From Domains to Requirements Lecture Notes in Software Engineering Budapest, 11–22 October 2010 1

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com – www.imm.dtu.dk/~db

Begun: Tuesday June 22, 2010. Compiled: November 12, 2010: 11:28

Abstract 2

We present "standard" domain description and requirements prescription examples using the RAISE [112] Specification Language, RSL [110]. The illustrated example is that of transportation networks.

These notes shalll serve as lecture notes for my lectures at Uppsala, Nov.8-19, 2010. The present document is the ordinary "book-form"-like notes. A separate document, compiled from the same files, present 11 sets of lecture slides. The "funny" small numbers you see in the present document, in margins and at almost end of display lines refer to slide page numbers of the slides document.

4

Lecture Notes 3

A Tentative Lecture Schedule

Contents

1 Introduction 5

1.1 The Problem

The problem to be solved by this technical note is to present in one specific formal specification language, RSL [112], a domain description and a requirements prescription developed according to the "triptych approach" [34].

1.2 General Remarks

Before we can design software we must have a robust understanding of its requirements. And before we can prescribe requirements we must have a robust understanding of the environment, or, as we shall call it, the domain in which the software is to serve – and as it is at the time such software is first being contemplated.

In consequence we suggest that software, "ideally"[1], be developed in three phases.

First a phase of **domain engineering.** In this phase a reasonably comprehensive description is constructed from an analysis of the domain. That description, as it evolves, is analysed with respect to inconsistencies, conflicts and completeness on one hand, and, on the other hand, in order to achieve pleasing concepts in terms of which to abstractly model the domain (Sect. 3).

Then a phase of **requirements engineering.** This phase is strongly based, as we shall see (in Sect. 4), on an available, necessary and sufficient domain description. Guided by the domain and requirements engineers the *requirements stakeholders* point out which domain description parts are to be left (*projected*) out of the *domain requirements*, and of those left what forms of *instantiations*, *determinations* and *extensions* are required. Similarly the requirements stakeholders, guided by the domain and requirements engineers, inform as to which domain *entities*, *actions*, *events* and *behaviours* are *shared* between the domain and the *machine*, that is, the *hardware* and the *software* being required. In these notes we shall only very briefly cover aspects of *machine requirements*.

And finally a phase of **software design.** We shall not cover this phase in these notes.

Methodology

These notes focus on methodology – where a method is seen as a set of principles (applied by engineers, not machines) for selecting and applying (often with some tool support) techniques (and tools) for the efficient construction of some artifact – here software.

1.2.1 What are Domains

By a domain we shall thus understand a universe of discourse, an area of nature subject to laws of physics and studies by physicists, or an area of human activity (subject to its interfaces with nature). There are other domains which we shall ignore. We shall focus on the human-made domains. "Large scale" examples are *the financial service industry:*

[1]Section 5.7 will discuss practical renditions of "idealism"!

banking, insurance, securities trading, portfolio management, etc., *health care: hospitals, clinics, patients, medical staff, etc.*, *transportation:* road, rail/train, sea, and air transport (vehicles, transport nets, etc.); *oil and gas systems:* pumps, pipes, valves, refineries, distribution, etc. "Intermediate scale" examples are *automobiles:* manufacturing or monitoring and control, etc.; and *heating systems*.

The above explication was "randomised": for some domains, to wit, *the financial service industry*, we mentioned major functionalities, for others, to wit, *health care*, we mentioned major entities. An objection can be raised, namely that the above characterisation – of what a domain is – is not sufficiently precise. We shall try, in the next section, to partially meet this objection.

1.2.2 What is a Domain Description

By a *domain description* we understand a description of the *entities*, the *actions*, the *events* and the *behaviours* of the domain, including its *interfaces* to other domains. A domain description describes the domain **as it is**. A domain description does not contain requirements let alone references to any software. Michael Jackson, in [138], refers to domain descriptions as *indicative* (stating objective fact), requirements prescriptions as *optative* (expressing wish or hope) and software specifications as *imperative* ("do it!"). A description is *syntax*. The meaning (*semantics*) of a domain description is usually a set of *domain models*. We shall take domain models to be *mathematical structures (theories)*. The form of domain descriptions that we shall advocate "come in pairs": precise, say, English, i.e., narrated text (narratives) alternates with clearly related formula text.

Description Languages Besides using as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements, we "pair" such narrative elements with corresponding enumerated clauses of a formal specification language. We shall be using the `RAISE` Specification Language, `RSL`, [112], in our formal texts. But any of the model-oriented approaches and languages offered by `Alloy` [137], `Event B` [3], `VDM` [107] and `Z` [233], should work as well. No single one of the above-mentioned formal specification languages, however, suffices. Often one has to carefully combine the above with elements of `Petri Nets` [199], `CSP: Communicating Sequential Processes` [128], `MSC: Message Sequence Charts` [136], `Statecharts` [120], and some temporal logic, for example either `DC: Duration Calculus` [235] or `TLA+` [147]. Research into how such diverse textual and diagrammatic languages can be meaningfully and proof-theoretically combined is ongoing [9].

1.2.3 Contributions of These Lecture Notes 7

We claim that the major contributions of the triptych approach to software engineering 8
as presented in these notes are the following: (1) the clear *identification* of domain engineering, or, for some, its clear *separation* from requirements engineering (Sects. 3 and 4); (2) the *identification* and *'elaboration'* of the pragmatically determined domain *facets* of

(a) intrinsics, (b) support technologies, (c) rules and regulations, (d) scripts (licenses and contracts), (e) management and organisation, and *(f) human behaviour* whereby 'elaboration' we mean that we provide principles and techniques for the construction of these facet description parts (Sects. 3.1–3.6); (3) the *re-identification* and *'elaboration'* of the concept of *business process reengineering* (Sect. 4.1); (4) the *identification* and *'elaboration'* of the technically determined *domain requirements facets* of *(g) projection, (h) instantiation, (i) determination, (j) extension* and *(k) fitting* requirements principles and techniques – and, in particular the *"discovery"* that these requirements engineering stages are strongly dependent on necessary and sufficient domain descriptions (Sects. 4.2.1–4.2.5); and (5) the *identification* and *'elaboration'* of the technically determined *interface requirements facets* of *(l) shared simple entity, (m) shared action, (n) shared event* and *(o) shared behaviour* requirements principles and techniques (Sects. 4.3.2–4.3.5). We claim that the facets of (2, 3, 4) and (5) are all relatively new.

