      [ hi ↦ [ ] | hi:HI • ∃ h:H • h ∈ hs ∧ mereo_H(h) = {} ] ∪
27d.      [ hi ↦ [ uid_L(l) ↦ hi′
27e.                      | hi′:HI • hi′ = mereo_L(l)\{hi} ]
27c.             | l:L,hi:HI • l ∈ ls ∧ hi ∈ mereo_L(l) ] end
```

Theorem: If the road net, n, is well-formed then wf_RM(derive_RM(n)).

2.4.2 Traffic Routes 68

28. A traffic route, tr, is an alternating sequence of hub and link identifiers such that
    a li:LI is in the mereology of the hub, h:H, identified by hi:HI, the predecessor of li:LI in route r, and
    b hi':HI, which follows li:LI in route r, is different from hi, and is in the mereology of the link identified by li.

```
type
28.   R′ = (HI|LI)*
28.   R = {| r:R′ • ∃ n:N • wf_R(r)(n) |}
value
28.   wf_R: R′ → N → Bool
28.   wf_R(r)(n) ≡
28.     ∀ i:Nat • {i,i+1}⊆inds r ⇒
28a.        is_HI(r(i)) ⇒ is_LI(r(i+1)) ∧ r(i+1) ∈ mereo_H(get_H(r(i))(n)),
28b.        is_LI(r(i)) ⇒ is_HI(r(i+1)) ∧  r(i+1) ∈ mereo_L(get_L(r(i))(n))
```

69

29. From a well-formed road map (i.e., a road net) we can generate the possibly infinite set of all routes through the net.
    a **Basis Clauses:**
       i. The empty sequence of identifiers is a route.

       ii. The one element sequences of link and hub identifiers of links and hubs of a road map (i.e., a road net) are routes.
       iii. If hi maps into some li in rm then ⟨hi,li⟩ and ⟨li,hi⟩ are routes of the road map (i.e., of the road net).
    b **Induction Clause:**
       i. Let r⌢⟨i⟩ and ⟨i′⟩⌢r′ be two routes of the road map.
       ii. If the identifiers i and i′ are identical, then r⌢⟨i⟩⌢r′ is a route.
    c **Extremal Clause:**
       i. Only such routes that can be formed from a finite number of applications of the above clauses are routes.

70

```
value
29.      gen_routes: M → Routes-infset
29.      gen_routes(m) ≡
29(a)i.      let rs = {⟨⟩}
29(a)ii.          ∪ {⟨li,hi⟩,⟨hi,li⟩|li:LI,hi:HI•...}
29(b)i.           ∪ {let r⌢⟨li⟩,⟨li′⟩⌢r′:R • {r⌢⟨li⟩,⟨li′⟩⌢r′}⊆rs,
29(b)i.                  r″⌢⟨hi⟩,⟨hi′⟩⌢r‴:R • {r″⌢⟨hi⟩,⟨hi′⟩⌢r‴}⊆rs in
29(b)ii.              r⌢⟨li⟩⌢r′,r″⌢⟨hi⟩⌢r‴ end} in
29(c)i.       rs end
```

71

[1] Circular Routes:

30. A route is circular if the same identifier occurs more than once.

```
value
30.  is_circular_route: R → Bool
30.  is_circular_route(r) ≡ ∃ i,j:Nat • {i,j}⊆inds r ∧ i≠j ⇒ r(i)=r(j)
```

72

[2] Connected Road Nets:

31. A road net is connected if there is a route from any hub (or any link) to any other hub or link in the net.

```
31.   is_conn_N: N → Bool
31.   is_conn_N(n) ≡
31.     let m = derive_RM(n) in
31.     let rs = gen_routes(m) in
31.     ∀ i,i′:(LI|HI) • {i,i′}⊆xtr_LIs(n)∪ xtr_HIs(n)
31.        ∃ r:R • r ∈ rs ∧ r(1)=i ∧ r(len r)=i′ end end
```

73

[3] Set of Connected Nets of a Net:

32. The set, cns, of connected nets of a net, n, is

    a the smallest set of connected nets, cns,

    b whose hubs and links together "span" those of the net n.

```
value
32.  conn_Ns: N → N-set
32.  conn_Ns(n) as cns
32a.     pre: true
32b.     post: conn_spans_HsLs(n)(cns)
32a.         ∧ ∼∃ kns:N-set • card kns < card cns
32a.             ∧ conn_spans_HsLs(n)(kns)
```

74

```
32b.  conn_spans_HsLs: N → N → Bool
32b.  conn_spans_HsLs(n)(cns) ≡
32b.     ∀ cn:N•cn ∈ cns ⇒ is_connected_N(n)(cn)
32b.    ∧ let (hs,ls) = (obs_Hs(obs_HS(n)),obs_Ls(obs_LS(n))),
32b.            chs = ∪{obs_Hs(obs_HS(cn))|cn ∈ cns},
32b.            cls = ∪{obs_Ls(obs_LS(cn))|cn ∈ cns} in
32b.       hs = chs ∧ ls = cls end
```

75

[4] Route Length:

33. The length attributes of links can be

    a added and subtracted,

    b multiplied by reals to obtain lengths,

    c divided to obtain fractions,

    d compared as to whether one is shorter than another, etc., and

    e there is a "zero length" designator.

```
value
33a.  +,− : LEN × LEN → LEN
33b.  ∗ : LEN × Real → LEN
33c.  / : LEN × LEN → Real
33d.  <,≤,=,≠,≥,> : LEN × LEN → Bool
33e.  ℓ0 : LEN
```

76

34. One can calculate the length of a route.


```
value
34.  length: R → N → LEN
34.  length(r)(n) ≡
34.     case r of:
34.        ⟨⟩ → ℓ0,
34.        ⟨si⟩⌢r′ →
34.           is_LI(si)→attr_LEN(get_L(si)(n))+length(r′)(n)
34.           is_HI(si)→length(r′)(n)
34.     end
```

77

[5] Shortest Routes:

35. There is a predicate, is_R, which,

    a given a net and two distinct hub identifiers of the net,

    b tests whether there is a route between these.

```
value
35.  is_R: N → (HI×HI) → Bool
35.  is_R(n)(fhi,thi) ≡
35a.    fhi ≠ thi ∧ {fht,thi}⊆xtr_HIs(n)
35b.    ∧ ∃ r:R • r ∈ routes(n) ∧ hd r = fhi ∧ r(len r) = thi
```

78

36. The shortest between two given hub identifiers

    a is an acyclic route, r,

    b whose first and last elements are the two given hub identifiers

    c and such that there is no route, r′ which is shorter.

```
value
36.  shortest_route: N → (HI×HI) → R
36a.  shortest_route(n)(fhi,thi) as r
36b.     pre: pre_shortest_route(n)(fhi,thi)
36c.     post: pos__shortest_route(n)(r)(fhi,thi)
```

79

```
36b.  pre_shortest_route: N → (HI×HI) → Bool
36b.  pre_shortest_route(n)(fhi,thi) ≡
36b.     is_R(n)(fhi,thi) ∧ fhi≠thi ∧ {fhi,thi}⊂xtr_HIs(n)

36c.  pos_shortest_route: N → R → (HI×HI) → Bool
36c.  pos_shortest_route(n)(r)(fhi,thi) ≡
36c.     r ∈ routes(n)
36c.    ∧ ∼∃ r′:R • r′ ∈ routes(n) ∧ length(r′) < length(r)
```

2.5 States 80

There are different notions of state. In our example these are some of the states: the road net composition of hubs and links; the state of a link, or a hub; and the vehicle position.

2.6 Actions 81

An action is what happens when a function invocation changes, or potentially changes a state. Examples of traffic system actions are: insertion of hubs, insertion of links, removal of hubs, removal of links, setting of hub state (hσ), setting of link state (lσ), moving a vehicle along a link, moving a vehicle from a link to a hub and moving a vehicle from a hub to a link. 82

37. 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.

83

value
37. ins_H: N → H $\xrightarrow{\sim}$ N
37. ins_H(n)(h) **as** n′, **pre**: pre_ins_H(n)(h), **post**: post_ins_H(n)(h)

37a. pre_ins_H(n)(h) ≡
37a. ∼∃ h′:H • h′ ∈ **obs_**Hs(n) ∧ **uid_**HI(h)=**uid_**HI(h′)
37b. ∧ **mereo_**H(h) = {}

37c. post_ins_H(n)(h)(n′) ≡
37c. **obs_**Hs(n) ∪ {h} = **obs_**Hs(n′)
37d. ∧ **obs_**Ls(n) = **obs_**Ls(n′)

2.7 Events 84

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". 85

38. Link disappearance is expressed as a predicate on the "before" and "after" states of the net. The predicate identifies the "missing" ℓink (!).
39. 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 uid_Π(ℓ).

38. link_dis: N × N → **Bool**
38. link_dis(n,n′) ≡
38. ∃ ℓ:L • pre_link_dis(n,ℓ) ⇒ post_link_dis(n,ℓ,n′)
39. pre_link_dis: N × L → **Bool**
39. pre_link_dis(n,ℓ) ≡ ℓ ∈ **obs_**Ls(n)

86

40. 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. $\{l''_1,l''_2,\ldots,l''_q\} \setminus \{\mathsf{uid\_\Pi}(\ell)\} \cup \{\mathsf{uid\_\Pi}(\ell_j)\}$
41. All other hubs and links of n are unaffected.

87

42. 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$_\alpha$ and h$_\beta$;
   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$_\alpha$ and
      ii. l$_{k\beta}$ connects hi$_k$ and h$_{k\beta}$

88

value
42. post_link_dis(n,ℓ,n′) ≡
42. **let** h_a,h_b:H •
42. **let** {li_a,li_b}=**mereo_**L(ℓ) **in**
42. (get_H(li_a)(n),get_H(li_b)(n)) **end in**
42a. **let** n″ = rem_L(n)(**uid_**L(ℓ)) **in**
42b. **let** h$_\alpha$,h$_\beta$:H • {h$_\alpha$,h$_\beta$}∩**obs_**Hs(n)={} **in**
42b. **let** n‴ = ins_H(n″)(h$_\alpha$) **in**
42b. **let** n⁗ = ins_H(n‴)(h$_\beta$) **in**
42c. **let** l$_{j\alpha}$,l$_{k\beta}$:L • {l$_{j\alpha}$,l$_{k\beta}$}∩**obs_**Ls(n)={}
42c. ∧ **mereo_**L(l$_{j\alpha}$) = {**uid_**H(h_a),**uid_**H(h$_\alpha$)}
42c. ∧ **mereo_**L(l$_{k\beta}$) = {**uid_**H(h_b),**uid_**H(h$_\beta$)} **in**
42(c)i. **let** n′′′′′ = ins_L(n⁗)(l$_{j\alpha}$) **in**
42(c)ii. n′ = ins_L(n′′′′′)(l$_{k\beta}$) **end end end end end end end**

2.8 Behaviours

89

2.8.1 Traffic

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

43. the continuous time varying discrete positions of vehicles, vp:VP[14],

44. where time is taken as a dense set of points.

type
44. c$\mathbb{T}$
43. cRTF = c$\mathbb{T}$ → (V $\overrightarrow{m}$ VP)

90

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

45. discrete time varying discrete positions of vehicles,

46. where time can be considered a set of linearly ordered points.

46. d$\mathbb{T}$
45. dRTF = d$\mathbb{T}$ $\overrightarrow{m}$ (V $\overrightarrow{m}$ VP)

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

type
47. RTF = d$\mathbb{T}$ $\overrightarrow{m}$ (VI $\overrightarrow{m}$ VP)

91

[3] Time: An Aside: We shall take a rather simplistic view of time [21, 65, 81, 98].

48. We consider d$\mathbb{T}$, or just $\mathbb{T}$, to stand for a totally ordered set of time points.

49. And we consider $\mathbb{TI}$ to stand for time intervals based on $\mathbb{T}$.

50. We postulate an infinitesimal small time interval δ.

51. $\mathbb{T}$, in our presentation, has lower and upper bounds.

52. We can compare times and we can compare time intervals.

53. And there are a number of "arithmetics-like" operations on times and time intervals.

92

[14] For VP see Item 12a on Page 22.

type
48. $\mathbb{T}$
49. $\mathbb{TI}$
value
50. δ:$\mathbb{TI}$
51. $\mathbb{MIN}, \mathbb{MAX}$: $\mathbb{T} \to \mathbb{T}$
51. $<,\leq,=,\geq,>$: $(\mathbb{T}\times\mathbb{T})|(\mathbb{TI}\times\mathbb{TI}) \to$ **Bool**
52. $-$: $\mathbb{T}\times\mathbb{T} \to \mathbb{TI}$
53. $+$: $\mathbb{T}\times\mathbb{TI},\mathbb{TI}\times\mathbb{T} \to \mathbb{T}$
53. $-,+$: $\mathbb{TI}\times\mathbb{TI} \to \mathbb{TI}$
53. $*$: $\mathbb{TI}\times$**Real** $\to \mathbb{TI}$
53. $/$: $\mathbb{TI}\times\mathbb{TI} \to$ **Real**

93

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

55. We declare a channel clk_ch.

value
54. clock: $\mathbb{T}$ → **out** clk_ch **Unit**
54. clock(t) ≡ ... clk_ch!t ... clock(t ⌈⌉ t+δ)
channnel
55. clk_ch:$\mathbb{T}$

2.8.2 Globally Observable Parts

94

There is given

56. a net, n:N,

57. a set of vehicles, vs:V-**set**, and

58. a monitor, m:M.

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

value
56. n:N = **obs_**N(Δ)
56. ls:L-**set** = **obs_**Ls(**obs_**LS(n)), hs:H-**set** = **obs_**Hs(**obs_**HS(n)),
56. lis:LI-**set** = {**uid_**L(l)|l:L•l ∈ ls}, his:HI-**set** = {**uid_**H(h)|h:H•h ∈ hs}
57. vs:V-**set** = **obs_**Vs(**obs_**VS(**obs_**F(Δ))), vis:V-**set** = {**uid_**V(v)|v:V•v ∈ vs}
58. m:**obs_**M(Δ)

2.8.3 Road Traffic System Behaviours 95

59. 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, based on
    - c the monitor and its unique identifier,
    - d an initial vehicle position map, and
    - e an initial starting time.

96

value
59c. mi:MI = **uid_**(m)
59d. vpm:VPM = vpr(vs)(n)
59e. t_0:T = clk_ch?

59. rts() =
59a. ‖ {veh(**uid_**V(v))(v)(vpm(**uid_**V(v)))|v:V•v ∈ vs}
59b. ‖ mon(mi)(m)([t_0 ↦ vpm])

where the "extra" monitor argument records the discrete road traffic, RTF, initially set to the singleton map from an initial start time, t_0 to the initial assignment of vehicle positions.

2.8.4 Channels 97

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.

60. Thus we declare a set of channels indexed by the unique identifiers of vehicles and communicating vehicle positions.

channel
60. {vm_ch[mi,vi]|vi:VI•vi ∈ vis}:VP

2.8.5 Behaviour Signatures 98

61. The road traffic system behaviour, rts, takes no arguments (hence the first **Unit**); and "behaves", that is, continues forever (hence the last **Unit**).
62. 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".
63. 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
61. rts: **Unit** → **Unit**
62. veh: vi:VI → v:V → VP → **out** vm_ch[vi],mi:MI **Unit**
63. mon: mi:MI → m:M → dRTF → **in** {vm_ch[mi,vi]|vi:VI•vi ∈ vis},clk_ch **Unit**

2.8.6 The Vehicle Behaviour 99

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

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

65. 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" !

101

65. veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
65a. vm_ch[mi,vi]!vp ; veh(vi)(v)(vp)
65b. ⌈⌉
65(b)i. **let** {hi′,thi}=**mereo_**L(get_L(tli)(n)) **in assert:** hi′=hi
65(b)ii. vm_ch[mi,vi]!onL(tli,hi,0,thi) ;
65(b)iii. veh(vi)(v)(onL(tli,hi,0,thi)) **end**
65c. ⌈⌉
65d. **stop**

102

66. 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.

67. or, internally non-deterministically,

68. the vehicle “disappears — off the radar” ! 103

```
64.        veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
66a.           vm_ch[mi,vi]!vp ; veh(vi)(v)(vp)
66b.       ⌈⌉
66c.           if f + δ<1
66(c)i.           then vm_ch[mi,vi]!onL(fhi,li,f+δ,thi) ;
66(c)i.                veh(vi)(v)(onL(fhi,li,f+δ,thi))
66(c)ii.          else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
66(c)iiA.              vm_ch[mi,vi]!atH(li,thi,li′);
66(c)iiB.              veh(vi)(v)(atH(li,thi,li′)) end end
67.         ⌈⌉
68.            stop
```

2.8.7 The Monitor Behaviour 104

69. 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”.

70. Either the monitor “does own work”

71. 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. 105

```
69.    mon(mi)(m)(rtf) ≡
70.        mon(mi)(own_mon_work(m))(rtf)
71.    ⌈⌉
71a.     ⌈⌉ { let ((vi,vp),t) = (vm_ch[mi,vi]?,clk_ch?) in
71b.          let rtf′ = rtf † [t ↦ rtf(max dom rtf) † [vi ↦ vp]] in
71c.          mon(mi)(m)(rtf′) end
71d.          end | vi:VI • vi ∈ vis }