1.2.4 Relation to Other Engineering Disciplines

An aeronautics engineer – to be hired by `Boeing` to their design team for a next generation aircraft – must be pretty well versed in applied mathematics and in aerodynamics. A radio communications engineer – to be hired by `Ericsson` to their design team for a next generation mobile telephony antennas – must be likewise pretty well versed in applied mathematics and in the physics of electromagnetic wave propagation in matter. And so forth. Software engineers hired for the development of software for hospitals, or for railways, know little, if anything, about health care, respectively rail transportation (scheduliming, rostering, etc.). The `Ericsson` radio communications engineer can be expected to understand Maxwell's Equations, and to base the design of antenna characteristics on the transformation and instantiation of these equations. It is therefore quite reasonable to expect the domain-specific software engineer to understand formalisation of their domains, to wit: *railways:* `www.railwaydomain.org`, and *pipelines:* `pipelines.pdf`, *logistics:* `logistics.pdf`, *transport nets:* `comet1.pdf`, *stock exchanges:* `tse-2.pdf` and *container lines:* `container-paper.pdf` – these latter five at `www.imm.dtu.dk/~db/`.

1.3 The Triptych Approach 6

The "triptych approach" calls for a thorough description (cum analysis) of the domain before one attempts prescribing requirements for specific software.

As part of the triptych approach to domain engineering one starts by exploring the description ontology of specification entities: *simple entities, actions, events* and *behaviours* (Sect. 2) before delving into the description ontology of facets: *intrinsics, support technologies, rules & regulations, scripts (licenses and contracts), management & organisation* and *human behaviour* (Sect. 3).

And, as part of the triptych approach to requirements engineering one starts by exploring the reengineering of business processes before delving into *domain requirements* concepts of *projection, instantiation, determination, extension* and *fitting* – followed by a number of *interface requirements* stages. The *terms in slanted script* are defined in Appendix B.

For a more pedagogic and didactical introduction to these terms we refer to either of [36, 50, 49, 45, 46, 47] or to [34, 39, 44].

1.4 On The Structure of These Lecture Notes

The presentation (i.e., structuring) of the technical material of these lecture nptes is not meant to suggest that all domain descriptions and requirements prescriptions follow this mold. As mentioned just above our presentation follows the structure of *simple entity, action, event,* and *behaviour* specification ontology (Sect. 2), then the structure of the *domain facets: intrinsics, support technology, rules & regulations, scripts (licenses, contracts), management & organisation* and *human behaviour* (Sect. 3), and finally the structure of the *business re-engineering* (Sect. 4.1.3), the *domain requirements* concepts of *projection, instantiation, determination, extension* and *fitting* (Sect. 4.2), and a number of *interface requirements* facets (Sect. 4.3).

I expect such students who might be pursuing specifications based on the example of this document to do so, either, as here, in `RSL` [110], or according to approaches embodied in `Alloy` [137], `CafeOBJ` [109], `Event B` [3], `VDM` [107] and `Z` [233]. But I do not expect them to follow exactly the order used in this document – although it migt well be a good idea, pedagogically and didactically.

Two remarks are in order:

- Rather I expect `Alloy, CafeOBJ, Event B, VDM-SL` and `Z` specifications to follow a "most natural order" appropriate for their approaches.
- The order in which I have chosen to present the current material reflects a both pedagogic and didactic views.[2] In a commercial project I might very well choose another decomposition of the material — being guided, however, by the need to cover all the footnoted (Footnote 2) facets.

1.5 The Comparative Methodology Endeavour

These notes are intended to replace:

- http://www.imm.dtu.dk/˜db/bjorner-8jan2010.pdf
- http://www.complang.tuwien.ac.at/bjorner/book.pdf

which were first suggested as a basis for the Comparative Methodology endeavour, cf.

[2]The sequence of the *simple entity, action, event, behaviour, domain facets: intrinsics, support technology, rules & regulations, scripts (licenses, contracts), management & organisation, human behaviour, domain requirements: projection, instantiation, determination, extension, fitting* and the *interface requirements* facets reflect these views.

- http://www2.imm.dtu.dk/˜db/comet/
- http://formalmethods.wikia.com/wiki/CoMet .

Rewriting the above referenced earlier notes into the present notes were begun after Kokichi Futatsugi's CafeOBJ lectures. I am happy to acknowledge being thus challenged.

1.6 Caveat

The many examples of Sect. A, the `RSL` Primer, stem from an earlier version of this attempt to give a 'model' presentation of domains and requirements. They have yet to coordinated with the the present rewrite of Sects. 2–4.

2 An Ontology of Specification Entities 9

Definition: Ontology. *In philosophy: A systematic account of Existence. To us: An explicit formal specification of how to represent the phenomena and concepts that are assumed to exist in some area of interest (some universe of discourse) and the relationships that hold among them. Further clarification: An ontology is a catalogue of concept[152]s and their relationships — including properties as relationships to other concepts.* 10
Definition: Specification. *We use the term 'specification' to cover the concepts of domain description[243]s, requirements prescription[615]s and software design[688]s. More specifically a specification is a definition[210], usually consisting of many definitions.*
Definition: Entity. *By an entity we shall understand either a simple entity[681]*[3]*, an action[12], an event[281] or a behaviour[79].*

2.1 Simple Entities 11

Definition: Simple Entity. *By a simple entity we shall loosely understand an individual, static[708] or inert[367] dynamic[260] and that simple entities "roughly correspond" to what we shall think of as value[802]s. We shall further allow simple entities to be either atomic[63] or composite[133], i.e., in the latter case having decomposable sub-entities. Simple entities have attribute[69]s. Composite entities have attribute[69]s, sub-entities and a mereology[451], the latter explains how the sub-entities are formed into the simple entity.* 12

2.1.1 Net, Hubs and Links 13

1. There are nets, hubs and links.
2. A net contains zero, one or more hubs.
3. A net contains zero, one or more links.

16

type
  1. N, H, L
value
  2. obs_Hs: N → H-**set**
  3. obs_Ls: N → L-**set**

2.1.2 Unique Hub and Link Identifiers 14

4. There are hub identifiers and there are link identifiers.
5. From a hub one can observe its hub identifier.
6. From a link one can observe its link identifier.

[3]The superscript [bracketed numbers] refer to Sect. B's Item 681 on page 217.

7. Hubs of a net have unique hub identifiers.
8. Links of a net have unique hub identifiers.

type
  4. HI, LI
value
  5. obs_HI: H → HI
  6. obs_LI: L → LI
axiom
  7. $\forall$ n:N, h,h′:H • {h,h′}⊆obs_Hs(n) ∧ h≠h′ ⇒ obs_HI(h)≠obs_HI(h′)
  8. $\forall$ n:N, l,l′:L • {l,l′}⊆obs_Ls(n) ∧ l≠l′ ⇒ obs_LI(l)≠obs_LI(l′)

2.1.3 Observability of Hub and Link Identifiers 15

9. From every hub (of a net) we can observe the identifiers of the zero, one or more distinct links (of that net) that the hub is connected to.

value
  9. obs_LIs: H → LI-**set**
axiom
  9. ∀ n:N,h:H•h ∈ obs_Hs(n) ⇒ ∀ li:LI•li ∈ obs_LIs(h) ⇒ L_exists(li)(n)
value
  L_exists: LI → N → **Bool**
  L_exists(li)(n) ≡ ∃ l:L•l ∈ obs_Ls(n)∧obs_LI(l)=li

10. From every link (of a net) we can observe the identifiers of the exactly two (distinct) hubs (of that net) that the link is connected to.

value
  10. obs_HIs: L → HI-**set**
axiom
  10. ∀ n:N,l:L•l ∈ obs_Ls(n) ⇒
  10.   **card** obs_HIs(l)=2 ∧ ∀ hi:HI•hi ∈ obs_HIs(l) ⇒ H_exists(hi)(n)
value
  H_exists: HI → N → **Bool**
  H_exists(hi)(n) ≡ ∃ h:H•h ∈ obs_Hs(n)∧obs_HI(h)=hi

2.1.4 A Theorem 17

Links implies Hubs

11. It follows from the above that if a net has at least one link then it has at least two hubs.

theorem:

11. $\forall$ n:N • **card** obs_Ls(n)$\geq$1 $\Rightarrow$ **card** obs_Hs(n)$\geq$2

2.1.5 Hub and Link Attributes 18

In preparation for later descriptions, narrative and formal, we make a slight detour to deal with hub and link attributes – but we omit, at present, from describing these attributes.

12. hub and link attributes, HAtrs and LAtrs, include the hub and link identifiers that can be observed from hubs and links, respecively.
13. These can be observed from hubs and links of nets.
14. And these can be provided as arguments when construction hubs and links.

```
type
12.  HAtrs, LAtrs
value
13.  obs_HAtrs: H → HAtrs
14.  obs_LAtrs: L → LAtrs
13.  obs_HI: HAtrs → HI
13.  obs_LIs: HAtrs → LI-set
14.  obs_LI: LAtrs → LI
14.  obs_HIs: LAtrs → HI-set
```

2.1.6 Hub and Link Generators 19

15. From [a (full) set of] hub attributes
    a) including an empty set of observable link identifiers

    one can generate a hub with

    a) the hub identifier being that of the argument hub attributes,
    b) the link identifiers of the hub being argument the empty set of link identifiers of the hub attributes and
    c) the argument hub attributes being those of the resulting hub,

```
15.  genH: HAtrs → H
15.  genH(hatrs) as h
15a.   pre  obs_LIs(hatrs)={}
15a.   post obs_HI(h)=obs_HI(hatrs)
15b.        ∧ obs_LIs(h)={}
15c.        ∧ obs_HAtrs(h)=hatrs
```

20

16. From the set of hub attributes and a net one can “similarly” generate a hub which is not a hub of the net.
17. From the set of link attributes one can “similarly” generate a link.
18. From the set of link attributes and a net one can “similarly” generate a link which is not a link of the net.

21

where the reader is to narrate and formalise the “similarities”!

```
16.  genH: HAtrs → N → H
16.  genH(hatrs)(n) as h
16.    pre obs_LIs(hatrs)={}
16.       ∧ ∼∃ h′:H•h′ ∈ obs_Hs(n) ∧ obs_HI(h′)=obs_HI(hatrs)
16.    post  h ∉ obs_Hs(n)
16.       ∧ obs_HI(h)=obs_HI(hatrs)
16.       ∧ obs_LIs(h)={}
16.       ∧ obs_HAtrs(h)=hatrs

17.  genL: LAtrs → L
17.  genL(latrs) as l
17.    pre  card obs_HIs(latrs)=2
17.    post obs_LI(l)=obs_LI(latrs)
17.        ∧ obs_LI(l)=obs_LI(latrs)
17.        ∧ obs_HIs(l)=obs_HIs(latrs)

18.  genL: LAtrs → N → L
18.  genL(latrs)(n) as l
18.    pre  card obs_LIs(latrs)=2
18.        ∧ obs_LIs(latrs)⊆xtr_LIs(n)
18.    post l ∉ obs_Ls(n)
18.        ∧ obs_LI(l)=obs_LI(latrs)
18.        ∧ obs_HIs(l)⊆obs_HIs(latrs)
18.        ∧ obs_LAtrs(l)=latrs
```

2.2 States 22

Definition: State. *By a state we shall understand a collection of one or more simple entities.*

2.3 Actions

Definition: Action. *By an action we shall understand something which potentially changes a state[705], that is, a function application to a state which potentially changes that state.*

2.3.1 Insert Hubs 23

19. One can insert a hub, h, into a net, n.

The hub to be inserted

20. must not be a hub of the net and

21. h cannot already be connected to any links.

    That is, we can only insert "isolated" hubs.

The result of inserting a hub, h, into a net, n, is a new net, n',

22. which is like n except that it now also has the hub h.

24

value
```
19.  insertH: HAtrs → N ~→ N
19.  insertH(hatrs)(n) as n′
19.  let h = genH(hatrs)(n) in
20.  pre   h ∉ obs_Hs(n)
21.        ∧ obs_LIs(h) = {}
22.  post obs_Ls(n)=obsLs(n′)
22.       ∧ obs_Hs(n′)=obs_Hs(n)∪{h}
22.       ∧ obs_HAtrs(h)=hatrs
19.       end
```

Theorem: Inserting a proper hub in a well-formed net that is, a net satisfying all relevant axioms, results in a likewise well-formed net.

2.3.2 Remove Hubs 25

23. One can remove a hub, h, from a net, n.

    The hub to be removed

24. must be a hub of the net and

25. h cannot be connected to any links.

    That is, the hub, h, may earlier – in is membership of the net – have been connected to links, but these must already, at the time of hub removal, have been removed, see below.

    That is, we can only remove "isolated" hubs.

26. The result of removing a hub, h, from a net, n, is a new net, n',

27. which is like n

28. except that it now no longer has hub h.