70.    own_mon_work: M → dRTF → M
```

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

3 Domains 106

3.1 Delineations

We characterise a number of terms.

[1] Domain: By a domain$_\delta$ 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. 107

[2] Domain Phenomena: By a domain phenomenon$_\delta$ 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. 108

[3] Domain Entity: By a domain entity$_\delta$ we shall understand a manifest domain phenomenon or a domain 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 phenomena or from other domain concepts. 109

[4] Endurant Entity: We distinguish between endurants and perdurants.

From Wikipedia: *By an endurant$_\delta$ (also known as a continuant$_\delta$ or a substance$_\delta$) 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.* 110

[5] Perdurant Entity: 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 of running. Other examples include an activation, a kiss, or a procedure.* 111

[6] Discrete Endurant: We distinguish between discrete endurants and continuous endurants.

By a discrete endurant$_\delta$, that is, a part, we shall understand something which is separate or distinct in form or concept, consisting of distinct or separate parts. 112

[7] Continuous Endurant: By a continuous endurant$_\delta$, that is, a material, we shall understand an endurant whose spatial characteristics are prolonged, without interruption, in an unbroken spatial series or pattern. 113

[8] Domain Parts and Materials: By a part$_\delta$ we mean a discrete endurant, a manifest entity which is fixed in shape and extent. By a material$_\delta$ a continuous endurant, a manifest entity which typically varies in shape and extent. 114

[9] Domain Analysis: By domain analysis$_\delta$ we shall understand an examination of a domain, its entities, their possible composition, properties and relations between entities, 115

[10] Domain Description: By a domain description$_\delta$ we shall understand a narrative description tightly coupled (say line-number-by-line-number) to a formal description. 116

[11] Domain Engineering: By domain engineering$_\delta$ 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[15], etc. 117

[12] Domain Science: By domain science$_\delta$ 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. 118

[13] Values & Types: By a value$_\delta$ we mean some mathematical quantity. By a type$_\delta$ we mean a largest set of values, each characterised by the same predicate, such that there are no other values, not members of the set, but which still satisfy that predicate. We do not give examples here of the kind of type predicates that may characterise types. 119

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. Values are phenomena and types are concepts thereof. 120

[14] Discrete Perdurant: By a discrete perdurant$_\delta$ we shall understand a perdurant which we consider as taking place instantaneously, in no time, or where whatever time interval it may take to complete is considered immaterial. 121

[15] Continuous Perdurant: By a continuous perdurant$_\delta$ we shall understand a perdurant whose temporal characteristics are likewise prolonged, without interruption, in an unbroken temporal series or pattern. 122

[16] Extensionality: By extensionality$_\delta$ `Merriam-Webster`[16] means *"something which relates to, or is marked by extension," "that is, concerned with objective reality"*. Our use basically follows this characterisation: We think of extensionality as a syntactic notion, one that characterises an exterior appearance or form We shall therefore think of part types and material types whether parts are atomic or composite, and how composite parts are composed as extensional features. 123

[17] Intentionality: By intentionality$_\delta$ `Merriam-Webster`[17] means: *"done by intention or design", "intended", "of or relating to epistemological intention", "having external reference"*. Our use basically follows this characterisation: we think of intentionality as a semantic notion, one that characterises an intention. We shall therefore think of part attributess and material attributes as intentional features.

[15] Section 2 (Pages 17–35) is an example of the basis for a theory of road traffic systems.
[16] Extensionality. Merriam-Webster.com. 2011, http://www.merriam-webster.com (16 August 2012).
[17] Intentionality. Merriam-Webster.com. 2011, http://www.merriam-webster.com (16 August 2012).

3.2 Formal Analysis of Entities 124

3.2.1 Theory

This section is a transcription of Ganter & Wille's [38] *Formal Concept Analysis, Mathematical Foundations*, the 1999 edition, Pages 17–18.
125

Some Notation: By $\mathcal{E}$ we shall understand the type of entities; by $\mathbb{E}$ we shall understand a value of type $\mathcal{E}$; by $\mathcal{Q}$ we shall understand the type of qualities; by $\mathbb{Q}$ we shall understand a value of type $\mathcal{Q}$; by $\mathcal{E}$**-set** we shall understand the type of sets of entities; by $\mathbb{ES}$ we shall understand a value of type $\mathcal{E}$**-set**; by $\mathcal{Q}$**-set** we shall understand the type of sets of qualities; and by $\mathbb{QS}$ we shall understand a value of type $\mathcal{Q}$**-set**.
126

Definition: 1 Formal Context: A formal context$_\delta$ $\mathbb{K} := (\mathbb{ES}, \mathbb{I}, \mathbb{QS})$ consists of two sets; $\mathbb{ES}$ of entities, $\mathbb{QS}$ of qualities, and a relation $\mathbb{I}$ between $\mathbb{E}$ and $\mathbb{Q}$. ■

To express that $\mathbb{E}$ is in relation $\mathbb{I}$ to a Quality $\mathbb{Q}$ we write $\mathbb{E}\cdot\mathbb{I}\cdot\mathbb{Q}$, which we read as *"entity $\mathbb{E}$ **has** quality $\mathbb{Q}$"*. Example endurant entities are a specific vehicle, another specific vehicle, etcetera; a specific street segment (link), another street segment, etcetera; a specific road intersection (hub), another specific road intersection, etcetera, a monitor. One can also list perdurant entities. Example endurant entity qualities are has mobility, has possible velocity, has possible acceleration, has length, has location, has traffic state, can vehicles be sensed, etcetera. One can also list perdurant entity qualities.
127
128

Definition: 2 Qualities Common to a Set of Entities: For any subset, $s\mathbb{ES} \subseteq \mathbb{ES}$, of entities we can define

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

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

Definition: 3 Entities Common to a Set of Qualities: For any subset, $s\mathbb{QS} \subseteq \mathbb{QS}$, of qualities we can define

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

"the set of entities which have all qualities in $s\mathbb{Q}$". ■

129

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

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

Now comes the "crunch": *In the* `TripTych` *domain analysis we strive to find formal concepts*
130 *and, when we think we have found one, we assign a type to it.*

In mathematical terms it turns out that formal concepts are Galois connections. We can, in other words, characterise domain analysis to be the "hunting" for Galois connections. Or, even more "catchy": domain types, whether they be endurant entity types or they be perdurant entity signatures are Galois connections. 131

• • •

The entities referred to by $\mathbb{E}$ are the domain entities that we shall deal with in this paper, and the qualities referred to by $\mathbb{Q}$ are the mereologies and attributes of discrete endurant entities and the signatures of actions, events and behaviours of discrete perdurant entities; with these terms becoming clearer as we progress through this paper. 132

• • •

Earlier in this section, two signatures were expressed as $\mathcal{DQ}$: $\mathcal{E} \to \mathcal{K} \to \mathcal{Q}$ and $\mathcal{DE}$: $\mathcal{Q} \to \mathcal{K} \to \mathcal{E}$ The "switch" between using $\mathcal{K}$ for types and $\mathbb{K}$ for values of that type is "explained":

- $\mathcal{K}$ is the Cartesian type: $\mathcal{E} \times \mathcal{I} \times \mathcal{Q}$, and
- $\mathbb{K} = (\mathbb{E}, \mathbb{I}, \mathbb{Q})$ is a value of that type.

3.2.2 Practice 133

3.3 Discussion 134

The crucial characterisation is that of domain entity, see Sect. 3.1[3] (Page 36). It is pivotal since all we describe: narrate and formalise, are domain entities. If we get the characterisation wrong we get everything wrong! What might get the characterisation, or its interpretation, wrong is the interpretation of domain entities: *"those phenomena that can be observed by the human eye or touched, for example, by human hands,"* and *"manifest domain phenomena or domain concepts, i.e., abstractions, derived from a domain entities"*. 135

The whole thing hinges of *what can be described, what constitutes a description and when is a text a bona fide description.*

Another set of questions are *of what we have chosen to constitute entities which should we describe, which not ?* 136

Philosophers have dealt with these questions. Recent writings are [5, 90, 36] and [26, 61, 104]. Going back in time we find [62, 58, 27]. Among the classics we mention [85, 84, 24, 63].

137

We shall only indirectly contribute to this philosophical discussion and do so by presenting the material of this paper; having studied, over the years, fragments of the above cited publications we have concluded with the suggestions of this paper: following the principles, techniques and tools presented here can lead the domain engineer to a large class of domain descriptionss, large enough for our "immediate future" needs ! We shall, in the conclusion, return to the questions of what can be described, what constitutes a description and when is a text a bona fide description ?

4 Discrete Endurant Entities 138

For pragmatics reasons we structure our treatment of discrete endurant domain entities as follows: First we treat the extensional aspects of parts, then their properties: the intentional aspects. One could claim that when we say "first parts" we mean fist: a syntactic analysis of parts into atomic and composite parts, etcetera; and when we say "then their properties" we mean: then a partial semantic analysis, something which "throws" light over parts, since parts really are distinguishable only through their properties.

4.1 Parts 139

4.1.1 What is a Part ?

By a part$_\delta$ we mean an observable manifest endurant.

Discussion: We use the term 'part' where others use different terms, for example, 'indi-
140 vidual', 'object', 'particular', 'thing', 'unit', or other.

Example: 5 Parts. Example parts have their types defined in the items as follows: N Item 1a Page 17, F Item 1b Page 17, M Item 1c Page 17, HS Item 2a Page 17, LS Item 2b Page 17, VS Item 3 Page 17, Vs Item 4a Page 18, V Item 4b Page 18, Hs Item 5 Page 18, Ls Item 6 Page 18, H Item 5a Page 18, L Item 6b Page 18.

4.1.2 Classes of "Same Kind" Parts 141

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$_\delta$, 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
142 as a proposition.

Example: 6 Part Properties. We continue Example 4. 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). ■

4.1.3 A Preview of Part Properties 143

For pragmatic reasons we group endurant properties into two categories: a group which we shall refer to as meta properties: *is discrete*, *is continuous*, *is atomic*, *is composite*, *has observers*, *is sort* and *has concrete type*; and a group which we shall refer to as part properties *has unique existence*, *has mereology* and *has attributes*. The first group is treated in this section; the second group in Sect. 6.

4.1.4 Formal Concept Analysis: Endurants 144

We refer to Sect. **??** on Page ??: *Formal Concept Analysis*.

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 thoroughly 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 sloppiness in general 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 [52, M.A. Jackson: Lexicon ...].

We shall return later to a proper treatment of formal concept analysis [38].

4.1.5 Part Property Values 145

By a part property value$_\delta$, i.e., a property value$_\delta$ of a part, we mean the value associated with an intentional property of the part.

Example: 7 Part Property Values. A link, l:L, may have the following intentional property values: LOCation value *loc_set*, LENgth value *123 meters* and *mereology* value $\{\kappa_i, \kappa_j\}$. ■ 146

Two parts of the same type are different if for at least one of the intentional properties of that part type they have different part property values. slut

Example: 8 Distinct Parts. Two links, l_a,l_b:L, may have the following respective property values: LOCation values *loc_set*$_a$, and *loc_set*$_b$, LENgth value *123 meters* and *123 meters*, i.e., the same, and *mereology* values $\{\kappa_i, \kappa_j\}$ and $\{\kappa_m, \kappa_n\}$ where $\{\kappa_i, \kappa_j\} \neq \{\kappa_m, \kappa_n\}$. When so, they are distinct, and the cadestral space *loc_set*$_a$ must not share any point with cadestral space *loc_set*$_b$.

4.1.6 Part Sorts 147

By an abstract type$_\delta$, or a sort$_\delta$, we shall understand a type which has been given a name but is otherwise undefined, that is, is a set of values of further undefined quantities [72, 71]. where these are given properties which we may express in terms of axioms over sort (including property) values. All of the above examples are examples of sorts. 148

Example: 9 Part Sorts. The discovery of N, F and M was made as a result of examining the domain, Δ, at domain index $\langle\Delta\rangle$; HS and LS at domain index $\langle\Delta$,N$\rangle$; Hs and H (Ls and L) at domain indexes $\langle\Delta$,HS$\rangle$ ($\langle\Delta$,LS$\rangle$); and Vs and V at domain index $\langle\Delta$,VS$\rangle$. ■

4.1.7 Atomic Parts 149

By an atomic part$_\delta$ we mean a part which, in a given context, is deemed *not* to consist of meaningful, separately observable proper sub-parts. A sub-part is a part. 150

Example: 10 Atomic Types. We have exemplified the following atomic types: H (Item 5b on Page 18), L (Item 6b on Page 18), V (Item 4b on Page 18) and M (Item 1c on Page 17).

Implicit tests, at domain indexes, by the domain analyser, for atomicity were performed as follows: for H at $\langle\Delta$, N,HS,Hs,H$\rangle$; for L at $\langle\Delta$, N,LS,Ls,L$\rangle$; for V at $\langle\Delta$, F,VS,Vs,V$\rangle$; and for M at $\langle\Delta$, M$\rangle$. ■

4.1.8 Composite Parts 151

By a composite part$_\delta$ we mean *a part which, in a given context, is deemed to indeed consist of*
152 *meaningful, separately observable proper sub-parts.*

Example: 11 Composite Types. We have exemplified the following composite types: N (Items 2a– 2b on Page 17), HS (Item 5 on Page 18), LS (Item 6 on Page 18), Hs (Item 5a on Page 18), Ls (Item 6a on Page 18), F (Item 3 on Page 17), VS (Item 4a on Page 18), Va (Item 4a on Page 18), respectively. Tests for compostionality of these were implicitly performed; for N at index $\langle\Delta$, N$\rangle$; for HS and LS at index $\langle\Delta$, N,HS$\rangle$ and $\langle\Delta$, N,LS$\rangle$; for Hs and Ls at indexes $\langle\Delta$, N,HS,Hs$\rangle$ and $\langle\Delta$, N,LS,Ls$\rangle$; for F at index $\langle\Delta$, F$\rangle$; for VS at index $\langle\Delta$, F,VS$\rangle$; and for Vs at index $\langle\Delta$, F,VS,Vs$\rangle$. ■

4.1.9 Part Observers 153

By a part observer$_\delta$ or a material observer$_\delta$ we mean a meta-physical operator$_\delta$ (a meta function),

72. **obs_**B: P → B

that is, one performed by the domain analyser, which "applies" (i.e., who applies it) to a composite part value[18], P, and which yields the sub-part of type B, of the examined part. The **obs_** "keyword" prefix to a part type name B is intended to alert the reader to the fact that
154 **obs_**B is a meta function.

We name these **obs_**erver functions obs_X to indicate that they are observing parts of type X. The **obs_**erver functions are not computable. They can not be mechanised. Therefore we
155 refer to them as mental. They can be "implemented" as, for example, follows:

Example: 12 Implementation of Observer Functions. I take you around a particular road net, n,say in my town. I point out to you, one-by-one, all the street intersections, $h_1, h_2, \ldots, h_n$, of that net. You "write" them down: as many characteristics as you (and I) can come across, including some choice of unique identifiers, their mereologies, and attributes, "one-by-one". In the end we have identified, i.e., visited, all the hubs in my town's road net n. ■

156

Example: 13 Observer Functions. We have exemplified the following obs_erver functions: **obs_**N (Item 1a on Page 17), **obs_**F (Item 1b on Page 17), **obs_**M (Item 1c on Page 17), **obs_**HS (Item 2a on Page 17), **obs_**LS (Item 2b on Page 17), **obs_**VS (Item 3 on Page 17), **obs_**Vs (Item 4a on Page 18), **obs_**Hs (Item 5 on Page 18) and **obs_**Ls (Item 6 on Page 18), where we list their "definitions", not their many uses. ■

[18] or composite part type

4.1.10 Part Types 157

By a concrete type$_\delta$ 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.

Example: 14 Concrete Types. Example concrete part types were exemplified in Vs = V-**set**: Item 4a on Page 18, Hs = H-**set**: Item 5a Page 18, Ls = L-**set**: Item 6a Page 18. ■ 158

Example: 15 Has Composite Types. The discovery of concrete types were done as follows: for HS, Hs = H-**set** at ⟨Δ,N,HS⟩, for LS, Ls = L-**set** at ⟨Δ,N,LS⟩, and for VS, Vs = V-**set** at ⟨Δ,F,VS⟩. ■

4.2 Part Properties 159

(I) By a property[19] we mean a pair a (finite) collection of one or more propositions.

(II) By an endurant property a property which holds of an endurant — which we *model* as a *pair* of a type and a value (of that type)[20].

(III) By a perdurant property$_\delta$ we shall mean a property which holds of an perdurant — which we, as a minimum, *model* as a *pair* of a perdurant name and a function type, that is, as a function signature. 160

Property Value Scales: With intentional properties we associate a property value scale. By a property value scale$_\delta$ of a part type we shall mean a value range that parts of that type will have their property values range over.

Example: 16 Property Value Scales. We continue Example 4. (i) The mereology property value scale$_\delta$ for hubs of a net range over finite sets of link identifiers of that net. (ii)The mereology property value scale$_\delta$ for links of a net range over two element sets of hub identifiers for that net. (iii)The range of location values for any one hub of a net is restricted to not share any cadestral point with any other hub's location value for that net.

161

Discussion: The notion of 'property' is central to much philosophical discussion; we mention a few (that we have studied): [36, The Ontology of Language: Properties, Individuals and Discourse], [89, Parts: A Study in Ontology] and [67, Properties].[21] Their reading has influenced our work. 162

The notion of 'property' is also central to the recent notion of concept analysis [38, Formal Concept Analysis – Mathematical Foundations]. Here the term concept is understood as *a property of a part*. There is no associated type and value notions such as we have expressed

[19] By saying 'a property' we definitely mean to distinguish our use of the term from one which refers to legal property such as physical (land) or intangible (legal rights) property.

[20] The type value may be a singleton, or lie within a range of discrete values, or lie within a range of continuous values. The ranges may be finite or may be infinite.

[21] A reading of the contents listing of [67] reveals an interpretation of *parts and properties:*

I Function and Concept, Gottlob Frege
II The World of Universals, Bertrand Russell
III On our Knowledge of Universals, Bertrand Russell
IV Universals, F. P. Ramsey
V On What There Is, W. V. Quine
VI Statements about Universals, Frank Jackson
VII 'Ostrich Nominalism' or 'Mirage Realism', Michael Devitt
VIII Against 'Ostrich' Nominalism, D. M. Armstrong
IX On the Elements of Being: I, Donald C. Williams
X The Metaphysic of Abstract Particulars, Keith Campbell
XI Tropes, Chris Daly
XII Properties, D. M. Armstrong
XIII Modal Realism at Work: Properties, David Lewis
XIV New Work for a Theory of Universals, David Lewis
XV Causality and Properties, Sydney Shoemaker
XVI Properties and Predicates, D. H. Mellor.

in (II) on the previous page and Footnote 20 on the preceding page. We shall have more to say about the relations between our concept of domain analysis and Will & Ganter's concept analysis in Sect. ?? on Page ?? and in Item (iii) Sect. 9.1.11 on Page 104.

163 We shall now unravel our 'Property Theory'[22] of parts.

We see three categories of part properties: unique identifiers, mereology and (general) attributes.

Each and every part has unique existence — which we model through unique identifiers. Parts relate (somehow) to other parts, that is, mereology — which we model a relations between unique identifiers. And parts usually have other, additional properties which we shall refer to as attributes — which we model as pairs of attribute types and attribute values.

4.2.1 Unique Identifiers 164

Example: 17 Unique Identifier Functions. We have only exemplified the following unique identifier meta-functions and types: **uid_**H, HI Item 7a on Page 19, **uid_**L, LI Item 7b on Page 19 and **uid_**V, VI Item 7c on Page 19. We did not find a need for defining unique identifier meta-functions for N, F, M, HS, Hs, LS, Ls, VS, and Vs. ■

165

[1] A Dogma of Unique Existence: We take, as a dogma, that every two parts whose intentional property values differ for at least one property, other than their unique identifiers, 166 are distinct and thus have distinct unique identifiers.

[2] A Simplification on Specification of Intentional Properties: So we make a simplification in our treatment of intentional part properties By postulating distinct unique identifiers we are forcing distinctness of parts and can dispense with, that is, do not have to explicitly ascribe such intentional properties whose associated values would then have to differ in order 167 to guarantee distinctness of parts,

[3] Discussion: Parts have unique existence. Whether they be spatial or conceptual. Two manifest parts cannot overlap spatially. A part is a conceptual part if it is an abstraction of a part. Two conceptual parts are identical if they have identical properties, that is, abstract the same manifest part, otherwise they are distinct. We shall therefore associate with each part a unique identifier, whether we may need to refer to that property or not. There are only manifest parts and conceptual parts. The above deserves a whole separate inquiry. In defense 168 of the above, perhaps somewhat dogmatically phrased position, we refer to Russel's [86].

[4] The uid_P Operator: More specifically we postulate, for every part, p:P, a meta-function:

73. **uid_**P: P → Π

where Π is the type of the unique identifiers of parts p:P. The **uid_** "keyword" prefix to a 169 part type name P is intended to alert the reader to the fact that **uid_**P is a meta function. In practice we "construct" the unique identifier type name for parts of type P by "suffixing" I to P, and we explicitly "postulate define" the meta-function shown in Item 73. How is the uid_PI meta-function "implemented"? Well, for a domain description it suffices to postulate it. If we later were to develop software in support of the described domain, then there are 170 many ways of "implementing" the uid_PIs.

[22] — with apologies to [96, 97, 36].

[5] Constancy of Unique Identifiers — Some Dogmas: We postulate the following dogmas: parts may be "added" to or "removed" from a domain; parts that are "added" to a domain have unique identifiers that are not identifiers of any other part of the history of the domain; parts that are "removed" from a domain will not have their identifiers reused should parts subsequently be "added" to the domain; and domains do not allow for the changing (update) of unique identifier values.

4.2.2 Mereology 171

Mereology: By mereology$_\delta$ (Greek: $\mu\epsilon\rho o\varsigma$) we shall understand the study and knowledge about the theory of *part-hood* relations: of the relations of *part* to *whole* and the relations of *part* to *part* within a *whole*. 172

In the following please observe the type font distinctions: *part*, etc., and part (etc.).

In the above definition of the term mereology we have used the terms *part-hood, part* and *whole* in a more general sense than we use the term part. 173

In this the "more general sense" we interpret *part* to include, besides what the term part covers in this paper, also concepts, abstractions, derived from the concept of part. 174

That is, by *part* we mean not only manifest phenomena but also intangible phenomena that may be abstract models of parts, or may be (further) abstract models of *part*s.

Example: 18 Manifest and Conceptual Parts. We refer to Example 4. A net, n:N (Item 1a on Page 17), is a manifest part whereas a map, rm:RM (Item 26 on Page 24), is a *part*. ■ 175

[1] Extensional and Intentional Part Relations: Henceforth we shall "merge" the two terms *part* and part into one meaning.

So henceforth the term part shall refer to both manifest, tangible and discrete endurants and to abstractions of these. We are forced to do so by necessity. Instead of describing the 176 manifest phenomena we are describing conceptual models of these; that is, instead of describing manifest parts we are describing their part types and part properties. 177

Thus we choose "mereology" to model relations between both parts and *part*s. We can thus distinguish between two kinds of such relations: extensional part relations which typically are spatial relations between manifest parts and intentional part relations which typically are conceptual relations between abstract parts. 178

Extensional relations between manifest parts are of the kind: one part, p:P, is *"adjacent to"* (*"physically neighbouring"*) another part, q:Q, one part, p:P, is *"embedded within"* (*"physically surrounded by"*) another part, q:Q, and one part, p:P, *"overlaps with"* another part, q:Q.[23] We model these relations, "equivalently", as follows: in the mereology of p, **mereo_**P(p), there is a reference, **uid_**Q(q), to q, and in the mereology of q, **mereo_**Q(q), there is a reference, **uid_**P(p), to p. 179

Intentional relations between abstractions are of the kind: part p:P has an attribute whose value always stand in a certain relation (for example, a copy of a fragment or the whole) to another part q:Q's "corresponding" attribute value.

Example: 19 Shared Route Maps and Bus Time Tables. We continue and we extend Example 4. The 'Road Transport Domain' of Example 4 has its fleet of vehicles be that of

[23] The reader may wonder: How can a manifest physical part "overlap" another such part? We shall comment on this conundrum later in this paper. [Conundrum: a question or problem having only a conjectural answer.]

a metropolitan city's busses which ply some of the routes according to the city road map
180 (i.e., the net) and according to a bus time table — which we leave undefined. We can now re-interpret the road traffic monitor to represent a coordinating bus traffic authority, CBTA. CBTA is now the "new" monitor, i.e., is a part. Two of its attributes are: a metropolitan area road map and a metropolitan area bus time table Vehicles are now busses and each bus follows a route of the metropolitan area road map of which it has a copy, as a vehicle attribute, "shared" with CBTA; each bus additionally runs according to the metropolitan area bus time table of which it has a copy, as a vehicle attribute, "shared" with CBTA. ■

181 We model these attribute value relations, " equivalently", as above: in the mereology of p, **mereo_**P(p), there is a reference, **uid_**Q(q), to q, and in the mereology of q, **mereo_**Q(q), there is a reference, **uid_**P(p), to p.

Example: 20 Monitor and Vehicle Mereologies. We continue Example 19 on the previous page.

74. **value** **mereo_**M: VI-**set**
75. **type** MI
76. **value** **uid_**M: M → MI
77. **value** **mereo_**V: V → MI ■

182 **[2] Unique Part Identifier Mereologies:** To express a unique part identifier mereology assumes that the related parts have been endowed, say explicitly, with unique part identifiers., say of unique identifier types $\Pi_j, \Pi_k, \ldots, \Pi_\ell$. A mereology meta function is now postulated:

78. **value** **mereo_**P: P → ($\Pi_j \mid \Pi_k \mid \ldots \mid \Pi_\ell$)-**set**,

or of some such signature, one which applies to parts, p:P, and yields unique identifiers of other, "the related", parts — where these "other parts" can be of any part type, including P. The **mereo_** "keyword" prefix to a part type name P is intended to alert the reader to the
183 fact that **mereo_**P is a meta function.

Example: 21 Road Traffic System Mereology. We have exemplified unique part identifier mereologies for hubs, mereo_H Item 8a on Page 20 and links, mereo_L Item 9a on Page 20. ■

Example: 22 Pipeline Mereology. This is a somewhat lengthy example from a domain now
184 being exemplified. We start by narrating a pipeline domain of pipelines and pipeline units.

79. A pipeline consists of pipeline units.
80. A pipeline unit is either
    a a well unit output connected to a pipe or a pump unit;
    b a pipe, a pump or a valve unit input and output connected to two distinct pipeline units other than a well;
    c a fork unit input connected to a pipeline unit other than a well and output connected to two pipeline units other than wells and sinks;

d a join unit input connected to two pipeline units other than wells and output connected to a a pipeline unit other than a sink; and

e a sink unit input connected to a valve.

185

type
79. PL
value
79. **obs_**Us: PL → U-**set**
type
80. U = WeU | PiU | PuU | VaU | FoU | JoU | SiU
value
80. **uid_**U: U → UI
80. **mereo_**U: U → UI-**set** × UI-**set**
80. i_**mereo_**U,o_**mereo_**U: U → UI-**set**
80. i_UIs(u) ≡ **let** (ius,_) = **mereo_**U(u) **in** ius **end**
80. o_UIs(u) ≡ **let** (_,ous) = **mereo_**U(u) **in** ous **end**
axiom
∀ pl:PL,u:U • u ∈ **obs_**Us(pl) ⇒
80a. **is_**WeU(u) → **card** i_UIs(u)=0 ∧ **card** o_UIs(u)=1,
80b. (**is_**PiU|**is_**PuU|**is_**VaU)(u) → **card** i_UIs(u)=1=**card** o_UIs(u),
80c. **is_**FoU(u) → **card** i_UIs(u)=1 ∧ **card** o_UIs(u)=2,
80d. **is_**JoU(u) → **card** i_UIs(u)=2 ∧ **card** o_UIs(u)=1,
80e. **is_**SiU(u) → **card** i_UIs(u)=1 ∧ **card** o_UIs(u)=0

The UI "typed" **value** and **axiom** Items 80 "reveal" the mereology of pipelines. ■ 186

[3] Concrete Part Type Mereologies: Let A_i and B_j, for suitable i, j denote distinct part types and let B_jl Let there be the following concrete type definitions:

type
a_1:A_1 = bs:B_1-**set**
a_2:A_2 = bc:B_{2_1} × B_{2_2} × ... × B_{2_n}
a_3:A_3 = bl:$B_3{}^*$
a_4:A_4 = bm:BI_4 $\overrightarrow{m}$ B_4

The above part type definitions can be interpreted mereologically: Part a:A_1 has sub-parts b_{1_i},b_{1_2},...,b_{1_m}:B_1 of bs parthood related to just part a:A_1. Parts a:A_2 has sub-parts b_{2_1},b_{2_2},...,b_{2_m}:B_2 of bc parthood related only to parts a:A_1 Parts a:A_3 has sub-parts b_{3_i}, for all indices i of the list bℓ, parthood related to parts a:A_3, and to part $b_{3_{i-1}}$ and part $b_{3_{i+1}}$, for 1<i<**len** bℓ by being "neighbours" and also to other b_{3_j} if the index j is known to b_{3_i} for $i \neq j$. Parts a:A_4 have all parts bm(bi_j) for index bi_j in the definition set **dom** bm, be parthood related to a:A_4 and to other such bm:B_4 parts if they know their indexes. 187

Example: 23 A Container Line Mereology. This example brings yet another domain into consideration.

81. Two parts, sets of container vessels, CV-**set**, and sets of container terminal ports, CTP-**set**, are crucial to container lines, CL.
82. Crucial parts of container vessels and container terminal ports are their structures of *bays*, bs:BS.
83. A bay structure consists of an indexed set of *bays*.
84. A *bay* consists of an indexed set of *rows*
85. A *row* consists of an index set of *stacks*.
86. A *stack* consists of a linear sequence of *containers*.

188

type
81. CP, CVS, CTPS
value
81. **obs_**CVS: CL → CVS
81. **obs_**CTPS: CL → CTPS
type
81. CVS = CV-**set**
81. CTPS = CTP-**set**
value
82. **obs_**BS: (CV|CTP) → BS
type
83. BI, BS, B = BI $\overrightarrow{m}$ B
value
84. **obs_**RS: B → RS
type
84. RI, RS, R = RI $\overrightarrow{m}$ R
value
85. **obs_**SS: R → SS
type
85. SI, SS, C = SI $\overrightarrow{m}$ S
86. S = C*

189
190 In Fig. 1 on the facing page is shown a container line domain index lattice. At the top ("root") there is the container line domain type name. Immediately below it are the, in this case, two sub-domains (that we consider), CVS and CTPS. For each of these two there are the corresponding CV and CTP sun-domains. For each of these one can observe the container bays, hence, definition-wise, shared sub-domain. It is then defined in terms of a sequence of increasingly more "narrowly" defined sub-domains. The lattice "ends" with the atomic sub-domain of containers, C. ■

• • •

Discussion: Mereology is a discipline of study within both philosophy and logic. Mereology, in one form or another, has been studied, by philosophers, over the millennia, in 'Ancient Greece' (Plato, Aristotle), 'Roman Times' (Boethius), 'Medieval Ages' (Abelard, Aquinas)

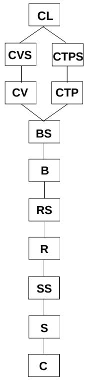


Figure 1: A container line domain index lattice

and in the 'Age of Enlightenment' (Kant), mereology became the subject also of a rigorous mathematical treatment, in the 1920s, by the Polish mathematician *Stanisław Leshniewski* [64, 70, 92]. Now it is also becoming a study within computer science [12, 16]. Modern study of mereology [102, 101, 25] treats it axiomatically. We shall, in contrast, suggest model-oriented descriptions of mereology. In [16] we indicate how a general model, $\mathcal{M}$, of mereology satisfies an axiomatic presentation, $\mathcal{A}$, a theory, that is, $\mathcal{M} \models \mathcal{A}$.

We present two classes of models of domain mereologies. One class of mereology models are based on the use of unique part identifiers. The other class of mereology models are based on concrete part type definitions. In either set of models the mereology that we shall express is about how a part is related to other parts and we "lightly" understand that relationship as a kind of connection: whether spatial connection in the form of a part, p, being either "somehow" *contained within* another, an "embracing" part, p', or "somehow" *adjacent to* another, a "neigbouring" part, p'; or conceptual connection in the form of properties of one part, p, being related to properties of one part, p, whether these properties be spatial or otherwise. 191

[4] Variability of Mereologies: The mereology of parts (of type P) may be a constant, i.e., static, or a variable, i.e., dynamic. That is, for some, or all, parts of a part type may need to be updated. We express the update of a part mereology as follows:

87. **value** **upd_mereo**_P: $(\Pi_i|\Pi_i|\ldots|\Pi_i)$-**set** → P → P

where **upd_mereo**_P($\{\pi_a, \pi_b, \ldots, \pi_c\}$)(p) results in a part p′:P where all part properties of p′ other than its mereology are as they "were" in p but the mereology of p′ is $\{\pi_a, \pi_b, \ldots, \pi_c\}$. 192

Example: 24 Insert Link. We continue Example 4, Item 42 on Page 30: In the post_link_dis predicate we referred to the undefined link insert function, ins_L. We now define that function:

88. The insert_Link action applies to a net, n, and a link, l,

89. and yields a new net, n′. 193

90. The conditions for a successful insertion are
    a that the link, l, is not in the links of net n,
    b that the unique identifier of l is not in the set of unique identifiers of the net n, and
    c that the mereology of link l has been prepared to be, i.e., is the two element set of unique identifiers of two hubs in net n.

91. The result of a successful insertion is
    a that the links of the new net, n′, are those of the previous net, n, "plus" link l;
    b that the hubs, "originally" h_a,h_b, connected by l, are only mereo-logically updated to each additional include the unique identifier of l; and
    c that all other hubs of n and n′ are unchanged.

194