26

value
```
23.  removeH: H → N ~→ N
26.  removeH(h)(n) as n′
24.  pre  h ∈ obs_Hs(n)
25.     ∧ obs_LIs(h) = {}
27.  post obs_Ls(n)=obsLs(n′)
28.     ∧ obs_Hs(n′)=obs_Hs(n)\{h}
```

Please note the almost line-by-line similarity of the insert and remove hub descriptions and that the only difference between these descriptions are the membership, union, respectively set difference operations ($\notin$, $\in$, $\cup$ respectively $\backslash$).

28

2.3.3 Insert Links 27

29. One can insert a link, ℓ, into a net, n.

The link to be inserted must

30. not be a link of the net,

31. but the observable hub identifiers must be those of hubs of the net.

The result of inserting a link, ℓ, into a net,

32. n, is a new net, n',

33. in which ℓ is now a member.
34. Let h_{j_i}, h_{k_i} be the two (distinct) hub identifiers of ℓ and
35. let h_j, h_k be the two (distinct) hubs of n which are identified by h_{j_i}, h_{k_i}.
36. All hubs of net n except h_j, h_k are the same as in n and are unchanged in n'.
37. The two hubs h_j, h_k of n become hubs h'_j, h'_k of n'
38. such that only the observable identifiers of connected links have changed to now also include the identifier of link ℓ,
39. and such that the observed attributes are those of the argument.

29

```
value
 29.  insertL: L × LAtrs → N ~→ N
 32.  insertL(l,latrs)(n) as n′
 30.  pre  l ∉ obs_Ls(n)
 31.     ∧ obs_HIs(l)⊆xtrHIs(n)
 33.  post obs_Ls(n′) = obs_Ls(n) ∪ {l}
 34.      ∧ let {hji,hki}=obs_HIs(l) in
 35.        let (hj,hk) = (getH(hji)(n),getH(hki)(n)) in
 31.        {hj,hk}⊆obs_Hs(n)
 36.      ∧ obs_Hs(n)\{hj,hk} = obs_Hs(n′)\{hj,hk}
 37.      ∧ let (hj′,hk′) = (getH(hji)(n′),getH(hki)(n′)) in
 38.        obs_LIs(hk′) = obs_LIs(hk′) ∪ {obs_LI(l)}
 38.      ∧ obs_LIs(hj′) = obs_LIs(hj′) ∪ {obs_LI(l)} end end end
 39.      ∧ obs_LAtrs(l) = latrs
```

30

```
 xtrHIs: N → HI-set
 xtrHIs(n) ≡ {obs_HI(h)|h:H•h ∈ obs_Hs(n)}

 getH: HI → N ~→ H
 getH(hi)(n) ≡ let h:H • h ∈ obs_Hs(n) ∧ obs_HI(h)=hi in h end
    pre ∃ h:H • h ∈ obs_Hs(n) ∧ obs_HI(h)=hi
```

33

2.3.4 Remove Links 31

40. One can remove a link, ℓ, from a net, n.

The link to be removed must

41. be a link of the net.

32

The result of removing a link, ℓ, from a net,

42. n, is a new net, n',
43. in which ℓ is no longer a member.
44. Let h_{j_i}, h_{k_i} be the two (distinct) hub identifiers of ℓ and
45. let h_j, h_k be the two (distinct) hubs of n which are identified by h_{j_i}, h_{k_i}.
46. h_j, h_k are in n'.
47. All hubs of net n except h_j, h_k are the same as in n and are unchanged in n'.
48. The two hubs h_j, h_k of n become hubs h'_j, h'_k of n'
49. such that only the observable identifiers of connected links have changed to now no longer include the identifier of link ℓ.

```
value
 40.  removeL: L → N ~→ N
 42.  removeL(l)(n) as n′
 41.  pre  l ∈ obs_Ls(n)
 43.  post obs_Ls(n′) = obs_Ls(n) \ {l}
 44.      ∧ let {hji,hki}=obs_HIs(l) in
 45.        let (hj,hk) = (getH(hji)(n),getH(hki)(n)) in
 46.        {hj,hk}⊆obs_Hs(n)
 47.      ∧ obs_Hs(n)\{hj,hk} = obs_Hs(n′)\{hj,hk}
 48.      ∧ let (hj′,hk′) = (getH(hji)(n′),getH(hki)(n′)) in
 49.        obs_LIs(hk′) = obs_LIs(hk′) \ {obs_LI(l)}
 49.      ∧ obs_LIs(hj′) = obs_LIs(hj′) \ {obs_LI(l)} end end end
```

Please note the almost line-by-line similarity of the insert and remove link descriptions and that the only difference between these descriptions are the union, respectively set difference operations (∪ respectively \).

2.3.5 Two Theorems 34

Idempotency With the preconditions satisfied by the insert and remove actions one can prove that first inserting a hub (link) into a net and then removing that hub (link) from the resulting net restores the original net:

```
theorem
  ∀ n,n′:N,h:H,l:L •
      pre insertH(h)(n) ∧ removeH(h)(n′) ∧ insertL(l)(n) ∧ removeL(l)(n′) ⇒
      removeH(h)(insertH(h)(n)) = n ∧ removeL(l)(insertL(l)(n))
```

Reachability 35 Any net that satisfies the axioms above can be constructed by sequences of insert hub and link actions.

```
theorem
  let n_nil:N • obs_Hs(n_nil)=obs_Ls(n_nil)={} in
  ∀ n:N ⊢ axioms 7. and 8 on page 14.; 9 on page 14. 10 on page 14. •
     ∃ hl:H*, ll:L* • let n′ = insertHs(hl)(n_nil) in insertHs(ll)(n′)=n end
  end

  insertHs: H* → N ~→ N
  insertLs: L* → N ~→ N

  insertHs(hl)(n) ≡ case hl of ⟨⟩ → n, ⟨h⟩^hl′ → insertHs(hl′)(insertH(h)(n)) end
  insertLs(ll)(n) ≡ case ll of ⟨⟩ → n, ⟨l⟩^ll′ → insertLs(ll′)(insertL(l)(n)) end
```

Informal proof: An informal proof goes like this: Take a net. For every hub, h, in that net, let h' be a version of h which has the same hub identifier, an empty set of observable link identifiers (of connected links), and otherwise all other attributes of h, let h' be a member of the list of hubs – and only such hubs. Let every and only such links in n be members of the list of links. Performing first the insertion of all hubs and then the insertions of all links will "turn the trick" ! **end of informal proof.** 36

2.4 Events 37

Definition: Event. *An event is something that occurs instantaneously. Events are manifested by certain* state[705] *changes, and by certain* interaction[392]*s between* behaviour[79]*s or* process[544]*es. The occurrence of events may "trigger" [further] actions. How the triggering, i.e., the* invocation[402] *of* functions *are brought about is usually left implied, or unspecified.* 38

A mudslide across a railway track or a road segment (i.e., a link) represents an event that effectively "removes" the link, or at least a segment of a link. Similarly if a train and/or automobile bridge collapses or a tunnel gets flooded or catches fire.