```
88.  ins_L: N → L → N
89.  ins_L(n)(l) as n′
90.    pre:
90a.       l ∉ obs_Ls(obs_LS(n))
90b.     ∧ uid_L(l) ∉ in xtr_LIs(n)
90c.     ∧ mereo_L(l) ⊆ xtr_HIs(n)
91.    post:
91a.       obs_Ls(obs_LS(n′))=obs_Ls(obs_LS(n))∪{l}
91.      ∧ let {hi_a,hi_b}=mereo_L(l) in
91.        let {h_a,h_b}={get_H(hi_a)(n),get_H(hi_b)(n)} in
91b.       get_H(hi_a)(n′)=upd_mereo_H(h_a)(mereo_H(h_a)∪{uid_L(l)})
91b.     ∧ get_H(hi_b)(n′)=upd_mereo_H(h_b)(mereo_H(h_b)∪{uid_L(l)})
91c.     ∧ obs_Hs(obs_HS(n))=obs_Hs(obs_HS(n))\{hi_a,hi_b}∪{h_a′,h_b′} end end
```

195 As for the very many other function definitions in this paper we illustrate one form of function definition annotations, and not always consistently the same "style". We do not pretend that our function definitions are novel, let alone a contribution of this paper; instead we rely on the reader having learnt, more laboriously than we this paper can muster, an appropriate function definition narrative style. ■

• • •

This point in this paper may also be an appropriate one for briefly discussing another aspect the form of of formal function definitions. Even to us, even though we certainly do not always adhere to this *desiderata,* a function definition ought be formulated in a few lines: 2–3, at most 4. If, as above, we do not achieve that, in a "first attempt",[24] then the developer ought split that function definition into several such. To do so often amounts to the separate development of *a domain theory:* a number of more-or-less "ultra-short" definitions and their repeated re-use in many contexts while also developing a number of theorems based also on axioms of that *domain theory*.

[24]We refer to some such "not too tersely expressed" function definitions: wf_RM Item 26 on Page 24 (where we suggest that the three line Item 26b become the body of an auxiliary predicate), and, notably, the above ins_L Item 88 on the previous page.

4.2.3 Attributes 196

Attribute: By a part attribute$_\delta$ we mean a part property other than part unique identifier and part mereology, and its associated attribute property value.

Example: 25 Road Transport System Part Attributes. We have exemplified, Example 4, a number of part attribute observation functions: attr_LΣ Item 10a on Page 20, attr_LΩ Item 10b on Page 20, attr_LOC, attr_LEN Item 10c on Page 20, attr_HΣ Item 11a on Page 21, attr_HΩ Item 11b on Page 21, attr_LOC Item 11c on Page 21, attr_VP, attr_onL, attr_atH, attr_VEL and attr_ACC Item 13 on Page 22. ■ 197

[1] Stages of Attribute Analysis: There are four facets to deciding upon part attributes: (i) determining on which attributes to focus; (ii) selecting appropriate attribute type names, (**viz.**, LΣ, LΩ, HΣ, HΩ, LEN, LOC, VP, atH, onL, VEL and ACC from the above example); (iii) determining whether an attribute type is a static attribute type (having constant value) (**viz.**, LEN, LOC), or a dynamic attribute type (having variable values)) (**viz.**, LΣ, LΩ, HΣ, HΩ, VP, atH, onL, VEL, ACC); and (iv) deciding upon possible concrete type definitions for (some of) those attribute types (**viz.**, LΣ, LΩ, HΣ, HΩ, VP, atH, onL). 198

Example: 26 Static and Dynamic Attributes. Continuing Example 4 we have: Dynamic attributes: LΣ Item 10a on Page 20; HΣ Item 11a on Page 21; VP, atH, onL Items 12a–12(a)ii on Page 22; and VEL and ACC both Item 13 on Page 22. All other attributes are considered static. ■ 199

Example: 27 Concrete Attribute Types. From Example 4: LΣ=(HI×HI) Item 10a on Page 20, LΩ=LΣ-**set** Item 10b on Page 20, HΣ=(LI×LI)-**set** Item 11a on Page 21 and HΩ=HΣ-**set** Item 11b on Page 21. ■ 200

[2] The attr_A Operator: To observe a part attribute we therefore describe the attribute observer signature

92. **attr_**A: P → A,

where P is the part type being examined for attributes, and A is one of the chosen attribute type names. The **attr_** "keyword" prefix to an attribute type name A is intended to alert the reader to the fact that **attr_**A is a meta function. The "hunt" for part attributes, i.e., attribute types, the resulting attribute function signatures and the chosen concrete attribute types is crucial for achieving successful domain descriptions. 201

[3] Variability of Attributes: Static attributes are constants. Dynamic attributes are variables. To express the update of any one specific dynamic attributevalue we use the meta-operator:

93. **value** **upd_attr_**A: A → P → P

where **upd_attr_**A(a)(p) results in a part p′:P where all part properties of p′ other than its the attribute value for attribute A are as they "were" in p but the attribute value for attribute A is a. The **upd_attr_** "keyword" prefix to an attribute type name A is intended to alert the reader to the fact that **upd_attr_**A is a meta function. 202

Example: 28 Setting Road Intersection Traffic Lights. We refer to Example 4, Items 11a (HΣ) and 11b (HΩ) on Page 21. The intent of the hub state model (a hub state as a set of pairs of unique link identifiers) is that it expresses the possibly empty set of allowed hub traversals, from a link incident upon the hub to a link emanating from that hub. 203

94. In order to "change" a hub state the set_hub_state action is performed,
95. It takes a hub and a hub state and yields a changed hub.
    The argument hub state must be in the state space of the hub.
    The result of setting the hub state is that the resulting hub has the argument state as its (updated) hub state.

value
94. set_hub_state: H → HΣ → H
95. set_hub_state(h)(hσ) ≡ **upd_attr_**HΣ(h)(hσ)
95. **pre**: hσ ∈ **attr_**HΩ(h)

The hub state has not changed if **attr_**HΣ(h) = hσ. ■

4.2.4 Properties and Concepts 204

Some remarks are in order.

[1] Inviolability of Part Properties: Given any part p of type P one cannot "remove" any one of its properties and still expect the the part to be of type P. Properties are what "makes" parts. To put the above remark in "context" let us review Ganter & Wille's formal concept analysis [38].

205 **[2] Ganter & Wille: Formal Concept Analysis:** This review is based on [38].

TO BE WRITTEN

206 **[3] The Extensionality of Part Attributes:**

TO BE WRITTEN

4.2.5 Properties of Parts 207

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

208 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 language sub-type and **nil** denotes the empty type.

4.3 States 209

By a state$_\delta$ we mean a collection of such parts some of whose part attribute values are dynamic, that is, can vary.

Example: 29 A Variety of Road Traffic Domain States. We continue Example 4. A link, l:L, constitutes a state by virtue of if its link traffic state lσ:**attr_**LΣ. A hub, h:H, constitutes a state by virtue of its hub traffic state hσ:**attr_**HΣ, and indepenently, its hub mereology lis:LI-**set**:**mereo_**H. A net, n:N, constitutes a state by virtue of if its link and hub states. A monitor, m:M, constitutes a state by virtue of if its vehicle position map vpm:**attr_**VPM. ■

4.4 An Example Domain: Pipelines 210

We close Sect. 4 with a "second main example", albeit "smaller", in text size, than Example 4. The domain is that of pipelines. The reason we bring this example is the following: Not all domain endurants are discrete domain endurants. Some domains possess continuous domain endurants. We shall call them materials. Two such materials are liquids, like `oil` (or `petroleum`), and gaseous, like `natural gas`. The description of such, as we shall later call 211
them, materials-based domains requires additional description concepts and new description techniques. The examples of this subsection, i.e., Sect. 4.4 illustrates these new concepts and techniques as do the examples of Sect. 6.1. 212

Example: 30 Pipeline Units and Their Mereology.

96. A pipeline consists of connected units, u:U.
97. Units have unique identifiers.
98. And units have mereologies, ui:UI:
    a pump[25], pu:Pu, pipe, pi:Pi, and valve[26], 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[27], we:We, [sink[28], 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.

213

[25] We abstract from such distinctions between *oil pipeline pumps* and *gas pipeline compressors*.

[26] 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.

[27] We abstract wells into initial injection stations where the liquid or gaseous material is injected into the line.

[28] We abstract partial and final delivery stations into sinks, places where the material is delivered to an agent outside the pipeline system.

type
96. U = Pu | Pi | Va | Fo | Jo | Si | We
97. UI
value
97. uid_U: U → UI
98. mereo_U: U → UI-**set** × UI-**set**
98. wf_mereo_U: U → **Bool**
98. wf_mereo_U(u) ≡
98a. **let** (iuis,ouis) = **mereo_**U(u) **in**
98a. is_(Pu|Pi|Va)(u) → **card** iusi = 1 = **card** ouis,
98b. is_Fo(u) → **card** iuis = 1 ∧ **card** ouis = 2,
98b. is_Jo(u) → **card** iuis = 2 ∧ **card** ouis = 1,
98c. is_We(u) → **card** iuis = 0 ∧ **card** ouis = 1,
98d. is_Si(u) → **card** iuis = 1 ∧ **card** ouis = 0 **end**
98e. sel_UIs_in: U → UI-**set**
98e. sel_UIs_in(u) ≡ **let** (iuis,_)=**mereo_**U(u) **in** iuis **end**
98f. sel_UIs_out: U → UI-**set**
98f. sel_UIs_out(u) ≡ **let** (_,ouis)=**mereo_**U(u) **in** ouis **end**

214

Example: 31 Pipelines: Nets and Routes.

99. A pipeline net consists of several properly connected pipeline units.

    Example 30 on the preceding page already described pipeline units.

    Here we shall concentrate on their connectedness, i.e., the wellformednes of pipeline nets.

100. A pipeline net is well-formed if
    a all routes of the net are acyclic, and
    b there are a non-empty set of well-to-sink routes that connect any well to some sink, and
    c all other routes of the net are embedded in the well-to-sink routes

215

type
99. PLN′
99. PLN = {| pln:PLN′ • **is_wf_**PLN(pln) |}
value
99. **obs_**Us: PLN → U-**set**
100. **is_wf_**PLN: PLN′ → **Bool**
100. **is_wf_**PLN(pln) ≡
100. **let** rs = routes{pln} **in**
100b. well_to_sink_routes(pln)≠{}
100c. ∧ embedded_routes(pln) **end**

216

101. An acyclic route is a route where any element occurs at most once.

102. A well-to-sink route of a net, pln, is a route whose first element designates a well in pln and whose last element designates a sink in pln.

103. One non-empty route, r′, is embedded in another route, r if the latter can be expressed as the concatenation of three routes: $r = r''{}^\frown r'{}^\frown r'''$ where r″ or r‴ may be empty routes ($\langle\rangle$). 217

type
105. $R' = UI^*$
100a. $R = \{r:R' \bullet is\_acyclic(r)\}$
value
100a. is_acyclic: R → **Bool**
100a. is_acyclic(r) ≡ ∀ i,j:**Nat**•i≠j∧{i,j}⊆**inds** r⇒r[i]≠r[j]

100b. well_to_sink_routes: PLN → R-**set**
100b. well_to_sink_routes(pln) ≡
100b. {r|r:R•r ∈ routes(pln) ∧ ∃ we:WE,si:Si •
100b. {we,si}⊆**obs_**Us(pln) ⇒ r[1]=we ∧ r[**len** r]=si} 218

104. One non-empty route, er, is_embedded in another route, r,
   a if there are two indices, i, j, into r
   b such that the sequence of r elements from and including i to and including j is er.

value
104. is_embedded: R × R → **Bool**
104. is_embedded(er,r) ≡
104a. ∃ i,j:**Nat**•{i,j}⊆**inds** r
104b. ⇒ er = ⟨r[k]|k:**Nat** • i≤k≤j⟩
104. **pre**: er≠⟨⟩ 219

105. A route, r, of a pipeline net is a sequence of unique unit identifiers, satisfying the following properties:
   a if r[i]=ui_i has ui_i designate a unit, u, of the pipeline then $\langle ui_i\rangle$ is a route of the net;
   b if $r_i{}^\frown\langle ui_i\rangle$ and $\langle ui_j\rangle{}^\frown r_j$ are routes of the net
      i. where u_i and u_j are the units (of the net) designated by ui_i and ui_j
      ii. and ui_j is in the output mereology of u_i and ui_i is in the input mereology of u_j
      iii. then $r_i{}^\frown\langle ui_i\rangle{}^\frown\langle ui_j\rangle{}^\frown r_j$ is a route of the net.
   c Only such routes that can be constructed by a finite number of "applications" of Items 105a and 105b are routes. 220

105. routes: PLN → R-**set**
105. routes(pln) ≡
105a. **let** rs = {⟨**uid_**UI(u)⟩|u:U•u ∈ **obs_**Us(pln)}
105(b)iii. ∪ { $r_i{}^\frown\langle ui_i\rangle{}^\frown\langle ui_j\rangle{}^\frown r_j$
105b. | $r_i{}^\frown\langle ui_i\rangle,\langle ui_j\rangle{}^\frown r_i$:R • {$r_i{}^\frown\langle ui_i\rangle,\langle ui_j\rangle{}^\frown r_j$}⊆rs
105(b)i. ∧ **let** u_i,u_j:U•{u_i,u_i}⊆**obs_**Us(pln)∧ui_i=**uid_**U(u_i)∧ui_j=**uid_**U(u_j)
105(b)ii. **in** ui_i ∈ iuis(u_j) ∧ ui_j ∈ ouis(u_i) **end** }
105c. **in** rs **end**

Section 6.1 will continue with several examples (Example 44 on Page 70, Example 45 on Page 70, Example 46 on Page 71, Example 47 on Page 72 and Example 48 on Page 73) following up on the two examples of this section.

5 Discrete Perdurant Entities 221

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 of running. Other examples include an activation, a kiss, or a procedure.*

A discrete perdurant$_\delta$ is a perdurant which is a discrete entity. We shall consider the 222
following discrete perdurants. actions (Sect. 5.2), events (Sect. 36), and discrete behaviours (Sect. 5.4).

Actions and events occur instantaneously, that is, in time, but taking no time, and to therefore be discrete action$_\delta$s and discrete event$_\delta$s.

5.1 Formal Concept Analysis: Discrete Perdurants 223

We refer to Sect. ?? on Page ??: *Formal Concept Analysis*.

The domain analyser examines collections of discrete perdurants. (i) In doing so the domain analyser discovers and thus identifies and lists a number of perdurant properties. (ii) Each of the discrete perdurants examined usually satisfies only a subset of these properties. (iii) The domain analyser now groups discrete perdurant into collections such that each collection have its discrete perdurants 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 discrete perdurants are put in some collection. (iv) The domain analyser now classify collections as actions, events or behaviours, and assign signatures to distinct collections. That is how we assign signatures to discrete perdurants.

5.2 Actions 224

By a function$_\delta$ we understand a mathematical concept, a thing which when applied to a value, called its argument, yields a value, called its result. A discrete action$_\delta$ can be understood as a function invoked on a state value and is one that potentially changes that value. Other terms for action are function invocation$_\delta$ and function application$_\delta$. 225

Example: 32 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. ■

5.2.1 Abstraction: On Modelling Domain Actions 226

We claim that we describe domain actions, but we actually describe functions, which are "somewhat far removed" from domains. So what are we actually claiming? We are claiming that there is an interesting class of actions and that they can all be abstracted into one,

possibly non-deterministic function whose properties are then claimed to "mimic" those of the actions in the interesting class.

5.2.2 Agents: An Aside on Actions 227

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'" [31, 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. 5.4. Agents will here, for
228 simplicity, be considered behaviours, and are treated in Sect. 5.4. As to the relation between intention and action 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.

5.2.3 Action Signatures 229

By an action signature we understand a quadruple: a function name, a function definition set type expression, a total or partial function designator (→, respectively $\stackrel{\sim}{\to}$), and a function image set type expression: fct_name: A → Σ (→|$\stackrel{\sim}{\to}$) Σ [× R], where $(X \mid Y)$ means either X or Y, and $[Z]$ means that for some signatures there may be a Z component meaning that the
230 action also has the effect of "leaving" a type Z value.[29]

Example: 33 Action Signatures: Nets and Vessels.

insert_Hub: N→H$\stackrel{\sim}{\to}$N;
remove_Hub: N→HI$\stackrel{\sim}{\to}$N;
set_Hub_Signal: N→HI$\stackrel{\sim}{\to}$HΣ$\stackrel{\sim}{\to}$N
load_Container: V→C→StackId$\stackrel{\sim}{\to}$V; and
unload_Container: V→StackId$\stackrel{\sim}{\to}$(V×C). ■

5.2.4 Action Definitions 231

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.

232

Example: 34 Transport Nets Actions. In Example 4 we gave an explicit example of an action: ins_H: Items 37–37d, while implicit references to net actions were made in the event predicates link_dis, pre_link_dis: Items 38–39c, post_link_dis (Items 38–39c): rem_L Item 42a and ins_L Items 42(c)i–42(c)ii. ■

233 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: 35 Container Line: Remove Container. We refer to Example 23 (Pages 47–48).

106. 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 106d).
   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.
107. 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.
108. 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
106. remove_C_from_V: V → StackId $\xrightarrow{\sim}$ (V×C)
106. remove_C_from_V(v)(stid) **as** (v′,c)
106c. (**obs_**Bs(**obs_**BS(v′),c)) = remove_C_from_BS(**obs_**Bs(**obs_**BS(v)))(stid)
106d. ∧ props(v)=props(v″)

106b. remove_C_from_BS: BS → StackId → (BS×C)
106a. remove_C_from_BS(bs)(stid) **as** (bs′,c)
107a. **pre**: valid_address(bs)(stid)

[29] We shall not here speculate on "what happens" to that resulting value.

107b. ∧ non_empty_designated_stack(bs)(stid)
108a. **post**: c = get_C(bs)(stid)
108b. ∧ unchanged_mereology(bs,bs′)
108c. ∧ popped_designated_stack(bs,bs′)(stid)
108d. ∧ unchanged_non_designated_stacks(bs,bs′)(stid)

The props function was introduced in Sect. 4.2.5 on Page 52.

This example hints at *a theory of container vessel bays, rows and stacks*. More on that is found in Appendix B. There you will find explanations of the valid_address (Item 202 on Page 122), non_empty_designated_stack (Item 203), unchanged_mereology (Item 204), popped_designated_stack (Item 205) and unchanged_non_designated_stacks (Item 206) functions. ■

234 There are other ways of defining functions. But the form of these are not germane to the aims of this paper.

Modelling Actions

- We refer to Sect. 5.1: Formal Concept Analysis of Discrete Perdurants on Page 57.
- 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"* [31].
  - 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.

235

- 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.