How are we to model such, and other events? 39

50. We choose to model the event" *"disappearance" of a segment of a link* identified by l_i:LI as the composition of the following actions:
    a) the removal of link l:L being affected, where l_i:LI identifies the link in the network;
    b) the insertion of two hubs, h',h'':H, corresponding to "points" (on link l:L) on either side of the mudslide or bridge – or other; and
    c) the insertion of two links, l',l'':L, between the hubs of the original link and the new hubs.
    d) l_i:LI must identify a link l:L of net n:N.

```
50b. newH: N → H-set → H
50b. newH(n)(hs) ≡ let h:H • h ∉ hs ∧ obs_LIs(h)={} in h end
50c. newL: N → L-set → (HI×HI) → L
50c. newL(n)(ls)(hi′,hi″) ≡ let l:L • l ∉ ls ∧ obs_HIs(l)={hi′,hi″} in l end
```

40

```
value
  50.  event_link_disappearance: LI → N ~→ N
  50a.    let l = xtrL(li)(n) in
  50a.    let {hi′,hi″} = obs_HIs(l) in
  50a.    let n′ = removeL(l)(n) in
  50b.    let h′= newH(n)(obs_Hs(n)) in
  50b.    let h″ = newH(n)(obs_Hs(n)∪{h′}) in
  50b.    let n″ = insertH(h′)(insertH(h″)(n)) in
  50c.    let l′ = newL(n)(obs_Ls(n))(obs_HI(h′),hi′) in
  50c.    let l″ = newL(n)(obs_Ls(n)∪{l′})(obs_HI(h″),hi″) in
  50c.    insertL(l′)(insertL(l″)(n″)) end end end end end end end end
  50d.  pre li ∈ xtrLIs(n)
```

The newH and newL generator (or constructor) functions are simplified versions of more realistic such functions. Hubs and links, as we shall see, have attributes beyond those obs_HI, obs_LI, obs_LIs and obs_HIs. Proper newH and newL generator definitions must express that initial values be ascribed to these other attributes. Examples of further hub and link attributes are: spatial location, name[4], mode[5], length for links, etcetera. So, eventually, the definitions of the newH and newL constructors will have to be redefined.

There will be very many other kinds of events in connection with transportation.

MORE TO COME

[4]Names of hubs and links must not be confused with hub and link identifiers: Two or more hubs and/or links may have the same name. Hub and link identifiers may be thought of as abstractions of some composition of locations and names in that no two hubs and/or links can "occupy" "overlapping" locations, that is, locations are unique.

[5]whether road, railway, shipping or air traffic hubs and links, or, even combinations of these

2.5 Behaviours 41

Definition: Behaviour. *By behaviour we shall understand the way in which something functions or operates. In the context of domain engineering[248] behaviour is a concept associated with phenomena, in particular manifest entities[272]. And then behaviour is that which can be observed about the value[802] of the entity[272] and its interaction[392] with an environment[275]. A simple, sequential behaviour is a sequence of zero, one or more actions and events.*

2.5.1 Behaviour Prescriptions 42

Usually behaviours follow a prescription.

In the case of net construction we refer to the prescription as a construction plan.

Construction Plans

51. The plan for constructing a net can be abstracted as 46
   a) a map, PLAN, which to each hub identifier associates
   b) a link-to-hub identifier map, LHIM, from the identifiers of links emanating from the hub to identifiers of connected hubs.

type
51a. PLAN = HI $\overrightarrow{m}$ LHIM
51b. LHIM = LI $\overrightarrow{m}$ HI

The hub identifiers of the definition set of construction plans are called the defining occurrences of hub identifiers.

The hub identifiers of the ranges of link-to-hub identifier map are called the using occurrences of hub identifiers.

Wellformedness of Construction Plans 43

52. Wellformed net construction plans satisfy three conditions:
   a) *All Links are Two-way Links:*
      i. Let h_k be any hub identifier of the construction plan.
      ii. For all link identifiers, l_j, of the LIHM, $lhim_k$, mapped into by h_k,
      iii. let h_ℓ be the hub identifier mapped into by l_j in $lhim_k$,
      iv. then l_j is in the link-to-hub-identifier map, $lhim_\ell$, mapped into by h_ℓ, 44
   b) *Using Hub Identifier Occurrences are Defined:*
      i. Let $lhim$ be any link-to-hub-identifier map of a construction plan.

      ii. For every hub identifier, h_i, mapped to by a link identifier, l_j, in $lhim$
      iii. there exists a hub identifier, h_k, that maps into l_j; and 45
   c) *No Junk:* To secure consistency between hub and link identifiers of a construction plan we impose: all the defined hub identifiers of a construction plan are in the range of some link to hub identifier map of that plan; and each of the hub identifiers of some link to hub identifier map are defined in the construction plan are in the range of some link to hub identifier map of that plan.

value
52. wf_PLAN: PLAN → **Bool**
52. wf_PLAN(plan) ≡
52a. all_links_are_two_way_links(plan) ∧
52b. hub_identifier_occurrences_are_defined(plan) ∧
52c. no_junk(plan)

```
52a.      all_links_are_two_way_links: PLAN → Bool
52a.      all_links_are_two_way_links(plan) ≡
52(a)i.     ∀ hk:HI • hk ∈ dom plan ⇒
52(a)ii.      ∀ lj:LI • lj ∈ dom plan(hk) ⇒
52(a)iii.       let hl = (plan(hk))(lj) in
52(a)iv.        lj ∈ dom plan(hl) end
```

```
52b.      hub_identifier_occurrences_are_defined: PLAN → Bool
52b.      hub_identifier_occurrences_are_defined(plan) ≡
52(b)i.     ∀ hlim:HLIM•hlim ∈ rng plan
52(b)ii.      ∀ lj:LI • lj ∈ dom lhim ⇒
52(b)iii.       ∃ hk:HI • hk ∈ dom plan ∧ lj ∈ dom plan(hk)
```

```
52c.  no_junk: PLAN → Bool
52c.  no_junk(plan) ≡ dom plan = ∪{rng(plan(hi))|hi:HI•hi ∈ dom plan}
```

2.5.2 Augmented Construction Plans 47

Hubs and links in nets possess attributes (cf. Item 4 on page 13.). Some attributes have already been dealt with: the identifiers of hubs and links that can be observed from hubs, respectively links (cf. Items 4. and 5 on page 13.) and the identifiers of hubs that can be observed from links and the identifiers of links that can be observed from hubs (cf. Items 9. and 10 on page 14.).

In addition hubs and links in nets possess further attributes:

- spatial location of hubs and links,
- (locally ascribed) names of hubs and links,
- lengths of links,
- etcetera.

48

We therefore augment construction plans to also reveal these attributes.

```
type
    APLAN = PLAN × HInfo × LInfo
    HInfo = HI →m HAtrs
    LInfo = LI →m LAtrs
```

49

53. The wellformedness of an augmented plan secures that
    a) all hubs identifiers defined in the construction plan are "detailed" in the hub information component, and that
    b) all links identifiers used in the construction plan are "detailed" in the in the link information component.

```
value
53.   wf_APLAN: APLAN → Bool
53.   wf_APLAN(plan,hinfo,linfo) ≡
53a.      dom plan = dom hinfo ∧
53b.      ∪{dom lhim|lhim:LHIM•lhim ∈ rang plan}=dom linfo
```

2.5.3 Sequential Construction Behaviours 50

54. From an augmented construction plan one can "extract" initial information about
    a) all hubs and
    b) all links.

```
value
54a. xtrH: HI → APLAN → HI × HAtrs, xtrH(hi)(_,hinfo,_) ≡ hinfo(hi)
54b. xtrL: LI → APLAN → LAtrs, xtrL(li)(_,_,linfo) ≡ linfo(li)
```

51

55. A net construction behaviour can be (functionally and non-deterministically) modelled as
    a) a sequence of hub insertions followed by

    b) a sequence of link insertions.

```
value
55.  net_construction: HInfo×LInfo → (HI-set×LI-set) → N → N
55.  net_construction(hinfo,linfo)(his,lis)(n) ≡
55.     case (his,lis) of
55a.        ({hi}∪ his′,_) →
55a.            net_construction(hinfo,linfo)(his′,lis)(insertH(hinfo(hi))(n)),
55b.        ({},{li}∪ lis′) →
55b.            net_construction(hinfo,linfo)({},lis′)(insertL(linfo(li))(n)),
55.         ({},{}) → n
55.     end
```

52

The net_construction function is initialised with the full sets of hub and link identifiers and with an empty net:

```
    net_construction(hinfo,linfo)(dom hinfo,dom linfo)(n_nil)
value
    n_nil:N • obs_Hs(n_nil) = {} = obs_Ls(n_nil)
```

The net_construction behaviour shown above defines only a subset of all the valid behaviours that will construct a net according to the augmented plan (plan,hinfo,linfo). Other valid behaviours would start with constructing at least two hubs but could then go onto construct some of the (zero, one or more) links that connect some of the already constructed hubs, etcetera. We challenge the reader to precise narrate and formally define such net_construction behaviours.

3 An Ontology of Domain Facets 53

3.0.4 Definitions

Definition: Domain. *An area of activity which some software[685] is to support (or supports) or partially or fully automate (resp. automates).*
The term 'application domain' is considered synonymous with the term 'domain'.
Definition: Domain Description. *A textual, informal or formal document which describes a domain **as it is**.* 54
Usually a domain description is a set of documents with many parts recording many facets of the domain: The *business process*[99]es, *intrinsics*[399], *support technology*[725], *rules and regulations*[640], *management and organisation*[445], and the *human behaviour*[345]s. 55
Definition: Domain Engineering. *The engineering of the development of a domain description[243], from identification of domain[239] stakeholder[703]s, via domain acquisition[240], domain analysis[241], terminologisation, and domain description[243] to domain validation[256] and domain verification[257].* 56
Definition: Domain Facet. *By a domain facet we understand one amongst a finite set of generic ways of analysing a domain: A view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain.*
We consider here the following domain facets: *business process*[99]es, *intrinsics*[399], *support technology*[725], *rules and regulations*[640], *management and organisation*[445], and *human behaviour*[345].

3.0.5 What Can Be Observed 57

- "Whether you can observe a thing or not depends on the theory which you use. It is the theory which decides what can be observed."
- Albert Einstein objecting to the placing of observables at the heart of the new quantum mechanics, during Heisenberg's 1926 lecture at Berlin; related by Heisenberg, quoted in *Unification of Fundamental Forces* (1990) by Abdus Salam ISBN 0521371406.

3.0.6 Business Processes 58

A Characterisation By a business process we shall understand a *behaviour*[79] of an enterprise, a business, an institution, a factory.

An Example The business processes of transportation evolves around freights or passengers being transported along routes by a vehicle (car, train, aircraft, ship) "propelled" by some locomotive force.

3.1 Intrinsics 59

Definition: Intrinsics. *By the intrinsics of a domain[239] we shall understand those phenomena and concepts of a domain which are basic to any of the other facets, with such a domain intrinsics initially covering at least one stakeholder[703] view.*

3.1.1 Net Topology Descriptors 60

Instead of dealing with the entire phenomenon of a net, that is, the real, physical, geographic "thing", we can describe essentials of a net, for example how its hub and links are connected. 61

56. One way of abstractly modelling a net descriptor is as a map, nd, from hub identifiers to simple maps, lihis, from link identifiers to hub identifiers,
57. such that
    a) for all hi in (the definition set of) nd it is the case that
    b) if hi maps to lihi,
    c) and in that link identifier to hub identifier map, li maps to hi′,
    d) then hi′ is different from hi and
    e) hi′ maps to an lihi′ in which li is defined and maps to hi.
    f) And there are only such pairings.

```
type
56.  ND′ = HI →m (LI →m HI)
56.  ND = {|nd′:ND•wf_ND(nd′)|}
value
57.  wf_ND: ND′ → Bool
57.  wf_ND(nd) ≡
57a.     ∀ hi:HI•hi ∈ dom nd ⇒
57b.        let lihi = nd(hi) in
57c.        ∀ li:LI • li ∈ dom lihi ⇒
57c.           let hi′ = (nd(hi))(li) in
57d.           hi ≠ hi′ ∧
57e.           hi′ ∈ dom nd ∧ li ∈ dom(nd(hi′)) ∧ hi=(nd(hi′))(li)
57f.        end end
```

From a net one can construct its net descriptor: 62

```
value
   conND: N → ND
   conND(n) ≡
      [ hi↦[ li↦hi′|li:LI,hi′:HI•li ∈ obs_LIs(getH(hi,n))∧{hi,hi′}=obs_HIs(getL(li,n)) ]|
         hi:HI•hi ∈ xtrHIs(n) ]
```

3.1.2 Link States and Link State Spaces 63

Links are (one of the) means of transport[6]. Hubs allow movement along one (hub-connected) link to be diverted onto another (hub-connected) link.