236

- Sooner or later the domain describer must decide on the function definition.
  - The form[30] 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.

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

5.3 Events 237

By an event$_\delta$ 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". 238

Example: 36 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'. *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'.

5.3.1 An Aside on Events 239

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 paper we therefore do not ascribe time points or time intervals with the occurrences of events[31].

5.3.2 Event Signatures 240

An event signature$_\delta$ *is a predicate signature having an event name (evt), 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**.

5.3.3 Event Definitions 241

An event definition$_\delta$ 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):* $evt(\sigma,\sigma') = \exists\ (ev{:}ET) \bullet pre\_evt(ev)(\sigma) \Rightarrow post\_evt(ev)(\sigma,\sigma')$.

There may be variations to the above form. 242

Example: 37 Road Transport System Event. Example 4, Sect. 2.7, Items 38–42(c)ii (Pages 29–30) exemplified an event definition.

243

Modelling Events

- We refer to Sect. 5.1: Formal Concept Analysis of Discrete Perdurants on Page 57.
- 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"* [31].
  - 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.

244

- 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.

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

5.4 Discrete Behaviours 245

We shall distinguish between discrete behaviours (this section) and continuous behaviours (Sect. 6.2). Roughly discrete behaviours proceed in discrete (time) steps — where, in this section, 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
246 the domain during that time (interval). Hence the behaviour is considered discrete.

Continuous behaviours are continuous in the sense of the calculus of mathematical analysis; to qualify as a continuous behaviour time must be an essential aspect of the behaviour.

Discrete behaviours can be modelled in many ways, for example using `CSP` [45]. `MSC` [49], `Petri Nets` [82] and `Statechart` [42]. We refer to Chaps. 12–14 of [9]. In this paper we shall use `RSL/CSP`.

5.4.1 What is Meant by 'Behaviour'? 247

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$_\delta$ we understand a set of sequences
248 of actions, events and behaviours.

A "slanted" characterisation runs as follows: by a behaviour$_\delta$ we shall understand either a sequential behaviour$_\delta$ consisting of a possibly infinite sequence of zero or more actions and events; or one or more communicating behaviour$_\delta$s whose output actions of one behaviour may synchronise and communicate with input actions of another behaviour; or two or more behaviours acting either as internal non-deterministic behaviour_δs (⌈⌉) or as external non-deterministic behaviour_δs ([]). 249

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.

5.4.2 Behaviour Narratives 250

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 was done in Example 4, 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.

5.4.3 Channels 251

We remind the reader that we are focusing exclusively on domain behaviours. Domain behaviours, as we shall see in Sect. 5.4.6, take their "root" in parts. We shall find, even when "parts" take the form of concepts, that these do not "overlap". They may share properties, but we can consider them "disjoint".[32] Hence communication between processes can be thought of as communication between "disjoint parts", and, as such, can be abstracted as taking place in a non-physical medium which we shall refer to as channels. 252

By a channel_δ we shall understand *a means of communicating entities between [two] behaviours*.

To express channel communications we, at present, make use of RSL [39]'s **out**put (ch ! v) / **in**put (ch ?) clauses and **channel** declarations,

type M
channel ch M,
value ch!v, ch?,

Variations of the above clauses are

type ChIdx, ChJdx
channel $\{ch[i]|i{:}ChIdx \bullet \mathcal{P}(i,...)\}$:M, $\{ch[i,j]|i{:}ChIdx,j{:}ChJdx \bullet \mathcal{P}(i,j,...)\}$:M
value ch[i]!v, ch[i]?, ch[i,j]!v, ch[i,j]?

where $\mathcal{P}$ is a suitable predicate over channel indices and possibly global domain values.

5.4.4 Behaviour Signatures 253

By a $\text{behaviour signature}_\delta$ we shall understand *a a function signature augmented by a clause which declares the* **in** *channels on which the function accepts inputs and the* **out** *channels on which the function offers output.*

[32] These previous sentences really beg more careful, at times philosophical arguments. Once this present, and at present, excluding Sect. 8, 90 page document, has found a reasonably stable form (after now 4–5 iterations, we plan to separate out a number of the places, such as this, which warrant careful motivations.

value behaviour: A → **in** in_chs **out** out_chs B

where (i) the form **in** in_chs **out** out_chs may be just **in** in_chs or **out** out_chs or both **in** in_chs
254 **out** out_chs that is, behaviour accepts input(s), or offers output(s), or both; where (ii) A typically is of the forms **Unit** if the behaviour "takes no arguments", that is: behaviour(), or PI×P
255 if the behavior is directly based on a part, p:P, for that is: behaviour(uid_P(p),p); where (iii) in_chs and out_chs are of the form either ch, or $\{ch[i]|i{:}ChIdx \bullet \mathcal{Q}(i,...)\}$ or $\{ch[i,j]|i{:}ChJdx,j{:}ChJdx \bullet \mathcal{R}(i,j,...)\}$, $\mathcal{Q}$, $\mathcal{R}$ are appropriate predicates; and where (iv) either B is either just **Unit** when the behaviour is typically a never-ending (i.e., cyclic) behaviours, or is some result type C.

5.4.5 Behaviour Definitions 256

This section is about the basic form of behaviour function definitions. We shall only be concerned with behaviours which define part behaviours.

By a $\text{part behaviour}_\delta$ we shall understand *a behaviour whose state is that of the part for which it is the behaviour.*

There are basically two cases for which we are interested in the form of the behaviour
257 definition: (i) the atomic part behaviour, and (ii) the composite part behaviour.

[1] Atomic Part Behaviours: Let p:P be an atomic part of type P. Then the basic form of a cyclic atomic behaviour definition is

value
atomic_core_part_behaviour(uid_P(p))(p) ≡
  let p′ = $\mathcal{A}$(uid_P(p))(p) **in**
  atomic_core_part_behaviour(uid_P(p))(p′) **end**
  post: uid_P(p) = uid_P(p′),

$\mathcal{A}$: PI → P → **in** ... **out** ... P,

where $\mathcal{A}$ usually is a terminating function which synchronises and communicates with other
258 part behaviours.

Example: 38 Atomic Part Behaviours. Example 4, Sect. 2.8.6 on Page 34 and Sect. 2.8.7 on Page 35 illustrates cyclic atomic behaviours: vehicle at Hub: Items 65–65d, on Page 34,
259 vehicle on Link: Items 64–68, on Page 35 and monitor: Items 69–71d, on Page 35. ■

[2] Composite Part Behaviours: Let p:P be an atomic part of type P. Then the basic form of a cyclic atomic behaviour definition is

value
composite_part_behaviour(uid_P(p))(p) ≡
  composite_core_part_behaviour(uid_P(p))(p)
  ∥ { part_behaviour(uid_P(p′))(p′)|p′:P•p′ ∈ **obs_**(p)}

core_part_behaviour: PI → P → **in** ... **out** ... **Unit**
core_part_behaviour(uid_P(p))(p) ≡
  let p′ = $\mathcal{C}$(uid_P(p))(p) **in**
  composite_core_part_behaviour(uid_P(p))(p′) **end**

post: uid_P(p) = uid_P(p′)

$\mathcal{C}$: PI → P → **in** ... **out** ... P,

where $\mathcal{C}$ usually is a terminating function which synchronises and communicates with other part behaviours. 260

Example: 39 Compositional Behaviours. Example 4, Sect. 2.8.3 on Page 33 illustrated compositionality, cf. Items 59– 59b on Page 33. ■

The next section illustrates the basic principles that we recommend when modelling behaviours of domains consisting of composite and atomic parts.

5.4.6 A Model of Parts and Behaviours 261

How often have you not "confused", linguistically, 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'. 262

The example additionally has a more general aim, *namely that of showing that to every mereology (or parts) there is a λ-expression here in the form of basically a CSP [45] program.* 263

Example: 40 Syntax and Semantics of Mereology. 264

[1] A Syntactic Model of Parts:

109. The *whole* contains a set of *parts*.
110. *Parts* are either *atomic* or *composite*.
111. From *composite parts* one can observe a set of *parts*.
112. All *parts* have *unique identifiers*

265

type
109. W, P, A, C
110. P = A | C
value
111. <u>**obs_**</u>Ps: (W|C) → P-**set**
type
112. PI
value
112. <u>**uid_**</u>Π: P → Π

266

113. From a *whole* and from any *part* of that *whole* we can e**xtr**act all contained *parts*.
114. Similarly one can e**xtr**act the *unique identifiers* of all those contained *parts*.

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

267

value
113. xtr_Ps: (W|P) → P-**set**
113. xtr_Ps(w) ≡ {xtr_Ps(p)|p:P•p ∈ <u>**obs_**</u>Ps(p)}
113.   **pre**: is_W(p)
113. xtr_Ps(p) ≡ {xtr_Ps(p)|p:C•p∈ <u>**obs_**</u>Ps(p)}∪{p}
113.   **pre**: is_P(p)
114. xtr_Πs: (W|P) → Π-**set**
114. xtr_Πs(wop) ≡ {<u>**uid_**</u>P(p)|p ∈ xtr_Ps(wop)}
115. <u>**mereo_**</u>P: P → Π-**set**
axiom
116. ∀ w:W
116.   **let** ps = xtr_Ps(w) **in**
116.   ∀ p:P • p ∈ ps • ∀ π:Π • π ∈ <u>**mereo_**</u>P(p) ⇒ π ∈ xtr_Πs(p) **end**

268

117. An attribute map of a *part* associates with *attribute names*, i.e., *type names*, their *values*, whatever they are.
118. From a *part* one can extract its attribute map.
119. Two *parts share attributes* if their respective attribute maps share *attribute names*.
120. 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.

269

type
117. AttrNm, AttrVAL,
117. AttrMap = AttrNm $\overrightarrow{m}$ AttrVAL
value
118. <u>**attr_**</u>AttrMap: P → AttrMap
119. share_Attributes: P×P → **Bool**
119. share_Attributes(p,p′) ≡
119.   **dom** <u>**attr_**</u>AttrMap(p) ∩
119.   **dom** <u>**attr_**</u>AttrMap(p′) ≠ {}
120. share_Properties: P×P → **Bool**
120. share_Properties(p,p′) ≡
120a.   share_Attributes(p,p′)
120b. ∨ <u>**uid_**</u>P(p) ∈ <u>**mereo_**</u>P(p′)
120b. ∨ <u>**uid_**</u>P(p′) ∈ <u>**mereo_**</u>P(p)

270

[2] A Semantics Model of Parts:

121. 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.
122. Let there be given a 'whole', w:W.
123. 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.

271

type
121. K = Π-**set axiom** ∀ k:K•**card** k=2
value
121. xtr_Ks: (W|P) → K-**set**
121. xtr_Ks(wop) ≡
121. **let** ps = xtr_Ps(w) **in**
121. {{**uid_**P(p),π}|p:P,π:Π•p∈ ps ∧ ∃ p′:P•p′≠p∧π=**uid_**P(p′) ∧ **uid_**P(p)∈uid_P(p′)} **end**

122. w:W

123. **channel** {ch[k]|k:xtr_Ks(w)}:M

272

124. Now the 'whole' *behaviour* whole is the parallel composition of *part processes*, one for each of the immediate parts of the *whole*.
125. 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*.

273

124. whole: W → **Unit**
124. whole(w) ≡ ‖ {part(**uid_**P(p))(p) | p:P•p ∈ xtr_Ps(w)}

125. part: π:Π → P → **Unit**
125. part(π)(p) ≡
125a. is_A(p) → atom(π)(p),
125b. _ → comp(π)(p)

274

126. 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.

.

value
126. comp: π:Π → p:P → **in**,**out** {ch[{π,π′}|{π′∈ **mereo_**P(p)}]} **Unit**
126. comp(π)(p) ≡
126a. comp_core(π)(p) ‖
126b. ‖ {part(**uid_**P(p′))(p′) | p′:P•p′ ∈ **obs_**Ps(p)}

275

127. An *atomic process* consists of just an *atomic core process*, *atom_core*

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

276

128. 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.

277

value
128. core: π:Π → p:P → **in**,**out** {ch[{π,π′}|{π′∈ **mereo_**P(p)}]} **Unit**
128. core(π)(p) ≡
128a. **let** p′ = update(π)(p)
128b. **in** core(π)(p′) **end**
128b. **assert:** **uid_**P(p)=π=**uid_**P(p′)

278

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.

6 Continuous Entities 279

There are two kinds of continuous entities: materials (Sect. 6.1) and continuous behaviours (Sect. 6.2). By a material$_\delta$ we small mean a continuous endurant, a manifest entity which typically varies in shape and extent. By a continuous behaviour$_\delta$ we small mean a continuous perdurant, which we may think of as a function from continuous Time to some structure, simple or complicated, of parts and materials.

6.1 Materials 280

Let us start with examples of materials.

Example: 41 Materials. Examples of endurant continuous entities are such as coal, air, natural gas, grain, sand, iron ore[33], minerals, crude oil, solid waste, sewage, steam and water. ■ 281

The above materials are either liquid materials (crude oil, sewage, water), gaseous materials
282 (air, 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"*.

6.1.1 Materials-based Domains

By a materials based domain$_\delta$ we shall mean a domain *many of whose parts serve to transport materials, and some of whose actions, events and behaviours serve to monitor and control the part transport of materials.* 283

Example: 42 Material Processing. (i) Oil or gas materials are ubiquitous to pipeline systems — so pipeline systems are oil or gas-based systems. (ii) Sewage is ubiquitous to waste management systems — so waste management systems are sewage-based systems. (iii) Water is ubiquitous to systems composed from reservoirs, tunnels and aqueducts which again are ubiquitous to hydro-electric power plants, irrigation systems or water supply utilities — so hydro-electric power plants, irrigation systems and water supply utilities are water-based systems. ■ 284

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.

6.1.2 "Somehow Related" Parts and Materials

We explain our use of the term "somehow related". 285

Example: 43 Somehow Related Materials and Parts. With `teletype font` we designate materials and with *slanted font* we imply parts or part processes. (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*. (ii) `Grain` is delivered to silos by trucks, piped through a network of pipes, forks and valves to vessels, etc. (iii) `Minerals` are *mined*, *conveyed* by *belts* to *lorries* or *trains* or *cargo vessels* and finally *deposited*. (iv) `Iron ore`, for example, is *'conveyed'*[34] into *smelters*, *'roasted'*, *'reduced'* and *'fluxed'*, *'mixed'* with other mineral ores to produce a molten, pure metal, which is then *'collected'* into *ingots*. ■

[33] – whether molten or not

[34] The single quote terms are verbs to which there corresponds part processes.

6.1.3 Material Observers 286

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
287 endurants, i.e., parts.

Example: 44 Pipelines: Core Continuous Endurant. We continue Examples 30 on Page 53 and 31 on Page 54. The core continuous endurant, i.e., material, of (say oil) pipelines is, yes, oil:

type
  O **material**
value
  <u>**obs_**</u>O: PLN → 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
288 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"[35] of a material distinguishes it from other "parts". But they do have other attributes when
289 occurring in connection with, that is, related to parts, for example, volume or weight.

Example: 45 Pipelines: Parts and Materials. We continue Examples 30 on Page 53 and 31 on Page 54.

129. 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.

130. 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.
290

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

type
129. PLS, B, U, Vol
129. O **material**
value
129a. **obs**_Bs: PLS → B-**set**
129b. **obs**_U: B → U
129c. **obs**_O: B → O
130. **attr**_PLS_Type: PLS → {"oil"|"gas"}
130a. **attr**_Vol: U → Vol
130b. vol: O → Vol
axiom
130c. ∀ pls:PLS,b:B•b ∈ **obs**_Bs(pls)⇒vol(**obs**_O(b))≤**attr**_Vol(**obs**_U(b))

Notice how bodies are composite and consists of a discrete, atomic part, the unit, and a material endurant, the oil. We refer to Example 46. ■

6.1.4 Material Properties 291

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 paper we shall, when dealing with systems focused on materials, concentrate on modelling techniques for transport, flows, leaks and losses, and input to systems and output from systems. 292

Formal specification languages like `Alloy` [50], `Event B` [1] , `CASL` [29]`CafeOBJ` [37], `RAISE` [40], `VDM` [18, 19, 35] and `Z` [105] 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. 9.3.1 where we discuss these issues at some length. 293

Example: 46 Pipelines: Parts and Material Properties. We refer to Examples 30 on Page 53, 31 on Page 54 and 45 on the preceding page.

131. Properties of pipeline units additionally include such which are concerned with flows (F) and leaks (L) of materials[36]:
   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,

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

   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.

294

132. There are "the usual" arithmetic and comparison operators of flows and leaks, and there is a smallest detectable (flow and) leak.

type
132. F, L

value
132. ⊕, ⊖: (F|L)×(F|L) → (F|L)
132. <,≤,=: (F|L)×(F|L) → **Bool**
132. ⊗: (F|L)×**Real** → (F|L)
132. /: (F|L)×(F|L) → **Real**
132. ℓ_0:L

131a. **attr**_cur_iF: U → UI → F
131b. **attr**_max_iF: U → UI → F
131c. **attr**_cur_oF: U → UI → F
131d. **attr**_max_oF: U → UI → F
131e. **attr**_cur_iL: U → UI → L
131f. **attr**_max_iL: U → UI → L
131g. **attr**_cur_oL: U → UI → L
131h. **attr**_max_oL: U → UI → L
131i. **attr**_cur_L: U → L
131j. **attr**_max_L: U → L

295 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.

133. Properties of pipeline materials may additionally include
   a kind of material[37],
   b paraffins,
   c naphtenes,
   d aromatics,
   e asphatics,
   f viscosity,
   g etcetera.

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

6.1.5 Material Laws of Flows and Leaks 296

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.

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

Example: 47 Pipelines: Intra Unit Flow and Leak Law. We continue our line of Pipeline System examples (cf. the opening line of Example 46 on the preceding page).

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

135. The flows into a unit equal

[37]For example `Brent Blend` Crude Oil

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.

298

axiom