We introduce the notions of the state of a link, the state of a hub, the state space of a link and the state space of a hub. States abstract directions of movement.

Links are, by our previous definitions, bi-directional: from one of the connected hubs to the other, and vice versa. And hubs are multi-directional: from potentially any link via the hub to potentially any link. 64

Let the observed hub identifiers of a link ℓ be $\{h_j, h_k\}$, then link ℓ can potentially be in any one of the four link states: $\{\{(h_j, h_k), (h_k, h_j)\}, \{(h_j, h_k)\}, \{(h_k, h_j)\}$ and $\{\{\}\}\}$. Any one particular link may always remain in one and the same state, or it may from time to time undergo transitions between any subset of the potential link state space. 65

58. Link states, $l\sigma{:}L\Sigma$, are set of pairs of hub identifiers.
59. Link state spaces are set of link states.
60. From a link one can generate the link state space of all potential link states.
61. From a link one can observe the current link state $l\sigma{:}L\Sigma$.
62. From a link one can observe the link state space $l\omega{:}L\Omega$.

66

```
type
58.   LΣ = (HI×HI)-set
59.   LΩ = LΣ-set
value
60.   generate_full_LΣ: L → LΣ
60.   generate_full_LΣ(l) ≡
60.      {}∪{(hi′,hi″)|hi′,hi″:HI•hi′≠hi″∧{hi′,hi″}=obs_HIs(l)}

60.   generate_LΩ: L → LΩ
60.      let fullLσ = generate_full_LΣ(l) in
```

[6]Other means are vehicles moving along links and crossing hubs and the locomotive force that drives the vehicles. Freight, including people, are what is being transported.

```
60.      {{},∪{σ|σ:LΣ•σ⊆fullLσ}} end

61.   obs_LΣ: L → LΣ
62.   obs_LΩ: L → LΣ-set
```

3.1.3 Hub States and Hub State Spaces 67

63. Hub states, $h\sigma{:}H\Sigma$, are sets of pairs of link identifiers ((l_i, l_k)), designating that if (l_i, l_k) is in the current hub state then movement can take place from the link designated by l_i (via hub h) to the link designated by l_k.
64. Hub state spaces are set of hub states.
65. From a hub one can generate the hub state space of all potential hub states.
66. From a hub one can observe the current hub state $h\sigma{:}H\Sigma$.
67. From a hub one can observe the hub state space $h\omega{:}H\Omega$.

68

```
type
63.   HΣ = (LI×LI)-set
64.   HΩ = HΣ-set
value
65.   generate_full_HΣ: H → HΣ
65.   generate_full_HΣ(h) ≡
65.      {}∪{(li′,li″)|li′,li″:LI•{li′,li″}⊆obs_LIs(h)}

60.   generate_HΩ: H → HΩ
60.      let fullHσ = generate_full_HΣ(h) in
60.      {{}∪{σ|σ:HΣ•σ⊆fullHσ}} end

66.   obs_HΣ: H → HΣ
66.   obs_HΩ: H → HΣ-set
```

3.1.4 State and State Space Wellformedness 69

68. States must be in appropriate state spaces.
69. State spaces must be subsets of all potential appropriate states.

axiom
  ∀ n:N,l:L,h:H • l ∈ obs_Ls(n) ∧ h ∈ obs_Hs(n) ⇒
58. obs_LΣ(l) ∈ obs_LΩ(l) ∧
59. obs_LΩ(l) ⊆ generate_full_LΣ(l) ∧
58. obs_HΣ(h) ∈ obs_HΩ(h) ∧
59. obs_HΩ(h) ⊆ generate_full_HΣ(h)

theorems:
  ∀ n:N,l:L,h:H • l ∈ obs_Ls(n) ∧ h ∈ obs_Hs(n) ⇒
    obs_LΣ(l) ⊆ {(hi′,hi″)|hi′,hi″:H•{hi′,hi″}⊆obs_HIs(l)} ∧
    obs_HΣ(h) ⊆ {(li′,li″)|li′,li″:L•{li′,li″}⊆obs_LIs(h)}

3.1.5 Concrete Types for Simple Entities 70

As an alternative for, or as a step of refinement from the earlier sorts of nets, hubs and links one can simplify matters by concrete types for these simple entities.

70. Nets are Cartesians of sets of hubs and links.
71. A link is a Cartesian of a link identifier, a set of exactly two hub identifiers, a link state, a link state space, and a number of presently further unspecified link attributes.
72. A hub is a Cartesian of a hub identifier, a set of zero, one or more link identifiers, a hub state, a hub state space, and a number of presently further unspecified hub attributes.

71

type
70. N = H-**set** × L-**set**
71. L :: obs_LI:LI × obs_HIs:HI-**set** × LΣ × LΩ × LAtrs
72. H :: obs_HI:HI × obs_LIs:LI-**set** × HΣ × HΩ × HAtrs

72

We leave it to the reader to narrate the wellformedness constraints.

axiom
```
  ∀ (hs,ls):N • ls≠{} ⇒ card hs ≥ 2 ∧
    ∀ l′,l″:L • {l′,l″}⊆ls ∧ l′≠l″ ⇒ obs_LI(l′)≠obs_LI(l″) ∧
    ∀ h′,h″:H • {h′,h″}⊆hs ∧ h′≠h″ ⇒ obs_HI(h′)≠obs_HI(h″) ∧
    ∀ l:(li,his,lσ,lω,latrs):L • l ∈ ls ⇒
      card his=2 ∧ his⊆{obs_HI(h″)|h‴:H • h‴ ∈ hs} ∧
      lσ ∈ generate_full_LΣ(l) ∧
      lσ ∈ lω ⊆ generate_full_LΣ(l) ∧
    ∀ h:(hi,lis,hσ,hω,hatrs):H • h ∈ hs ⇒
      lis⊆{obs_LI(l‴)|l‴:L • l‴ ∈ ls} ∧
      hσ ∈ generate_full_HΣ(h) ∧
      hσ ∈ hω ⊆ generate_full_HΣ(h)
```

74

3.1.6 Example Hub Crossings 73

Figure 1 shows four hub/partial link corner diagrams (1.–4.). These are intended to show four distinct hub states. Let the center diagram (5.) of Fig. 1 indicate the link identifiers of the four partial links of each of the four hub/partial link diagrams.