```
134.  ∀ pls:PLS,b:B\We\Si,u:U •
134.      b ∈ obs_Bs(pls)∧u=obs_U(b)⇒
134.      let (iuis,ouis) = mereo_U(u) in
135.      sum_cur_iF(iuis)(u) =
135a.         sum_cur_iL(iuis)(u)
135b.       ⊕ attr_cur_L(u)
135c.       ⊕ sum_cur_oF(ouis)(u)
135d.       ⊕ sum_cur_oL(ouis)(u)
134.      end
```

136. The sum_cur_iF (cf. Item 135) sums current input flows over all input connectors.

137. The sum_cur_iL (cf. Item 135a) sums current input leaks over all input connectors. 299

138. The sum_cur_oF (cf. Item 135c) sums current output flows over all output connectors.

139. The sum_cur_oL (cf. Item 135d) sums current output leaks over all output connectors.

```
136.  sum_cur_iF: UI-set → U → F
136.  sum_cur_iF(iuis)(u) ≡ ⊕ ⟨attr_cur_iF(ui)(u)|ui:UI•ui ∈ iuis⟩
137.  sum_cur_iL: UI-set → U → L
137.  sum_cur_iL(iuis)(u) ≡ ⊕ ⟨attr_cur_iL(ui)(u)|ui:UI•ui ∈ iuis⟩
138.  sum_cur_oF: UI-set → U → F
138.  sum_cur_oF(ouis)(u) ≡ ⊕ ⟨attr_cur_iF(ui)(u)|ui:UI•ui ∈ ouis⟩
139.  sum_cur_oL: UI-set → U → L
139.  sum_cur_oL(ouis)(u) ≡ ⊕ ⟨attr_cur_iL(ui)(u)|ui:UI•ui ∈ ouis⟩
      ⊕: (F×F)|F* → F | (L×L)|L* → L
```

where ⊕ is both an infix and a distributed-fix function which adds flows and or leaks. ■ 300

Example: 48 Pipelines: Inter Unit Flow and Leak Law.

140. 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.


```
140.   ∀ pls:PLS,b,b′:B,u,u′:U•
140.      {b,b′}⊆obs_Bs(pls)∧b≠b′∧u′=obs_U(b′)
140.       ∧ let (iuis,ouis)=mereo_U(u),(iuis′,ouis′)=mereo_U(u′),
140.             ui=uid_U(u),ui′=uid_U(u′) in
140.         ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
140a.            attr_cur_oF(us′)(ui′) ⊖ attr_leak_oF(us′)(ui′)
140b.          = attr_cur_iF(us)(ui) ⊕ attr_leak_iF(us)(ui)
140.         end
140.     comment: b′ precedes b
```

301

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. ■

6.2 Continuous Behaviours 302

This section is still under research and development.

The aim of this section is to relate discrete behaviour domain models of some fragments of a domain to continuous behaviour domain models of other fragments of that domain.

By a continuous behaviour model$_\delta$ we mean *a domain description that emphasises the behaviour of materials, that is, how they flow through parts, and related matters.*

6.2.1 Fluid Dynamics 303

Continuous behaviour domain models classically express the fluid dynamics$_\delta$ of flows of fluids, that is, the natural science of liquids and gasses. 304

The natural science of fluids (from `Wikipedia:`) *"are based on foundational axioms of fluid dynamics which are the conservation laws, specifically, conservation of mass, conservation of linear momentum (also known as Newton's Second Law of Motion), and conservation of energy (also known as First Law of Thermodynamics). These are based on classical mechanics. They are expressed using the Reynolds Transport Theorem."* 305

[1] Descriptions of Continuous Domain Behaviours: We are not going to exemplify such descriptive natural science models. Their mathematics, besides being elegant and beautiful, includes familiarity with Bernoulli Equations, Navier Stokes Equations, etc.

For continuous behaviour domain models we shall refer to such mathematical models of the natural science of fluids. 306

[2] Prescriptions of Required Continuous Domain Behaviours: By a prescriptive domain model$_\delta$ we mean *a desirable behaviour specification as in, for example, a requirements prescription* of a continuous time dynamic system.

We are also not going to illustrate prescriptive domain models. Their mathematics, besides also being elegant and beautiful, is based on the descriptive natural science models; but are now part of the engineering realm of *Control Theory*. It includes such disciplines as fuzzy control [69], stochastic control [56] and adaptive control [4], etc. 307

Example: 49 Pipelines: Fluid Dynamics and Automatic Control. We refer to Example 50 on the next page. In that example, next, we expect domain models for the fluid dynamics of individual pipeline units: wells, pumps, pipes, valves, forks, joins and sinks, as well as models (one or more) for sequences of such units, extending, preferably to entire nets: from wells to sinks. And we expect requirements description models again for each of some of the individual units: pumps and valves in particular: when they need and how they are controlled: regulating pumps and valves and which unit attributes need be monitored. ■

6.2.2 A Pipeline System Behaviour 308

We shall model the behaviours of a composite pipeline system. We shall be using basically the same form of the description as first illustrated in Sects. 2.8.2—2.8.7 (Pages 32–35) of Example 4. That system, Sects. 2.8.2—2.8.7, can be interpreted as illustrating the central monitoring of vehicles spread over a wide geographical area. The system to be illustrated in Example 50 can likewise be interpreted as illustrating the central monitoring of pipeline units (and their oil) spread over a wide geographical area. 309

Example: 50 A Pipeline System Behaviour. We consider (cf. Examples 30 on Page 53 and 31 on Page 54) the pipeline system units to represent also the following behaviours: pls:PLS, Item 129a on Page 70, to also represent the system process, pipeline_system, and for each kind of unit, cf. Example 30, there are the unit processes: unit, well (Item 98c on Page 53), pipe (Item 98a), pump (Item 98a), valve (Item 98a), fork (Item 98b), join (Item 98b) and sink (Item 98d on Page 53). 310

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) ≡
98c. is_We(u) → well(uid_U(u))(u),
98a. is_Pu(u) → pump(uid_U(u))(u),
98a. is_Pi(u) → pipe(uid_U(u))(u),
98a. is_Va(u) → valve(uid_U(u))(u),
98b. is_Fo(u) → fork(uid_U(u))(u),
98b. is_Jo(u) → join(uid_U(u))(u),
98d. is_Si(u) → sink(uid_U(u))(u)

311

We illustrate essentials of just one of these behaviours.

98b. 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**
98b. fork(ui)(u) ≡
98b. **let** u′ = core_fork_behaviour(ui)(u) **in**
98b. 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},
312 along channels u_u_ch[oui_1], u_u_ch[oui_2].

The core_··· _behaviour[s](ui)(u) also communicate with the pipeline_system behaviour. What we have in mind here is to model a traditional supervisory control and data acquisition, SCADA system.

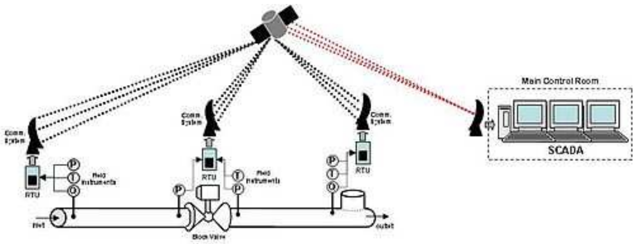


Figure 2: A supervisory control and data acquisition system

313

141. SCADA is then part of the scada_pipeline_system behaviour.

141. scada_pipeline_system: PLS →
141. **in**,**out** { pls_u_ch[ui]:ui:UI•i ∈ UIs(pls) } **Unit**
141. scada_pipeline_system(pls) ≡
141. scada(props(pls)) ‖ pipeline_system(pls)

props was defined in Sect. 4.2.5 Page 52.

We refer to Example 49 on Page 74: for all the core_··· _behaviours we expect the scada monitor to be expressed in terms of a prescriptive domain model which prescribes some optimal
314 form of control of the pipeline net.

142. 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
142. Props
value
142. scada: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis } **Unit**
142. scada(props) ≡
142a. scada(scada_own_work(props))
142b. ⌈⌉ scada(scada_data_acqui_work(props))
142c. ⌈⌉ scada(scada_control_work(props))

315 We leave it to the readers imagination to describe scada_own_work.

143. 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
143. scada_data_acqui_work: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis } Props
143. scada_data_acqui_work(props) ≡
143a. [] { **let** (ui,data) = pls_ui_ch[ui] ? **in**
143b. scada_input_update(ui,data)(props) **end**
143c. | ui:UI • ui ∈ uis }

143b. scada_input_update: UI × Data → Props → Props
type
143a. Data

316

144. 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.

144. scada_control_work: Props → **in**,**out** { pls_ui_ch[ui] | ui:UI•ui ∈ ∈ uis } Props
144. scada_control_work(props) ≡
144a. **let** (ui,ctrl) = analyse_scada(ui,props) **in**
144b. pls_ui_ch[ui] ! ctrl ;
144c. scada_output_update(ui,ctrl)(props) **end**

144c. scada_output_update UI × Ctrl → Props → Props
type
144a. 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. 322

7 A Domain Discovery Calculus 317

TO BE WRITTEN

7.1 An Overview 318

TO BE WRITTEN

7.1.1 Domain Analysers 319

MORE TO COME

- IS_ENTITY,
  IS_ENDURANT,
  IS_PERDURANT,
  IS_DISCRETE,
  IS_CONTINUOUS,
  IS_MATERIALS_BASED,
  IS_ATOMIC,
  IS_COMPOSITE and
  HAS_CONCRETE_TYPES.

7.1.2 Domain Discoverers 320

MORE TO COME

- PART_SORTS,
  MATERIAL_SORTS,
  PART_TYPES,
  UNIQUE_ID,
  MEREOLOGY,
  ATTRIBUTES,
  ACTION_SIGNATURES,
  EVENT_SIGNATURES and
  BEHAVIOUR_SIGNATURES.

7.1.3 Domain Indexes 321

We first made a reference to the concept of a "domain lattice" in Sect. 2.1.3 (Page 18).
In Fig. 3 on the facing page we show a similar "lattice" for the domain of road transport systems as illustrated in this paper.

MORE TO COME

7.2 Domain Analysers 323

TO BE WRITTEN

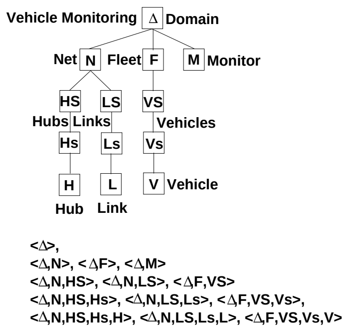


Figure 3: A domain lattice for the road transport system and the full set of domain indexes

7.2.1 Some Meta-meta Discoverers 324

- IS_ENTITY MORE TO COME
- IS_ENDURANT MORE TO COME
- IS_PERDURANT MORE TO COME
- IS_DISCRETE MORE TO COME
- IS_CONTINUOUS MORE TO COME

7.2.2 IS_MATERIALS_BASED 325

IS_MATERIALS_BASED

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

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

If Item 145 holds of a domain Δ_{Name} then the domain describer can apply MATERIAL_SORTS (Item 148 on Page 81).

Example: 51 Is Materials-based Domain. Example 45 on Page 70 Item 129 on Page 71.

7.2.3 IS_ATOMIC 326

IS_ATOMIC

The IS_ATOMIC analyser serves that purpose:

value

IS_ATOMIC: Index $\xrightarrow{\sim}$ **Bool**
IS_ATOMIC(ℓ^$\langle$t$\rangle$) ≡ **true** | **false** | **chaos**

Example: 52 Is Atomic Type. The IS_ATOMIC analyser has been applied in the following cases in Example 4: Sect. 2.1.1 on Page 17 Item 1c (M) on Page 17, Sect. 2.1.2 on Page 17 Item 4b (V) on Page 18, Sect. 2.1.3 on Page 18 Item 5b (H) on Page 18 and Sect. 2.1.3 on Page 18 Item 6b (L) on Page 18.

7.2.4 IS_COMPOSITE 327

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(ℓ^$\langle$t$\rangle$) ≡ **true** | **false** | **chaos**

328

Example: 53 Is Composite Type. The IS_COMPOSITE analyser has been applied in the following cases in Example 4: N: Sect. 2.1.2 on Page 17 Items 2a and 2b on Page 17, HS: Sect. 2.1.2 on Page 17 Item 2a on Page 17, Hs: Sect. 2.1.3 on Page 18 Item 5a on Page 18, LS: Sect. 2.1.2 on Page 17 Item 2b on Page 17, Ls: Sect. 2.1.3 on Page 18 Item 6a on Page 18, F: Sect. 2.1.2 on Page 17 Item 3 on Page 18, VS: Sect. 2.1.2 on Page 17 Item 4b on Page 18 and Vs: Sect. 2.1.2 on Page 17 Item 4a on Page 18. ■

7.2.5 HAS_A_CONCRETE_TYPE 329

HAS_A_CONCRETE_TYPE

146. Thus we introduce the analyser:

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

330

Example: 54 Has Concrete Types. The HAS_CONCRETE_TYPE analyser has been applied in the following cases in Example 4: VS, Vs: Sect. 2.1.2 on Page 17 Item 4a on Page 18, HS, Hs: Sect. 2.1.3 on Page 18 Item 5a on Page 18, LS, Ls: Sect. 2.1.3 on Page 18 Item 6a on Page 18 ■

7.3 Domain Discoverers 331

7.3.1 PART_SORTS 332

PART_SORTS

147. The part type discoverer PART_SORTS
   a applies to a simply indexed domain, $\ell\frown\langle t \rangle$,
   b where t denotes a composite type, and yields a pair
      i. of narrative text[38] and
      ii. formal text which itself consists of a pair:
         A. a set of type names
         B. each paired with a part (sort) observer.

value 333
147. PART_SORTS: Index $\xrightarrow{\sim}$ (**Text**×RSL)
147a. PART_SORTS($\ell\frown\langle t \rangle$):
147(b)i. [narrative, possibly enumerated texts ;
147(b)iiA. **type** $t_1, t_2, ..., t_m$,
147(b)iiB. **value** obs_t_1:$t \to t_1$,obs_t_2:$t \to t_2$,...,obs_t_m:$t \to t_m$
147b. **pre**: IS_COMPOSITE($\ell\frown\langle t \rangle$)]

334

Example: 55 Discover Part Sorts. We refer to Example 4. The PART_SORTS discoverer has been applied in the followig cases: Δ: Sect. 2.1.1 on Page 17 Items 1a–1c on Page 17, N, HS, LS: Sect. 2.1.2 on Page 17 Items 2a–2b on Page 17, HS: Sect. 2.1.2 on Page 17 Item 5 on Page 18, LS: Sect. 2.1.2 on Page 17 Item 6 on Page 18, Hs: Sect. 2.1.2 on Page 17 Item 5a on Page 18, Ls: Sect. 2.1.2 on Page 17 Item 6a on Page 18, F, VS: Sect. 2.1.2 on Page 17 Item 3 on Page 18, and VS, Vs: Sect. 2.1.2 on Page 17 Item 4a on Page 18. ■

7.3.2 MATERIAL_SORTS 335

MATERIAL_SORTS

148. The MATERIAL_SORTS discovery function applies to a domain, usually designated by $\langle \Delta_{\mathsf{Name}} \rangle$ where Name is a pragmatic hinting at the domain by name.
149. The result of the domain discoverer applying this meta-function is some narrative text
150. and the **type**s of the discovered materials
151. usually affixed a comment
   a which lists the "somehow related" part types

[38] In this paper we omit the narratives.

   b and their related materials observers.

148. MATERIAL_SORTS: ⟨Δ⟩ → (**Text** × RSL) 336
148. MATERIAL_SORTS($\langle \Delta_{\mathrm{Name}} \rangle$):
149. [narrative text ;
150. **type** M_a, M_b, ..., M_c **materials**
151. **comment**: related part **type**s: P_i, P_j, ..., P_k
151. obs_M_n : $P_m \to M_n$, ...]
145. **pre**: IS_MATERIALS_BASED($\langle \Delta_{\mathrm{Name}} \rangle$)

Example: 56 Material Sort. The MATERIAL_SORTS discoverer has been applied: O: Example 45 on Page 70, Items 129 and 129c on Page 71. ■

7.3.3 PART_TYPES 337

PART_TYPES

152. 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, t_c = te,
      ii. together with the sort definitions of so far undefined type names of te.
      iii. An observer function observes t_c from t.
      iv. The PART_TYPES discoverer is not defined if the designated sort is judged to not warrant a concrete type definition.

338
152. PART_TYPES: Index $\xrightarrow{\sim}$ (**Text**×RSL)
152. PART_TYPES($\ell\frown\langle t \rangle$):
152a. [narrative, possibly enumerated texts ;
152(b)i. **type** t_c = te,
152(b)ii. t_α, t_β, ..., t_γ,
152(b)iii. **value** obs_t_c: $t \to t_c$
152(b)iv. **pre**: HAS_CONCRETE_TYPE($\ell\frown\langle t \rangle$)]
152(b)ii. **where:** type expression te contains
152(b)ii. type names t_α, t_β, ..., t_γ

339

Example: 57 Part Types. The PART_TYPES discoverer has been applied in Example 4: VS, Vs: Sect. 2.1.2 on Page 17 Item 4a on Page 18, HS, Hs: Sect. 2.1.3 on Page 18 Item 5 on Page 18, and LS, Ls: Sect. 2.1.3 on Page 18 Item 6 on Page 18. ■

7.3.4 UNIQUE_ID 340

UNIQUE_ID

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

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

341

Example: 58 Unique ID. We refer to Example 4, Sect. 2.2.1 Page 19: LI, Item 7a, HI, Item 7b and VI, Item 7c. ■

7.3.5 MEREOLOGY 342

MEREOLOGY

154. Let type names t_1, t_2, ..., t_n denote the types of all parts of a domain.

155. Let type names ti_1, ti_2, ..., ti_n[39], be the corresponding type names of the unique identifiers of all parts of that domain.

156. 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.

157. Together with the "discovery" of the mereology function there usually follows some axioms.

343

type
154. t_1, t_2, ..., t_n
155. t_{idx} = ti_1 | ti_2 | ... | ti_n

156. MEREOLOGY: Index $\stackrel{\sim}{\rightarrow}$ Index-**set** $\stackrel{\sim}{\rightarrow}$ (**Text**×RSL)
156. MEREOLOGY(ℓ⁀⟨t⟩)({ℓ_i⁀⟨t_j⟩,...,ℓ_k⁀⟨t_l⟩}):
156. [narrative, possibly enumerated texts ;
156. **either:** {}
156. **or:** **value** mereo_t: t → ti_x
156. **or:** **value** mereo_t: t → ti_x-**set** × ti_y-**set** × ... × ti_x-**set**
157. **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.

344

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

Example: 59 Mereologies. The MEREOLOGY discoverer was applied in Example 4, Sect. 2.2.2 on Page 19, Items 8a–9b on Page 20, Example 20 on Page 46, Items 74–77 on Page 46, Example 22 on Page 46, Items 79–80e on Page 47 and Example 30 on Page 53, Items 96–98d on Page 54.

7.3.6 ATTRIBUTES 345

ATTRIBUTES

158. Attributes have types. We assume attribute type names to be distict from part type names.

159. ATTRIBUTES applies to parts of type t and yields a pair of
   a 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 sort at of t.

346

type
158. at = at_1 | at_2 | ... | at_n
value
159. ATTRIBUTES: Index → (**Text**×RSL)
159. ATTRIBUTES(ℓ⁀⟨t⟩):
159a. [narrative, possibly enumerated texts ;
159(b)i. **type** at_1, at_2, ..., at_m
159(b)ii. **value** attr_at_1:t→at_1,attr_at_2:t→at_2,...,attr_at_m:t→at_m]

where m≤n

347

Example: 60 Attributes. The ATTRIBUTES discoverer was applied in Example 4, Sect. 2.2.3 for attributes of Links, Items 10–10c Pages 20–21, Hubs, Items 11–11c Pages 21–21, and Vehicles, Items 12–12 Pages 22–22; as well as in many other examples.

7.3.7 ACTION_SIGNATURES 348

ACTION_SIGNATURES

160. 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.

349

160 ACTION_SIGNATURES: Index → Index-**set** $\stackrel{\sim}{\to}$ (**Text**×RSL)
160 ACTION_SIGNATURES(ℓ⌢⟨t⟩)({ℓ_1⌢⟨t_1⟩,ℓ_2⌢⟨t_2⟩,...,ℓ_n⌢⟨t_n⟩}):
160 [narrative, possibly enumerated texts ;
160 **type** t_a,t_b,... t_c,
160b **value**
160b act_i:$te_{i_d} \stackrel{\sim}{\to} te_{i_r}$,$act_j$:$te_{j_d} \stackrel{\sim}{\to} te_{j_r}$,...,$act_k$:$te_{k_d} \stackrel{\sim}{\to} te_{k_r}$
160c **where:**
160c type names in $te_{(i|j|...|k)_d}$ and in $te_{(i|j|...|k)_r}$ are either
160c type names t_a, t_b, ... t_c or are type names defined by the
160c indices which are prefixes of ℓ_m⌢⟨T_m⟩ and where T_m is
160c in some signature $\mathsf{act}_{i|j|...|k}$]

350

Example: 61 Action Signatures. The ACTION_SIGNATURES discoverer was applied in Example 4: ins_H, Item 37 on Page 29, Sect. 5.2.3 on Page 58, see Example 33 on Page 58, etcetera.

7.3.8 EVENT_SIGNATURES 351

EVENT_SIGNATURES

161. 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.

352

161 EVENT_SIGNATURES: Index → Index-**set** $\stackrel{\sim}{\to}$ (**Text**×RSL)
161 EVENT_SIGNATURES(ℓ⌢⟨t⟩)({ℓ_1⌢⟨t_1⟩,ℓ_2⌢⟨t_2⟩,...,ℓ_n⌢⟨t_n⟩}):
161a [narrative, possibly enumerated texts omitted ;
161a **type** t_a,t_b,... t_c,
161b **value**
161b evt_pred$_i$: $te_{d_i} \times te_{r_i}$ → **Bool**
161b evt_pred$_j$: $te_{d_j} \times te_{r_j}$ → **Bool**
161b ...
161b evt_pred$_k$: $te_{d_k} \times te_{r_k}$ → **Bool**]

161c **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.
■

353

Example: 62 Event Signatures. Example 4, Sect. 2.7 on Page 29 Item 38 on Page 30.

7.3.9 DISCRETE_BEHAVIOUR_SIGNATURES 354

BEHAVIOUR_SIGNATURES

162. The BEHAVIOUR_SIGNATURES meta-function, besides narrative texts, yields
163. 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.

355

162. BEHAVIOUR_SIGNATURES: Index→ Index-**set** $\stackrel{\sim}{\to}$ (**Text**×RSL)
162. BEHAVIOUR_SIGNATURES(ℓ⌢⟨t⟩)({ℓ_1⌢⟨t_1⟩,ℓ_2⌢⟨t_2⟩,...,ℓ_n⌢⟨t_n⟩}):
163a. [narrative, possibly enumerated texts ;
163(b)i. **type** m = m_1 | m_2 | ... | m_μ, $\mu \geq 1$
163(b)ii. i = i_1 | i_2 | ... | i_n, $n \geq 0$
163(b)iii. **channel** c:m, {vc[x]|x:i_a}:m, {mc[x,y]|x:i_b,y:i_c}:m,...
163(b)iv. **value**
163(b)iv. bhv_1: ate_1 → $inout_1$ rte_1,
163(b)iv. ... ,
163(b)iv. bhv_m: ate_m → $inout_m$ rte_m.]
163(b)iv. **where** type expressions ate_i and rte_i for all i involve at least
163(b)iv. two types t'_i, t''_j of respective indexes ℓ_i⌢⟨t_i⟩, ℓ_j⌢⟨t_j⟩,
163(b)v. **where** **Unit** may appear in either ate_i or rte_j or both.
163(b)v. **where** $inout_i$: **in** k | **out** k | **in**,**out** k
163(b)v. **where** k: c **or** vc[x] **or** {vc[x]|x:i_a•x ∈ xs} **or**
163(b)v. {mc[x,y]|x:i_b,y:i_c • x ∈ xs ∧ y ∈ ys} **or** ...

356

Example: 63 Behaviour Signatures. The BEHAVIOUR_SIGNATURES discoverer was applied in several examples: Example 4, Sect. 2.8.5 on Page 33 Items 61–63 on Page 34; Sects. 5.4.3 on Page 63 to 5.4.4 on Page 63 inclusive, ; Example 50 on Page 75; etcetera.

7.4 Some Technicalities 357

7.4.1 Order of Analysis and "Discovery"

Analysis and "discovery", that is, the "application" of the analysis meta-functions of Sect. 7.2 and the "discovery" meta-functions of Sect. 7.3 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.

7.4.2 Analysis and "Discovery" of "Leftovers" 358

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 paper we do not suggest 359
any meta-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.

7.5 Laws of Domain Descriptions 360

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. 361

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 $\Re$).

7.5.1 1st Law of Commutativity 362

We make a number of assumptions: the following two are well-formed indices of a domain: ι': $\langle\Delta\rangle^\frown\ell'^\frown\langle\mathsf{A}\rangle$, ι'': $\langle\Delta\rangle^\frown\ell''^\frown\langle\mathsf{B}\rangle$, 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. 363

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:

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

When a domain description development satisfies Law 164., under the above assumptions, then we say that the development — modulo type, action, event and behaviour name "assignments" — satisfies a mild form of commutativity.

7.5.2 2nd Law of Commutativity 364

Let us assume that we are exploring the sub-domain at index ι: $\langle\Delta\rangle^\frown\ell^\frown\langle\mathsf{A}\rangle$. Whether we first "discover" $\mathcal{A}$ttributes and then $\mathcal{M}$ereology (including $\mathcal{U}$nique identifiers) or first "discover"
365 $\mathcal{M}$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 $\langle\Delta\rangle^\frown\ell^\frown\langle\mathsf{A}\rangle$, and ιs for a suitable set of indices. Thus we wish the following law to hold:

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

here modulo attribute and unique identifier type name renaming.

7.5.3 3rd Law of Commutativity 366

Let us again assume that we are exploring the sub-domain at index ι: $\langle\Delta\rangle^\frown\ell^\frown\langle\mathsf{A}\rangle$ where ιs is a suitable set of indices. Whether we are exploring actions, events or behaviours at that domain
367 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:

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

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

7.5.4 1st Law of Stability 368

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:

167. $(\mathcal{D}(\iota)(\iota s);\mathcal{A}\text{\_and\_}\mathcal{D}\text{\_seq})(\Re)\ \simeq\ (\mathcal{D}(\iota)(\iota s);\mathcal{A}\text{\_and\_}\mathcal{D}\text{\_seq};\mathcal{D}(\iota)(\iota s))(\Re)$

where $\mathcal{D}$ is any discovery function, $\mathcal{A}$_and_$\mathcal{D}$_seq is any specific sequence of intermediate analyses and discoveries, and where ι and ιs are suitable indices, respectively sets of indices.

7.5.5 2nd Law of Stability 369

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:

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

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

7.5.6 Law of Non-interference 370

When performing a discovery meta-operation, $\mathcal{D}$ on any index, ι, and possibly index set, ιs, 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:

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

371

We say that when 169. is satisfied for any discovery meta-function $\mathcal{D}$, for any indices ι and ιs 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.

7.6 Discussion 372

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. 373

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. 374

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. 375 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. 376

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. 377 We have therefore, in this paper, 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.

8 Requirements Engineering 378

We shall give a terse overview of some facets of requirements engineering. Namely those which "relate" domain engineering to requirements engineering. The relation is the following: one can "derive", not automatically, but systematically, domain requirements and significant aspects of interface requirements from domain descriptions.

8.1 A Requirements "Derivation" 379

8.1.1 Definition of Requirements

IEEE Definition of 'Requirements'

By a requirements we understand (cf. IEEE Standard 610.12 [48]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*.

8.1.2 The Machine = Hardware + Software 380

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

8.1.3 Requirements Prescription 381

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.

8.1.4 Some Requirements Principles 382

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, formulate tests (theorems, properties for model checking) whose actualisation show adherence to the requirements.

We shall not show adherence to the above rules.

8.1.5 A Decomposition of Requirements Prescription 383

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.

8.1.6 An Aside on Our Example 384

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"* [99]. 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 385
be paid for by its 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 95) of the next section, Sect. 8.2. So all the other parts of the next section serve to build up to the Extension section, Sect. 8.2.4 on Page 95.

8.2 Domain Requirements 386

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 re- 387
quirements 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. 388

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 389
example is that of a tollway system — in contrast to the general nets. See Fig. 4 on the following page.

The links of the general net of Fig. 4 on the next page 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. 390

8.2.1 Projection 391

By domain projection$_\delta$ we mean that a subset of the domain description is kept. In the tollway example we actually keep all the parts, their properties and therefore the types and functions

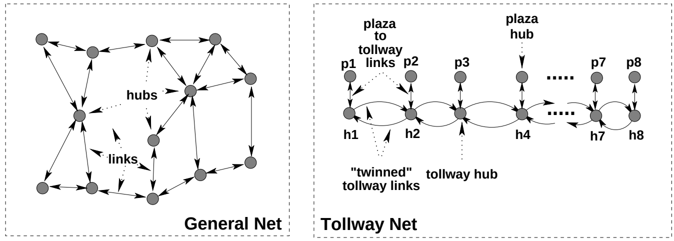


Figure 4: General and Tollway Nets

derived from these, Thus we keep: 1a–1c (N, F, M) 2–2b (HS, LS), 5a–6b (Hs, Ls, H, L), 7a–7b (HI, LI), 10a–10c (LΣ, LΩ, LEN, LOC) and 11a–11c (HΣ, HΩ, LOC) , 3–4b, 7c (VS, Vs, V), 8a–9b (**mereo_L**), 12a–12(a)iii, 13 (VP, atH, onL, FRAC, **attr_**VP), We do not keep any actions or events (!), But we keep the behaviours: 59–59b (trs), 61–63 (trs, veh, mon), 65–65d, 64–68 (veh), 69–71d (mon).

8.2.2 Instantiation 392

393

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

170. The net is now concretely modelled as a pair of sequences.
171. One sequence models the plaza hubs, their plaza-to-tollway link and the connected tollway hub.
172. The other sequence models the pairs of "twinned" tollway links.
173. From plaza hubs one can observe their hubs and the identifiers of these hubs.
174. The former sequence is of m such plaza "complexes" where $m \geq 2$; the latter sequence is of $m-1$ "twinned" links.
175. From a tollway net one can abstract a proper net.

394

```
type
170. TWN = PC* × TL*
171. PC = PH × L × H
172. TL = L × L
value
171. obs_H: PH → H, obs_HI: PH → HI
axiom
174. ∀ (pcl,tll):TWN •
174.    2≤len pcl∧len pcl=len tll+1

value
175. abs_HsLs: TWN → (Hs×Ls)
175. abs_HsLs(pcl,tll) as (hs,ls)
175.   pre: wf_TWN(pcl,tll)
175.   post:
175.     hs = {h,h′|(h,_,h′):PC • (h,_,h′)∈ elems pcl}
175.   ∧ ls = {l|(_,l,_):PC • (_,l,_)∈ elems pcl} ∪
175.           {l,l′|(l,l′):TL•(l,l′)∈ elems tll}
```

395

[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:

176. 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

177. 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 $(p_{i+1}, l_{i+1}, h_{i_1}))$.

396

```
value
   Instantiation_wf_TWN: TWN → Bool
   Instantiation_wf_TWN(pcl,tll) ≡
176.   ∀ i:Nat • i ∈ inds pcl⇒
176.      let (pi,li,hi)=pcl(i) in
176a.       obs_LIs(li)={obs_HI(pi),obs_HI(hi)}
176b.     ∧ obs_LI(li)∈ obs_LIs(pi)∩ obs_LIs(hi)
177.    ∧ let (li′,li′′) = tll(i) in
177.        i < len pcl ⇒
177.         let (pi′,li′′′,hi′) = pcl(i+1) in
177a.          obs_HIs(li) = obs_HIs(li′)
177a.          = {obs_HI(hi),obs_HI(hi′)}
       end end end
```

8.2.3 Determination 397

By domain determination$_\delta$ we mean, as illustrated in this example, making part property values less in-determinate, i.e., more determinate.

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[40]: 398

[1] Model Well-formedness wrt. Determination:: We need define well-formedness wrt. determination. Please study Fig. 5 on the following page. 399

178. All hub and link state spaces contain just one hub, respectively link state.
179. 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

180. 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

[40] 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 179a and 179b.

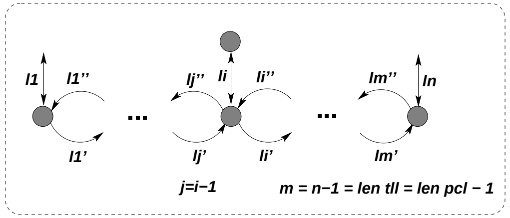


Figure 5: Hubs and Links

181. the jth tollway hub, h_j (for $1 \leq j \leq$ **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 $j = i+1 =$ **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.

400

```
value
179. Determination_wf_TWN: TWN → Bool
179. Determination_wf_TWN(pcl,tll) ≡
179.   ∀ i:Nat• i ∈ inds tll ⇒
179.     let (pi,li,hi) = pcl(i),
179.        (npi,nli,nhi) = pcl(i+1), in
179.        (li′,li′′) = tll(i) in
178.     obs_HΩ(pi)={obs_HΣ(pi)}∧obs_HΩ(hi)={obs_HΣ(hi)}
178.   ∧ obs_LΩ(li)={obs_LΣ(li)}∧obs_LΩ(li′)={obs_LΣ(li′)}
178.   ∧ obs_LΩ(li′′)={obs_LΣ(li′′)}
179a.   ∧ obs_LΣ(li)
179a.        = {(obs_HI(pi),obs_HI(hi)),(obs_HI(hi),obs_HI(pi))}
179a.   ∧ obs_LΣ(nli)
179a.        = {(obs_HI(npi),obs_HI(nhi)),(obs_HI(nhi),obs_HI(npi))}
179b.   ∧ {(obs_LI(li),obs_LI(li))}⊆obs_HΣ(pi)
179b.   ∧ {(obs_LI(nli),obs_LI(nli))}⊆obs_HΣ(npi)
180a.   ∧ obs_LΣ(li′)={(obs_HI(hi),obs_HI(nhi))}
180b.   ∧ obs_LΣ(li′′)={(obs_HI(nhi),obs_HI(hi))}
181.   ∧ case i+1 of
181a.       2 → obs_HΣ(h_1)=
181a.              {(obs_LΣ(l_1),obs_LΣ(l_1)), (obs_LΣ(l_1),obs_LΣ(l_1′′)),
181a.              (obs_LΣ(l′′_1),obs_LΣ(l_1)), (obs_LΣ(l′′_1),obs_LΣ(l′_1))},
181b.     len pcl → obs_HΣ(h_i+1)=
181b.             {(obs_LΣ(l_len pcl),obs_LΣ(l_len pcl)),
181b.              (obs_LΣ(l_len pcl),obs_LΣ(l′_len tll)),
181b.              (obs_LΣ(l′′_len tll),obs_LΣ(l_len pcl)),
181b.              (obs_LΣ(l′′_len tll),obs_LΣ(l′_len tll))},
```

181c. _ → obs_HΣ(h_i)=
181c. {(obs_LΣ(l_i),obs_LΣ(l_i)), (obs_LΣ(l_i),obs_LΣ(l′_i)),
181c. (obs_LΣ(l_i),obs_LΣ(l″_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i)),
181c. (obs_LΣ(l″_i),obs_LΣ(l′_i−1)), (obs_LΣ(l″_i),obs_LΣ(l′_i))}
179. **end end**

8.2.4 Extension 401

By domain extension$_\delta$ 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.* 402

Backgorund: The road traffic monitoring domain of Example 4, notably Sects. 2.8.6–2.8.7, (Items 65–71d Pages 34–35), illustrated the intangible abstraction of road traffic in the form of the recording of a discrete version of that traffic:[41]

46. d𝕋
45. dRTF = d𝕋 $\overrightarrow{m}$ (VI $\overrightarrow{m}$ VP)

by the road traffic system:

value
59. trs() =
59a. ∥ {veh(**uid_**V(v))(v)(vpm(**uid_**V(v)))|v:V•v ∈ vs}
59b. ∥ mon(mi)(m)([t_0 ↦ vpm])

403

We say that the road traffic, dRTF is intangible since the dRTF function, being a function, is an intangible. The domain extension is now making that "function" a tangible notion. There is no presumption, in defining the monitor behaviour, that there is indeed a mechanised behaviour, i.e., a computerised process that "implements" that monitor. Since one can speak of the monitor behaviour, one can, as well define it. 404

The Extension: We now "implement" a version of the above monitor behaviour. The proposed domain extension builds upon the monitor and the ability of vehicles to communicate their vehicle positions to the monitor, cf. Items 65a and 65a Page 34, Items 66a, 66(c)i and 66(c)iiA Page 34 and Item 71a Page 35. Instead of this "directness" we interpret links and 405 hubs of the tollway system as behaviours endowed with sensors. Vehicle behaviours now interact with link and hub behaviours communicating their positions which the link and hub behaviours communicate to a tollway system monitor. The domain extension then consists of the extension of links and hubs with sensors and the modelling of their vehicle interactions and their interaction with the tollway system monitor. 406

The Formalisation: We introduce

182. rather simple link and hub behaviours, and

[41] In dRTF we change V into a reference to vehicles VI.

183. an array of channels for the interaction of vehicle behaviours with link and hub behaviours.

And we modify

184. the vehicle and monitor behaviours and

185. the vehicle/monitor channel

407 the latter to now serve at the channel for link and hub interactions with the refined monitor behaviour.

value
175. (hs,ls):(Hs,Ls) = abs_HsLs(twn)
22. his:HI-**set** = {**uid_**H(h)|h:H•h ∈ hs}
21. lis:LI-**set** = {**uid_**L(l)|l:L•l ∈ ls}
channel
183. {vlh_ch[vi,si]|vi:VI,si:(LI|HI)•vi ∈ vis∧si ∈ lis ∪ his}:VP
185. {lhm_ch[si,mi]|si:(LI|HI)•si ∈ lis ∪ his}:(VI×VP)
value
183. link: li:LI → L → **in** { vlh_ch[vi,si]|si:LI•si ∈ lis } **Unit**
183. hub: hi:HI → H → **in** { vlh_ch[vi,si]|si:HI•si ∈ his } **Unit**
182. link(li)(l) ≡
182. (...⌈⌉ ⌈⌉ {**let** (vi,vp) = vlh_ch[vi,li]? **in** lhm_ch[li,mi]!(vi,vp)|vi:VI•vi ∈ vis **end**});link(li)(l)
182. hub(hi)(h) ≡
182. (...⌈⌉ ⌈⌉ {**let** (vi,vp) = vlh_ch[vi,hi]? **in** lhm_ch[hi,mi]!(vi,vp)|vi:VI•vi ∈ vis **end**});hub(hi)(h)
59. trs() =
59a. ∥ {veh(**uid_**V(v))(v)(vpm(**uid_**V(v)))|v:V•v ∈ vs}
59b. ∥ mon(mi)(m)([t_0 ↦ vpm])
182. **∥ {link(uid_L(l))(l)|l:L•l ∈ ls}**
182. **∥ {hub(uid_H(h))(h)|h:H•h ∈ hs}**

408

The modifications to the vehicle behaviour is shown in Items 65a′, 65(b)ii′, 66a′, 66(c)i′, 66(c)iiA′ and 71a′ (Pages 96–97).

65. veh(vi)(v)(vp:atH(fli,hi,tli)) ≡
65a′. **vlh_ch[vi,hi]**!(vi,vp) ; veh(vi)(v)(vp)
65b. ⌈⌉
65(b)i. **let** {hi′,thi}=**mereo_**L(get_L(tli)(n)) **in assert:** hi′=hi
65(b)ii′. **vlh_ch[vi,tli]**!(vi,onL(hi,tli,0,thi)) ;
65(b)iii. veh(vi)(v)(onL(hi,tli,0,thi)) **end**
65c. ⌈⌉
65d. **stop**

409

64. veh(vi)(v)(vp:onL(fhi,li,f,thi)) ≡
66a′. **vlh_ch[vi,li]**!(vi,vp) ; veh(vi)(v)(vp)
66b. ⌈⌉
66c. **if** f + δ<1

```
66(c)i′.     then vlh_ch[vi,li]!(vi,onL(fhi,li,f+δ,thi)) ;
66(c)i.          veh(vi)(v)(onL(fhi,li,f+δ,thi))
66(c)ii.     else let li′:LI•li′ ∈ mereo_H(get_H(thi)(n)) in
66(c)iiA′.       vlh_ch[vi,thi]!(vi,atH(li,thi,li′)) ;
66(c)iiB.        veh(vi)(v)(atH(li,thi,li′)) end end
67.       ⌈⌉
68.        stop

69.  mon(mi)(m)(rtf) ≡
70.       mon(mi)(own_mon_work(m))(rtf)
71.     ⌈⌉
71a′.    [] { let ((vi,vp),t) = (lhm_ch[si,mi]?,clk_ch?) in
71b.         let rtf′ = rtf † [ t ↦ rtf(max dom rtf) † [ vi ↦ vp ] ] in
71c.         mon(mi)(m)(rtf′) end
71d.         end | si:(LI|HI) • si ∈ lis ∪ his}
```

The extension, in this example, does not really amount to much. We say that we have 410
extended links and hubs with sensors. But we have not really modelled these sensors. We have modelled their intent, but not their extent. A more complete extension, which has to be done, but which is not shown in this paper, would now model these sensors as they rely on the unique vehicle identifier to be sensed. We shall, regrettably, omit this aspect of our 411
presentation of the extension. There are so very many ways in which sensors and their object: the vehicles, can interact. Vehicles can be equipped with radio frequency identification tags, etcetera. Whichever sensor technology is chosen, it must be described. A description includes both it proper and its erroneous functioning. Such (IT equipment &c.) descriptions may be expressed in a number of steps: First, as here, a `RSL/CSP` [47, 8]. model. Then a "derived" description models temporal properties — using `Duration Calculus, DC` [106], or `Temporal Logic of Actions, TLA+` [59]. Finally a timed-automata [2, 73] model which "implements" the `DC` model.

8.3 Interface Requirements Prescription 412

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$_\delta$ if it is present in some related forms, in both the domain and the machine. 413

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.

8.3.1 Shared Parts 414

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,
415 and many others.

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 $d_{r.net}$
416 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
417 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 effort it must be secured that "standard" actions
418 on nets, hubs and links can be supported by the chosen relational database system DB_{rel}.

[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.

419 Essentially these prescriptions concretise the insert link action.

[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.

8.3.2 Shared Actions 420

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.)

8.3.3 Shared Events 421

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.

8.3.4 Shared Behaviours 422

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.

8.4 Machine Requirements 423

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

8.5 Discussion of Requirements "Derivation" 424

We have indicated how the domain engineer and the requirements engineer can work together to
"derive" significant fragments of a requirements prescription. This puts requirements engineering 425
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 paper that job would not be so straightforward as we now suggest.

9 Conclusion 426

This paper, 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 4–7 possibly represent
427 two publishable journal papers. Section 8 has been "added" to the 'tutorial' notes. The style of the two tutorial "parts", Sects. 4–7 and Sect. 8 are, necessarily, different: Sects. 4–7 are in the form of research notes, whereas Sect. 8 is in the form of "lecture notes" on methodology. Be that as it may. Just so that you are properly notified !

9.1 Comparison to Other Work 428

In this section we shall only compare our contribution to domain science & engineering as presented above to that found in the broader literature with respect to the computer science and software engineering term 'domain'. Finally we shall also not compare our work on a
429 description calculus as we find no comparable literature ! Our comparison hinges on basically the following two facets: domain analysis and domain description. We shall see that the former term, seen across the surveyed literature, covers techniques that are claimed used in many steps of software engineering, but that they seldom, if ever, involve formal concept analysis as we understand it (cf. Sects. ?? on Page ??, 4.1.4 on Page 40 and 5.1 on Page 57).

9.1.1 Ontological Engineering: 430

Ontological engineering is described mostly on the Internet, see however [7]. 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
431 work and processed by various tools. There does not seem to be a concern for "deriving" such ontologies into requirements for software. Usually ontology presentations either start with the presentation or makes reference to its reliance of an upper ontology. Instead the ontology databases appear to be used for the computerised discovery and analysis of relations
432 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.

9.1.2 Knowledge and Knowledge Engineering: 433

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 understand-*

ing 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. 434

The aim of knowledge engineering was formulated, in 1983, by an originator of the concept, Edward A. Feigenbaum [34]: knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise. Knowledge engineering focuses on continually build- 435 ing 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. 436

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 [20]. 437

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 [93, 57].

9.1.3 Prieto-Dĩaz: Domain Analysis: 438

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. 439

In this section we shall mainly be comparing the TripTych approach to domain analysis to that of Reubén Prieto-Dĩaz's approach [78, 79, 80]. Firstly, the two meanings of domain analysis basically coincide. Secondly, in, for example, [78], Prieto-Dĩaz's domain analysis is focused on the very important stages that precede the kind of domain modelling that we have described. Major concerns of Prieto-Dĩaz's approach are **selection** of what appears to 440 be similar, but specific entities, **identification** of common features, **abstraction** of entities and **classification**. In comparison **selection** and **identification** is assumed in our approach, but using Ganter & Wille's *Formal Concept Analysis* [38] where Prieto-Dĩaz really does not report on a systematic, let alone formal approach to identification. **Abstraction** (from values to types and signatures) and **classification** into parts, materials, actions, events and behaviours is what we have focused on; as we have also focused on their formalisation. All-in-all we 441 find Prieto-Dĩaz's work relevant to our work: relating to it by providing guidance to pre-modelling steps, thereby emphasising issues that are necessarily informal, yet difficult to get started on by most software engineers. Where we might differ is on the following: although Prieto-Dĩaz does mention a need for domain specific languages, he does not show examples

of domain descriptions in such DSLs. We, of course, basically use mathematics as the DSL. In the TripTych approach to domain analysis we provide a full ontology and suggest a domain description calculus. In our approach we do not consider requirements, let alone software components, as does Prieto-Dĩaz, but we find that that is not an important issue.

9.1.4 Software Product Line Engineering: 442

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 443 build software based on reusable software components. These are not the primary concerns of TripTych domain science & engineering. But they do become concerns as we move from domain descriptions to requirements prescriptions. But it strongly seems that software product line engineering is not really focused on the concerns of domain description — such as is TripTych domain engineering. It seems that software product line engineering is primarily based, as is, for example, FODA: Feature-oriented Domain Analysis, on analysing features of software systems. Our [15] 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 [6, 103, 3, 94, 43, 87, 23, 28, 32, 75].

9.1.5 M.A. Jackson: Problem Frames: 444

The concept of problem frames is covered in [53]. 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, 445 the same as for us, the problem world analysis. In the problem frame approach the software developer plays three, that is, all the TripTych rôles: domain engineer, requirements engineer and software engineer "all at the same time", 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 and the design of software. These, really are minor points. 446 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 [51] 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 paper. They have to do with software verification and validation. These aspects are covered in [41, 51]. .

9.1.6 Domain Specific Software Architectures (DSSA): 447

It seems that the concept of DSSA was formulated by a group of ARPA[42] project "seekers" who also performed a year long study (from around early-mid 1990s); key members of the DSSA project were Will Tracz, Bob Balzer, Rick Hayes-Roth and Richard Platek [95]. The [95] definition of domain engineering is *"the process of creating a DSSA: domain analysis and domain modelling followed by creating a software architecture and populating it with software*

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

components.” This definition is basically followed also by [68, 88, 66]. Defined and pursued 448
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 architecture on these findings. Thus DSSA turns matter 449
“upside-down” with respect to TripTych requirements development by starting with software
components, assuming that these satisfy some requirements, and then suggesting domain spe-
cific 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 de-
scriptionss without, in principle, considering software components, whether already existing,
or being subsequently developed. Of course, given a domain descriptions it is obvious that one 450
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. It seems to this author that had the DSSA promoters based their studies and practice 451
on also using formal specifications, at all levels of their study and practice, then some very
interesting insights might have arisen.

9.1.7 Domain Driven Design (DDD) 452

Domain-driven design (DDD)[43] *“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 con-
ceptual heart of the problem.”*[44] We have studied some of the DDD literature, mostly only 453
accessible on The Internet, but see also [44], and find that it really does not contribute to
new insight into domains such as wee see them: it is just “plain, good old software engineering
cooked up with a new jargon.

9.1.8 Feature-oriented Domain Analysis (FODA): 454

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 [55] states: *“FODA
is a necessary first step”* for software reuse. To the extent that TripTych domain engineering 455
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 [30].
Since FODA “leans” quite heavily on ‘Software Product Line Engineering’ our remarks in that
section, above, apply equally well here.

9.1.9 Unified Modelling Language (UML) 456

Three books representative of UML are [22, 83, 54]. The term domain analysis appears numerous times in these books, yet there is no clear, definitive understanding of whether it, the

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

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
457 either software design (as it most often is), or requirements prescription. Certainly, in UML,
in [22, 83, 54] 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”
458 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.

9.1.10 Requirements Engineering: 459

There are in-numerous books and published papers on requirements engineering. A seminal
one is [100]. I, myself, find [60] full of very useful, non-trivial insight. [33] is seminal in that it
460 brings a number or early contributions and views on requirements engineering. Conventional
text books, notably [74, 77, 91] all have their “mandatory”, yet conventional coverage of
requirements engineering. None of them “derive” requirements from domain descriptions, yes,
OK, from domains, but since their description is not mandated it is unclear what “the domain”
is. Most of them repeatedly refer to domain analysis but since a written record of that domain
461 analysis is not mandated it is unclear what “domain analysis” really amounts to. Axel van
Laamsweerde’s book [100] 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 [60] can beneficially be followed also in TripTych requirements
development. Neither [100] nor [60] preempts TripTych requirements development.

9.1.11 Summary of Comparisons 462

It should now be clear from the above that there are basically two notions from above that relate to our notion of domain analysis. (i) Prieto-Dĩaz’s notion of *‘Domain Analysis’*, and (ii) Jackson’s notion of *Problem Frames*. But it should also be clear that none of the surveyed literature, except, of course, Ganter & Wille’s [38] *Formal Concept Analysis, Mathematical Foundations*, covers our notion of domain analysis as it hinges crucially on Ganter & Wille’s formal concept analysis.

9.2 What Have We Omitted: Domain Facets 463

One can further structure domain descriptions along the lines of the following domain facets:

- intrinsics,
- support technologies,
- rules & regulations,
- incl. scripts,
- organisation & management and
- human behaviour

of domains. We refer to [13] for an early treatment of domain facets.

9.2.1 Intrinsics 464

By intrinsics$_\delta$ we shall mean *the entities in terms of which all other domain facets are expressed.*

Example: 64 Road Transport System Intrinsics. We refer to Example 4. The following parts are typical of intrinsic parts: N, HS, Hs, LS, Ls, H, L; F, VS, Vs, V. ■

9.2.2 Support Technologies 465

By a support technology$_\delta$ we shall mean *a human (soft technological) or a hard technological means of supporting, that is, presenting entities and carrying out functions: actions and behaviours.*

Example: 65 Tollroad System Support Technologies. We refer to Example 8.2.4 (Pages 95–97). The link sensors, the hub sensors, and the monitor are examples of support technologies. ■

9.2.3 Rules & Regulations 466

[1] Rules: By a rule$_\delta$ we shall mean *some, usually syntactically expressed predicate which expresses whether an action (say of a behaviour) violates some state property.*

Example: 66 Road Transport System Rules. We refer to Sect. 8.2.4 (Pages 95–97). If a vehicle somehow disables its ability to be sensed then a rule has been violated. ■ 467

[2] Regulation: By a regulation$_\delta$ we shall mean *some, usually syntactically expressed state-to-state transformer which expresses how an erroneous state resulting from a rule violation can be restored to a state in which rule adherence is "restored".* ■

Example: 67 Road Transport System Regulations. We refer to Sect. 8.2.4 (Pages 95–97). A pseudo vehicle identification and position replaces a failed sensing of a vehicle at a hub or link. Additional precautionary measures may be taken. ■

9.2.4 Scripts 468

By a script$_\delta$ we shall mean *a usually syntactic text which describes as set of actions expected to be taken by human actors of a system, including the assumptions under which these actions, or alternatives are to be taken.* 469

Example: 68 Pipeline System Scripts. We refer to Example 50. When closing a valve somewhere along a route all pumps upstream from the valve must first be shut down. Similarly when starting a pump somewhere along a route all valves downstream from the pump must first be opened. For a specific pipeline net this gives rise to a number of scripts, basically one for each pump and valve action. ■

9.2.5 Organisation & Management 470

[1] Organisation: By organisation$_\delta$ we shall mean *a partitioning of parts, actions and behaviours.*

Example: 69 Tollroad System Organisation. We refer to Sect. 8.2.4 (Pages 95–97). A simplest reasonable organisation is the set of links and hubs, including their sensors, and the monitor. ■ 471

[2] Management: By management$_\delta$ we shall mean *a partitioning of human staff into possibly a hierarchy strategy, tactics and operational managers, each taking care of the monitoring and*
472 *control of the rules & regulations for decreasing size sets of organisation partitions.*

Example: 70 Tollroad System Management. We refer to Sect. 8.2.4 (Pages 95–97). There is one strategic management structure for up to several tollroad systems. It is to be commonly described wrt., for example, policies of fixed or varying fee structures; etcetera. In the case of tollroad systems it seems reasonable to also have just one tactical management structure. It is to be commonly described wrt., for example, when to invoke one from a set of fee structures; etcetera. Etcetera.

9.2.6 Human Behaviour 473

By human behaviour$_\delta$ we shall mean *the sometimes diligent, sometimes sloppy, sometimes delinquent, or sometimes outright criminal carrying out of actions and behaviours of the domain.*

We omit giving examples.

9.3 What Needs More Research 474

MORE TO COME

9.3.1 Modelling Discrete & Continuous Domains

MORE TO COME

9.3.2 Domain Types and Signatures Form Galois Connections

We plan, in the Fall of 2012, to study whether an altogether different treatment of endurant domain entity types and perdurant domain entity signatures can illuminate the veracity of the title of this section.

9.3.3 A Theory of Domain Facets ?

We refer to Sect. 9.2. MORE TO COME

9.3.4 Other Issues

MORE TO COME

9.4 What Have We Achieved 475

We claim that there are four major contributions being reported upon: (i) strongly hinting that *domain types and signatures form Galois connections*, (ii) the separation of domain engineering from requirements engineering, (iii) the separate treatment of domain science & engineering: as "free-standing" with respect, ultimately, to computer science, and endowed
476 with quite a number of domain analysis principles and domain description principles; and (iv) 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
477 from domain descriptions to cast a different light on requirements engineering.

What we have not shown in this paper is the concept of domain facets; this concept is dealt with in [13] — 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.

9.5 General Remarks 478

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 479
"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 480

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 481
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 situa- 482
tion 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, deter-
mine values), etc. Nobody would hire non-skilled labour for the engineering development 483
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. 484

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.

9.6 Acknowledgements 485

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 paper. I in particular thank Dr. Magne Haveraaen 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. !

10 Bibliographical Notes 486

10.1 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] K. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley Publishing Company, 1989.

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

[6] J. Bayer, J.-M. DeBaud, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, and T. Widen. PuLSE: A Methodology to Develop Software Product Lines. In *Symposium on Software Reusability*, volume SSR'99, pages 122–131, May 1999.

[7] V. Benjamins and D. Fensel. The Ontological Engineering Initiative (KA)2. Internet publication + Formal Ontology in Information Systems, University of Amsterdam, SWI, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands and University of Karlsruhe, AIFB, 76128 Karlsruhe, Germany, 1998. http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.htm.

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

[9] D. Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.

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

.

[11] D. Bjørner. 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. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer.

[13] D. Bjørner. Domain Engineering. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[14] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[15] D. Bjørner. 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.

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

[17] 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.

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

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

[20] D. Bjørner and J. F. Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify" ? In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages 191–198. ICOT, June 1–5 1992.

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

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

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

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

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

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

[27] B. L. Clarke. A Calculus of Individuals Based on 'Connection'. *Notre Dame J. Formal Logic*, 22(3):204–218, 1981.

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

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

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

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

[32] 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.

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

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

[35] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[36] C. Fox. *The Ontology of Language: Properties, Individuals and Discourse*. CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.

[37] 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.

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

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

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

[41] C. A. Gunter, E. L. Gunter, M. A. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.

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

[43] M. Harsu. A Survey on Domain Engineering. Technical Report, Institute of Software Systems, Tampere University of Technology, Finland, 2002. P.O. Box 553, 33101 Tampere.

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

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

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

[47] T. Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [46]. See also `http://www.usingcsp.com/`.

[48] 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.

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

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

[51] 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.

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

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

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

[55] 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.

[56] S. Karlin and H. M. Taylor. *An Introduction to Stochastic Modeling*. Academic Press, 1998. ISBN 0-12-684887-4.

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

[58] S. Kripke. *Naming and Necessity*. Harvard University Press, Cambridge, MA, USA, 1980. (See also: http://plato.stanford.edu/entries/rigid-designators).

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

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

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

[62] H. S. Leonard and N. Goodman. The Calculus of Individuals and its Uses. *Journal of Symbolic Logic*, 5:45–44, 1940.

[63] S. Leśniewksi. 0 Podstawack Matematyki (Foundations of Mathematics). *Prezeglad Filosoficzny*, 30-34, 1927-1931.

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

[65] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [76].

[66] 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.

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

[68] E. Mettala and M. H. Graham. The Domain Specific Software Architecture Program. Project Report CMU/SEI-92-SR-009, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, June 1992.

[69] K. Michels, F. Klawonn, R. Kruse, and A. Nürnberger. *Fuzzy Control: Fundamentals, Stability and Design of Fuzzy Controllers*. Springer, 19 October 2010.

[70] D. Miéville and D. Vernant. *Stanisław Leśniewksi aujourd'hui*. Grenoble, October 8-10, 1992.

[71] R. Milne. RSL Proof Rules. Research Report RAISE/CRI/DOC/5/V1, CRI A/S, 30 March 1990.

[72] R. Milnes. Semantic Foundations for RSL. Research Report RAISE/CRI/DOC/4/V1, CRI A/S, 30 March 1990.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[89] P. M. Simons. *Parts: A Study in Ontology*. Clarendon Press, 1987.

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

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

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

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

[94] 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.

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

[96] R. Turner. *Truth and Modality for Knowledge Representation*. Pitman, 1990.

[97] R. Turner. *Computational Linguistics and Formal Semantics*, chapter Properties, Propositions and Semantic Theory, pages 159–180. Studies in Natural Langhuage Processing, eds. M. Rosner and R. Johnson. Cambridge University Press, 1992.

[98] 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.

[99] 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.

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

[101] A. C. Varzi. *On the Boundary between Mereology and Topology*, pages 419–438. Hölder-Pichler-Tempsky, Vienna, 1994.

[102] A. C. Varzi. *Spatial Reasoning in a Holey*[45] *World*, volume 728 of *Lecture Notes in Artificial Intelligence*, pages 326–336. Springer, 1994.

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

[45] holey: something full of holes

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

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

[106] 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.

Appendices

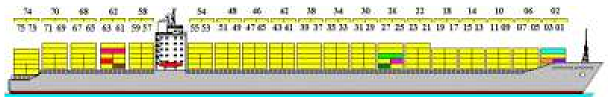
A A TripTych Ontology

186. domains Sect. 3 pg 36
  - a domain Sect. 3.1.1 pg 36
  - b domain phenomenon Sect. 3.1.2 pg 36
  - c domain entity Sect. 3.1.3 pg 36
  - d domain analysis Sect. 3.1.4 pg 36
  - e domain description Sect. 3.1.5 pg 37
  - f domain engineering Sect. 3.1.6 pg 37
  - g domain science Sect. 3.1.7 pg 37
  - h domain values and types Sect. 3.1.8 pg 37
  - i endurant entity Sect. 3.1.9 pg 36
  - j perdurant entity Sect. 3.1.10 pg 36
  - k discrete endurant Sect. 3.1.11 pg 36
  - l continuous endurant Sect. 3.1.12 pg 36
  - m discrete perdurant Sect. 3.1.13 pg 37
  - n continuous perdurant Sect. 3.1.14 pg 37
187. discrete endurant domain entities Sect. 4 pg 40
  - a parts Sect. 4.1 pg 40
    - i. abstract sorts Sect. 4.1.6 pg 41
    - ii. atomic parts Sect. 4.1.7 pg 41
    - iii. composite parts Sect. 4.1.8 pg 42
    - iv. part observers Sect. 4.1.9 pg 42
    - v. concrete types Sect. 4.1.10 pg 43
  - b part properties Sect. 4.2 pg 43
    - i. unique identifiers Sect. 4.2.1 pg 44
    - ii. mereology Sect. 4.2.2 pg 45
    - iii. attributes Sect. 4.2.3 pg 51
  - c states Sect. 4.3 pg 53
188. discrete perdurant domain entities Sect. 5 pg 57
  - a actions Sect. 5.2 pg 57
    - i. action signatures Sect. 5.2.3 pg 58
    - ii. action definitions Sect. 5.2.4 pg 58
  - b events Sect. 5.3 pg 61
    - i. event signatures Sect. 5.3.2 pg 61
    - ii. event predicate definitions Sect. 5.3.3 pg 61
  - c discrete behaviours Sect. 5.4 pg 62
    - i. behaviour signatures Sect. 5.4.4 pg 63
    - ii. behaviour definitions Sect. 5.4.5 pg 64
189. continous entities Sect. 6 pg 69
  - a materials Sect. 6.1 pg 69
    - i. materials-based domains Sect. 6.1.1 pg 69
    - ii. part/material relations Sect. 6.1.2 pg 69
    - iii. material observers Sect. 6.1.3 pg 70
    - iv. material properties Sect. 6.1.4 pg 71
    - v. laws of material flows and losses Sect. 6.1.5 pg 72
  - b continuous behaviours Sect. 6.2 pg 74

B On A Theory of Container Stowage 487

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.

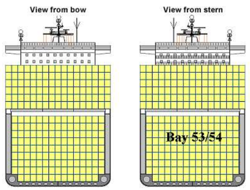
B.1 Some Pictures 488



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[46] 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. 489

Bay numbers.



Ship stowage cross section

Down along the vessel, horisontally, from front to aft, containers are grouped, in numbered bays. 490

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

[46] http://www.containerhandbuch.de/chb_e/stra/index.html?/chb_e/stra/stra_01_01_00.html

B.2 Parts 491

B.2.1 A Basis

190. From a container vessel (cv:CV) and from a container terminal port (ctp:CTP) one can observe their bays (bays:BAYS).

type
190. CV, CTP, BAYS
value
190. obs_BAYS: (CV|CTP) → BAYS

492

191. 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
191. BId, B
191. BS = BId $\overrightarrow{m}$ B
value
191. obs_BS: BAYS → BS (i.e., BId $\overrightarrow{m}$ B)

493

192. From a bay, b:B, one can observe its rows, rs:ROWS.
193. The rows, rs:RS, (of a bay) are mereologically structured as an (RId) indexed set of individual rows (r:R).

type
192. ROWS, RId, R
193. RS = RId $\overrightarrow{m}$ R
value
192. obs_ROWS: B → ROWS
193. obs_RS: ROWS → RS (i.e., RId $\overrightarrow{m}$ R)

494

194. From a row, r:R, one can observe its stacks, STACKS.
195. The stacks, ss:SS (of a row) are mereologically structured as an (SId) indexed set of individual stacks (s:S).

type
194. STACKS, SId, S
195. SS = SId $\overrightarrow{m}$ S
value
194. obs_STACKS: R → STACKS
195. obs_SS: STACKS → SS (i.e., SId $\overrightarrow{m}$ S)

495

196. A stack (s:S) is mereologically structured as a linear sequence of containers (c:C).

type
196. C
196. S = C*

The containers of the same stack index across stacks are called the tier at that index, cf. photo on Page 119.. 496

197. A container is here considered a composite part
   a of the container box, k:K
   b and freight, f:F.
198. 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.

497

type
197. C, K, F, P
value
197a. obs_K: C → K
197b. obs_F: C → F
198a. obs_Ps: F → P-**set**
type
198b. PI
198c. CI
value
198b. uid_P: P → PI
198c. uid_C: C → CI

B.2.2 Mereological Constraints 498

199. For any bay of a vessel the index sets of its rows are identical.
200. For a bay of a vessel the index sets of its stacks are identical.

axiom
199. ∀ cv:CV •
199. ∀ b:B•b ∈ **rng** obs_BS(obs_BAYS(cv))⇒
199. **let** rws=obs_ROWS(b) **in**
199. ∀ r,r′:R•{r,r′}⊆**rng** obs_RS(b)⇒**dom** r=**dom** r′
200. ∧ **dom** obs_SS(r) = **dom** obs_SS(r′) **end**

B.2.3 Stack Indexes 499

201. A container stack (and a container) is designated by an index triple: a bay index, a row index and a stack index.
202. A container index triple is valid, for a vessel, if its indices are valid indices.

type
201. StackId = BId×RId×SId
value
202. valid_address: BS → StackId → **Bool**
202. valid_address(bs)(bid,rid,sid) ≡
202. bid ∈ **dom** bs
202. ∧ rid ∈ **dom** (obs_RS(bs))(bid)
202. ∧ sid ∈ **dom** (obs_SS((obs_RS(bs))(bid)))(rid)

500

The above can be defined in terms of the below.

type
BayId = BId
RowId = BId×RId
value
202. valid_BayId: V → BayId → **Bool**
202. valid_BayId(v)(bid) ≡ bid ∈ **dom** obs_BS(obs_BAYS(v))

202. get_B: V → BayId $\xrightarrow{\sim}$ B
202. get_B(v)(bid) ≡ (get_B(bs))(bid) **pre**: valid_BId(v)(bid)

202. get_B: BS → BayId $\xrightarrow{\sim}$ B
202. get_B(bs)(bid) ≡ (obs_BS(obs_BAYS(v)))(bid) **pre**: bid ∈ **dom** bs

501

202. valid_RowId: V → RowId → **Bool**
202. valid_RowId(v)(bid,rid) ≡ rid ∈ **dom** obs_RS(get_B(v)(bid))
202. **pre**: valid_BayId(v)(bid)

202. get_R: V → RowId $\xrightarrow{\sim}$ R
202. get_R(v)(bid,rid) ≡ get_R(obs_BS(v))(bid,rid) **pre**: valid_RowId(v)(bid,rid)

202. get_R: BS → RowId $\xrightarrow{\sim}$ R
202. get_R(bs)(bid,rid) ≡ (obs_RS(get_RS(bs(bid))))(rid)
202. **pre**: valid_RowId(v)(bid,rid)

502

202. get_S: V → StackId $\xrightarrow{\sim}$ S
202. get_S(v)(bid,rid,sid) ≡ (obs_SS(get_R(get_B(v)(bid,rid))))(sid)
202. **pre**: valid_address(v)(bid,rid,sid)

503

202. get_C: V → StackId $\xrightarrow{\sim}$ C
202. get_C(v)(stid) ≡ get_C(obs_BS(v))(stid) **pre**: get_S(v)(bid,rid,sid) ≠ ⟨⟩

202. get_C: BS → StackId $\xrightarrow{\sim}$ C
202. get_C(bs)(bid,rid,sid) ≡ **hd**(obs_SS(get_R((bs(bid))(rid))))(sid)
202.   **pre**: get_S(bs)(bid,rid,sid) ≠ ⟨⟩

202. valid_addresses: V → StackId-**set**
202. valid_addresses(v) ≡ {adr|adr:StackId•valid_address(adr)(v)}

504

203. The predicate non_empty_designated_stack checks whether the designated stack is non-empty.

203. non_empty_designated_stack: V → StackId → **Bool**
203. non_empty_designated_stack(v)(bid,rid,sid) ≡ get_S(v)(bid,rid,sid) ≠ ⟨⟩

505

204. Two vessels have the same mereology if they have the same set of valid-addresses.

value
204. unchanged_mereology: BS × BS → **Bool**
204. unchanged_mereology(bs,bs′) ≡ valid_addresses(bs) = valid_addresses(bs′)

506

205. 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.

205. popped_designated_stack: BS × BS → StackId → **Bool**
205. popped_designated_stack(bs,bs′)(stid) ≡
205a.   **tl** get_S(v)(stid) = get_S(bs′)(stid)

507

206. 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.

206. unchanged_non_designated_stacks: BS × BS → StackId → **Bool**
206. unchanged_non_designated_stacks(bs,bs′)(stid) ≡
206a.   ∀ adr:StackId•adr ∈ valid_addresses(v)\{stid}⇒
206a.   get_S(bs)(adr) = get_S(bs′)(adr)
206.   **pre**: unchanged_mereology(bs,bs′)

B.2.4 Stowage Schemas

508

207. 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.[47]

509

value
207. obs_StoSchema: V → StoSchema
type
207a. StoSchema = BId $\overrightarrow{m}$ BaySchema
207b. BaySchema = RId $\overrightarrow{m}$ RowSchema
207c. RowSchema = SId $\overrightarrow{m}$ StaSchema
207d. StaSchema = **Nat** $\overrightarrow{m}$ C_Type
207e. C_Type
axiom
207f. ∀ stsc:StaSchema • **dom** stsc = {1..**max dom** stsc}

510

208. One can define a function which from an actual vessel "derives" its "current stowage schema".

208. cur_sto_schema: V → StoSchema
208. cur_sto_schema(v) ≡
208.   **let** bs = obs_BS(obs_BAYS(v)) **in**
208.   [bid ↦ **let** rws = obs_RS(obs_ROWS(bs(bid))) **in**
208.   [rid ↦ **let** ss = obs_SS(obs_STACKS(rws)(rid)) **in**
208.   [sid ↦ ⟨ analyse_container(ss(i))|i:**Nat**•i ∈ **inds** ss ⟩
208. | sid:SId•sid ∈ ss] **end**
208. | rid:RId•rid ∈ **dom** rws] **end**
208. | bid:BId•bid ∈ **dom** ds] **end**

208. analyse_container: C → C_Type

511

209. Given a stowage schema and a current stowage schema one can check the latter for conformance wrt. the former.

[47]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.

209. conformance: StoSchema × StoSchema → **Bool**
209. conformance(stosch,cur_stosch) ≡
209.   **dom** cur_stosch = **dom** stosch
209. ∧ ∀ bid:BId • bid ∈ **dom** stosch ⇒
209.   **dom** cur_stosch(bid) = **dom** stosch(bid)
209.   ∧ ∀ rid:RId • rid ∈ **dom**(stosch(bid))(rid) ⇒
209.   **dom**(cur_stosch(bid))(rid) = **dom**(stosch(bid))(rid)
209.   ∧ ∀ sid:SId • sid ∈ **dom**(cur_stosch(bid))(rid)
209.   ∀ i:**Nat** • i ∈ **inds**((cur_stosch(bid))(rid))(sid) ⇒
209.   conform((((cur_stosch(bid))(rid))(sid))(i),
209.   (((stosch(bid))(rid))(sid))(i))

209. conform: C_Type × C_Type → **Bool**

512

210. From a vessel one can observe its mandated stowage schema.

211. The current stowage schema of a vessel must always conform to its mandated stowage schema.

value
210. obs_StoSchema: V → StoSchema

211. stowage_conformance: V → **Bool**
211. stowage_conformance(v) ≡
211.   **let** mandated = obs_StoSchema(v),
211.   current = cur_sto_schema(v) **in**
211.   conformance(mandated,current) **end**

B.3 Actions 513

B.3.1 Remove Container from Vessel

106. The remove_Container_from_Vessel action applies to a vessel and a stack address and conditionally yields an updated vessel and a container.

106a. 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 106d).

106b. The remove_C_from_BS function yields a pair: an updated set of bays and a container.

106c. 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).

106d. 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.

107. The pre-condition for remove_C_from_BS(bs)(stid) is

107a. that stid is a valid_address in bs, and

107b. that the stack in bs designated by stid is non_empty.

108. The post-condition for remove_C_from_BS(bs)(stid) wrt. the updated bays, bs′, is

108a. that the yielded container, i.e., c, is obtained, get_C(bs)(stid), from the top of the non-empty, designated stack,

108b. that the mereology of bs′ is unchanged, unchanged_mereology(bs,bs′). wrt. bs.

108c. that the stack designated by stid in the "input" state, bs, is popped, popped_designated_stack(bs,bs′)(stid), and

108d. that all other stacks are unchanged in bs′ wrt. bs, unchanged_non_designated_stacks(bs,bs′)(stid).

514

value
106. remove_C_from_V: V → StackId $\xrightarrow{\sim}$ (V×C)
106. remove_C_from_V(v)(stid) **as** (v′,c)
106c.   (**obs_**Bs(**obs_**BS(v′),c)) = remove_C_from_BS(**obs_**Bs(**obs_**BS(v)))(stid)
106d.   ∧ props(v)=props(v″)

106b. remove_C_from_BS: BS → StackId → (BS×C)
106a. remove_C_from_BS(bs)(stid) **as** (bs′,c)
107a.   **pre**: valid_address(bs)(stid)
107b.   ∧ non_empty_designated_stack(bs)(stid)
108a.   **post**: c = get_C(bs)(stid)
108b.   ∧ unchanged_mereology(bs,bs′)
108c.   ∧ popped_designated_stack(bs,bs′)(stid)
108d.   ∧ unchanged_non_designated_stacks(bs,bs′)(stid)

The props function was introduced in Sect. 4.2.5 on Page 52.

B.3.2 Remove Container from CTP 515

We define a remove action similar to that of Sect. B.3.1 on the previous page.

212. 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 106c and 106d.

value
212. remove_C_from_CTP: CTP → StackId $\xrightarrow{\sim}$ (CTP×C)
212. remove_C_from_CTP(ctp)(stid) **as** (ctp′,c)
106c.   (obs_BS(ctp′),c) = remove_C_from_BS(obs_BS(ctp))(stid)
106d.   ∧ props(ctp)=props(ctp″)

B.3.3 Stack Container on Vessel 516

213. Stacking a container at a vessel bay stack location

a

b

c

value
213. stack_C_on_vessel: BS → StackId $\stackrel{\sim}{\to}$ C $\stackrel{\sim}{\to}$ BS
213a. stack_C_on_vessel(bs)(stid)(c) **as** bs′
213a. **comment:** bs is bays of a v:V, i.e., bs = obs_BS(v)
213b. **pre**:
213c. **post**:

B.3.4 Stack Container in CTP 517

214.

215.

216.

217.

value
214. stack_C_in_CTP: CTP → StackId → C $\stackrel{\sim}{\to}$ CTP
215. stack_C_in_CTP(ctp)(stid)(c) **as** ctp′
216. **pre**:
217. **post**:

B.3.5 Transfer Container from Vessel to CTP 518

218.

219.

220.

221.

value
218. transfer_C_from_V_to_CTP: V→StackId$\stackrel{\sim}{\to}$CTP→StackId$\stackrel{\sim}{\to}$(V×CTP)
219. transfer_C_from_V_to_CTP(v)(v_stid)(ctp)(ctp_stid) ≡
220. **let** (c,v′) = remove_C_from_V(v)(v_stid) **in**
220. (v′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

B.3.6 Transfer Container from CTP to Vessel 519

222.

223.

224.

value
222. transfer_C_from_CTP_to_V: CTP→StackId$\stackrel{\sim}{\to}$V→StackId$\stackrel{\sim}{\to}$(CTP×V)
223. transfer_C_from_CTP_to_V(ctp)(ctp_stid)(v)(v_stid) ≡
224. **let** (c,ctp′) = remove_C_from_CTP(ctp)(ctp_stid) **in**
224. (ctp′,stack_C_in_CTP(ctp)(ctp_stid)(c)) **end**

C Indexes

520

C.1 RSL Index

C.2 Formalisation Index

C.3 Definition Index

C.4 Example Index

C.5 Concept Index

C.6 Language, Method and Technology Index

C.7 Selected Author Index

D RSL: The Raise Specification Language 521

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. 522

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 523

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. 524

[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 525
[12] A $\overrightarrow{m}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following 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.

[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 526

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′ • $\mathcal{P}$(v) |}

527

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 528

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 529

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

type
A, B, ..., C

D.3 The RSL Predicate Calculus 530

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 531

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 532

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 533

D.4.1 Arithmetic

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

538

D.4.2 Set Expressions 534

[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**

535

[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 $\stackrel{\sim}{\to}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}

540

D.4.3 Cartesian Expressions 536

[1] Cartesian Enumerations: Let e range over values of Cartesian types involving $A, B, \ldots, C$, then the below expressions are simple Cartesian enumerations:

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

D.4.4 List Expressions 537

[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,... ⟩, ...} ∈ A^ω

⟨ 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.

[2] List Comprehension: The last line below expresses list comprehension.

type
  A, B, P = A → **Bool**, Q = A $\stackrel{\sim}{\to}$ B
value
  comprehend: A^ω × P × Q $\stackrel{\sim}{\to}$ B^ω
  comprehend(l,P,Q) ≡
    ⟨ Q(l(i)) | i **in** ⟨1..**len** l⟩ • P(l(i))⟩

D.4.5 Map Expressions 539

[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 $\overrightarrow{m}$ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [ ], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M

[2] Map Comprehension: The last line below expresses map comprehension:

type
  U, V, X, Y
  M = U $\overrightarrow{m}$ V
  F = U $\xrightarrow{\sim}$ X
  G = V $\xrightarrow{\sim}$ Y
  P = U → **Bool**
value
  comprehend: M×F×G×P → (X $\overrightarrow{m}$ Y)
  comprehend(m,F,G,P) ≡
    [F(u) ↦ G(m(u)) | u:U • u ∈ **dom** m ∧ P(u)]

D.4.6 Set Operations 541

[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** $\xrightarrow{\sim}$ **Nat**

542

[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

543

544

[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.
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.

545

[4] Set Operator Definitions: The operations can be defined as follows (≡ is the definition symbol):

value
  $s' \cup s'' \equiv$ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  $s' \cap s'' \equiv$ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  $s' \setminus s'' \equiv$ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  $s' \subseteq s'' \equiv$ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  $s' \subset s'' \equiv$ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  $s' = s'' \equiv$ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  $s' \neq s'' \equiv$ 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 546

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 547

[1] List Operator Signatures:

value
  hd: $A^\omega \stackrel{\sim}{\to}$ A
  tl: $A^\omega \stackrel{\sim}{\to} A^\omega$
  len: $A^\omega \stackrel{\sim}{\to}$ **Nat**
  inds: $A^\omega \to$ **Nat-infset**
  elems: $A^\omega \to$ A-**infset**
  .(.): A^ω × **Nat** $\stackrel{\sim}{\to}$ A
  ⌢: A* ≠ A^ω A^ω A^ω A^ω **BoBbol**

548

[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}
  ⟨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⟩

549

[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 elements larger than or equal to i, gives the ith element of the list. 550
- ⌢: 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.

551 The operations can also be defined as follows:

[4] List Operator Definitions:

value
  is_finite_list: $A^\omega \to$ **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 }

552

  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 553

[1] Map Operator Signatures and Map Operation Examples:

value

$\mathrm{m(a)}\colon \mathrm{M} \to \mathrm{A} \xrightarrow{\sim} \mathrm{B},\ \mathrm{m(a)} = \mathrm{b}$

$\textbf{dom}\colon \mathrm{M} \to \mathrm{A}\textbf{-infset}$ [domain of map]

$\quad \textbf{dom}\ [\mathrm{a1}\mapsto\mathrm{b1},\mathrm{a2}\mapsto\mathrm{b2},...,\mathrm{an}\mapsto\mathrm{bn}] = \{\mathrm{a1},\mathrm{a2},...,\mathrm{an}\}$

$\textbf{rng}\colon \mathrm{M} \to \mathrm{B}\textbf{-infset}$ [range of map]

$\quad \textbf{rng}\ [\mathrm{a1}\mapsto\mathrm{b1},\mathrm{a2}\mapsto\mathrm{b2},...,\mathrm{an}\mapsto\mathrm{bn}] = \{\mathrm{b1},\mathrm{b2},...,\mathrm{bn}\}$

$\dagger\colon \mathrm{M} \times \mathrm{M} \to \mathrm{M}$ [override extension]

$\quad [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}''] \dagger [\mathrm{a}'\mapsto\mathrm{b}'',\mathrm{a}''\mapsto\mathrm{b}'] = [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}'',\mathrm{a}''\mapsto\mathrm{b}']$

554

$\cup\colon \mathrm{M} \times \mathrm{M} \to \mathrm{M}$ [merge $\cup$]

$\quad [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}''] \cup [\mathrm{a}'''\mapsto\mathrm{b}'''] = [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'',\mathrm{a}'''\mapsto\mathrm{b}''']$

$\backslash\colon \mathrm{M} \times \mathrm{A}\textbf{-infset} \to \mathrm{M}$ [restriction by]

$\quad [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']\backslash\{\mathrm{a}\} = [\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']$

$/\colon \mathrm{M} \times \mathrm{A}\textbf{-infset} \to \mathrm{M}$ [restriction to]

$\quad [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']/\{\mathrm{a}',\mathrm{a}''\} = [\mathrm{a}'\mapsto\mathrm{b}',\mathrm{a}''\mapsto\mathrm{b}'']$

$=,\neq\colon \mathrm{M} \times \mathrm{M} \to \textbf{Bool}$

$^\circ\colon (\mathrm{A} \underset{m}{\to} \mathrm{B}) \times (\mathrm{B} \underset{m}{\to} \mathrm{C}) \to (\mathrm{A} \underset{m}{\to} \mathrm{C})$ [composition]

$\quad [\mathrm{a}\mapsto\mathrm{b},\mathrm{a}'\mapsto\mathrm{b}'] \circ [\mathrm{b}\mapsto\mathrm{c},\mathrm{b}'\mapsto\mathrm{c}',\mathrm{b}''\mapsto\mathrm{c}''] = [\mathrm{a}\mapsto\mathrm{c},\mathrm{a}'\mapsto\mathrm{c}']$

555

[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 of the left operand map to the elements that are not in the right operand set.

556

- /: 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.

557

[3] Map Operation Redefinitions: The map operations can also be defined as follows:

value

$\textbf{rng}\ \mathrm{m} \equiv \{\ \mathrm{m(a)} \mid \mathrm{a{:}A} \bullet \mathrm{a} \in \textbf{dom}\ \mathrm{m}\ \}$

$\mathrm{m1} \dagger \mathrm{m2} \equiv$

$\quad [\ \mathrm{a}\mapsto\mathrm{b} \mid \mathrm{a{:}A},\mathrm{b{:}B} \bullet$

$\qquad \mathrm{a} \in \textbf{dom}\ \mathrm{m1} \setminus \textbf{dom}\ \mathrm{m2} \land \mathrm{b}=\mathrm{m1(a)} \lor \mathrm{a} \in \textbf{dom}\ \mathrm{m2} \land \mathrm{b}=\mathrm{m2(a)}\ ]$

$\mathrm{m1} \cup \mathrm{m2} \equiv [\ \mathrm{a}\mapsto\mathrm{b} \mid \mathrm{a{:}A},\mathrm{b{:}B} \bullet$

$\qquad\qquad \mathrm{a} \in \textbf{dom}\ \mathrm{m1} \land \mathrm{b}=\mathrm{m1(a)} \lor \mathrm{a} \in \textbf{dom}\ \mathrm{m2} \land \mathrm{b}=\mathrm{m2(a)}\ ]$

$\mathrm{m} \setminus \mathrm{s} \equiv [\ \mathrm{a}\mapsto\mathrm{m(a)} \mid \mathrm{a{:}A} \bullet \mathrm{a} \in \textbf{dom}\ \mathrm{m} \setminus \mathrm{s}\ ]$

$\mathrm{m} / \mathrm{s} \equiv [\ \mathrm{a}\mapsto\mathrm{m(a)} \mid \mathrm{a{:}A} \bullet \mathrm{a} \in \textbf{dom}\ \mathrm{m} \cap \mathrm{s}\ ]$

$\mathrm{m1} = \mathrm{m2} \equiv$

$\quad \textbf{dom}\ \mathrm{m1} = \textbf{dom}\ \mathrm{m2} \land \forall\ \mathrm{a{:}A} \bullet \mathrm{a} \in \textbf{dom}\ \mathrm{m1} \Rightarrow \mathrm{m1(a)} = \mathrm{m2(a)}$

$\mathrm{m1} \neq \mathrm{m2} \equiv \sim(\mathrm{m1} = \mathrm{m2})$

$\mathrm{m}^\circ\mathrm{n} \equiv$

$\quad [\ \mathrm{a}\mapsto\mathrm{c} \mid \mathrm{a{:}A},\mathrm{c{:}C} \bullet \mathrm{a} \in \textbf{dom}\ \mathrm{m} \land \mathrm{c} = \mathrm{n(m(a))}\ ]$

$\quad \textbf{pre rng}\ \mathrm{m} \subseteq \textbf{dom}\ \mathrm{n}$

D.5 λ-Calculus + Functions 558

D.5.1 The λ-Calculus Syntax

type /* A BNF Syntax: */

$\quad \langle \mathrm{L} \rangle ::= \langle \mathrm{V} \rangle \mid \langle \mathrm{F} \rangle \mid \langle \mathrm{A} \rangle \mid (\ \langle \mathrm{A} \rangle\ )$

$\quad \langle \mathrm{V} \rangle ::=$ /* variables, i.e. identifiers */

$\quad \langle \mathrm{F} \rangle ::= \lambda\langle \mathrm{V} \rangle \bullet \langle \mathrm{L} \rangle$

$\quad \langle \mathrm{A} \rangle ::= (\ \langle \mathrm{L} \rangle\langle \mathrm{L} \rangle\ )$

value /* Examples */

$\quad \langle \mathrm{L} \rangle$: e, f, a, ...

$\quad \langle \mathrm{V} \rangle$: x, ...

$\quad \langle \mathrm{F} \rangle$: λ x • e, ...

$\quad \langle \mathrm{A} \rangle$: f a, (f a), f(a), (f)(a), ...

D.5.2 Free and Bound Variables 559

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 560

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]($\lambda x \bullet P$)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ $\lambda y \bullet$ **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 561

- 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 562

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 563

Functions can be defined explicitly:

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

  g: Arguments $\stackrel{\sim}{\to}$ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)

564

Or functions can be defined implicitly:

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

  g: Arguments $\stackrel{\sim}{\to}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

The symbol $\stackrel{\sim}{\to}$ 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 565

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 566

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 567

Predicative **let** expressions:

```
let a:A • 𝒫(a) in 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 568

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 569

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 570

```
⟨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 571

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 572

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

D.7.3 Statement Sequences and skip 573

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 574

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 575

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

D.7.6 Iterative Sequencing 576

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

D.8 Process Constructs 577

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 578

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 579

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 580

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 581

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.[48]

[48] For schemes, classes and objects we refer to [9, Chap. 10]