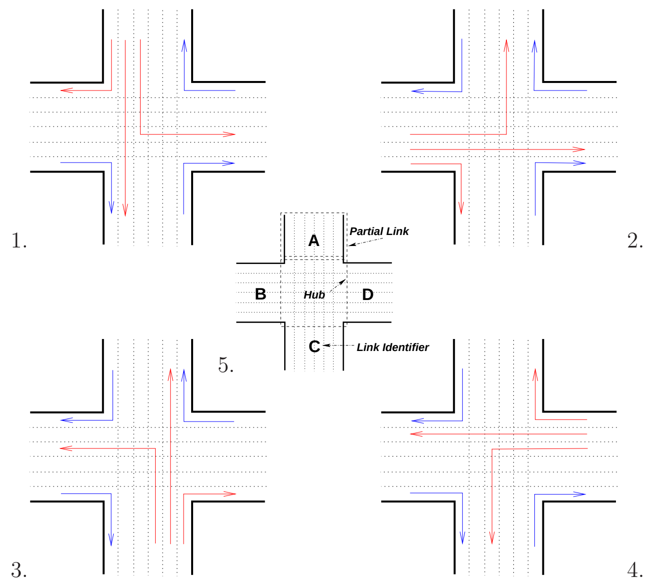

Figure 1: Four “Safe” Flows

The top left hub/link diagram (1.) thus can be claimed to depict hub state $\{(A,B), (A,C), (A,D), (B,C), (C,D), (D,A)\}$.

Photo 2 on the facing page shows a semaphore which seems to be able to display all kinds of states.

The point of this example is to show that a hub may take on many states, that not all hub states may be desirable (viz., lead to crossing traffic if so interpreted), and that to reach from one hub state to another one must change the state.

3.1.7 Actions Continued 75

73. The action change_HΣ takes a hub, h, in some state, and a desired next state, $h\sigma'$, and results in a hub, h', which
    a) has the same hub identifier as h,

Figure 2: A General Purpose Traffic Light

79

is connected to the same links as h,

has the same hub state space as h,

has the same attributes (names and values) as h,

b) but whose state may have changed.

73b. The new state of h' ought be $h\sigma'$, but electro-mechanical or other failures in setting the state may set the new state to any state of the potential states of h (i.e., h'), not just to any state in the hub state space of h.

76

80

```
value
73.  change_HΣ: H × HΣ → H
73.  change_HΣ((hi,lis,hσ,hω,hatrs),hσ′) ≡
73b.     let hσ′′′ ∈ generate_full_HΣs in
73a.     (hi,lis,hσ′′′,hω,hatrs) end
```

Had we specified that the resulting state must be $h\sigma'$ then we had prescribed a requirements to a change operation. As it is now we have described a domain phenomenon, namely that operations may fail.

3.2 Support Technologies 77

Definition: Support Technology. *By a support technology we understand a facet[285] of a domain[239], one which reflects its (current) dependency on human, mechanical, electro-mechanical, electronic and/or other technologies (i.e., tools) in order to carry out its business process[99]es.*

3.2.1 Traffic Signals 78

A traffic signal represents a technology in support of visualising hub states and in effecting state changes.

74. A hub state is now modelled as a triple: the link identifier l_i ("coming from"), a colour (**red**, **yellow**, and **green**), and another the link identifier l_j ("going to").

75. Signalling is now a sequence of one or more pairs of next hub states and time intervals:

$$< (h\sigma_1, ti_1), (h\sigma_2, ti_2), ..., (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) >, n > 0$$

The idea of a signalling is to first change the designated hub to state $h\sigma_1$, then wait ti_1 time units, then set the designated hub to state $h\sigma_2$, then wait ti_2 time units, etcetera, ending with final state σ_n and a (supposedly) long time interval ti_n before any decisions are to be made as to another signalling.

The set of hub states $\{h\sigma_1, h\sigma_2, ..., h\sigma_{n-1}\}$ of

$$< (h\sigma_1, ti_1), (h\sigma_2, ti_2), ..., (h\sigma_{n-1}, ti_{n-1}), (h\sigma_n, ti_n) >, n > 0$$

are called intermediate states.

Their purpose is to secure an orderly phase out of green via yellow to red and phase in of red via yellow to green in some order for the various directions.

We leave it to the reader to devise proper wellformedness conditions for signaling sequences as they depend on the hub topology.

76. A street signal (a semaphore) is now abstracted as a map from pairs of hub states to signalling sequences.

The idea is that given a hub one can observe its semaphore, and given the state, $h\sigma$ (not in the above set), of the hub "to be signalled" and the state $h\sigma_n$ into which that hub is to be signalled "one looks up" under that pair in the semaphore and obtains the desired signalling.

```
type
74.  HΣ = LI × Colour × LI
74.  Colour == red | yellow | green
```

```
75. Signalling = (HΣ × TI)*
75. TI
76. Sempahore = (HΣ×HΣ) →m Signalling
value
76. obs_Semaphore: H → Sempahore
```

81

77. A hub semaphore, sema, contains only such hub states as are observed in the hub state space.
    a) Let hsps be the set of "from/to" hub state pairs in semaphore sema.
    b) Then hs is the set of all hub states mentioned in hsps.
    c) To hs join all the hub states mentioned in any signalling, sg, of sema.

```
77. hub_state_space: Sempahore → HΣ-set
77. hub_state_space(sema) ≡
77a.    let hsps={hsp|hsp:(HΣ×HΣ)•hsp ∈ dom sema} in
77b.    let hs={hσ′,hσ″|hσ′,hσ″:HΣ•(hσ′,hσ″)∈ hsps} in
77c.    hs ∪ ∪{{hσ|(hσ,ti):(HΣ×TI)•(hσ,ti)∈ elems sg}|sg:Signalling•sg ∈ rng sema}
77.     end end
axiom
77. ∀ h:H • ∪ obs_HΩ(h) = hub_state_space(obs_Semaphore(h))
```

3.2.2 Traffic "Control" 82

78. Given two hub states, $h\sigma_{\text{init}}$ and $h\sigma_{\text{end}}$, where $h\sigma_{\text{init}}$ designates a present hub state and $h\sigma_{\text{end}}$ designates a desired next hub state after signalling.

79. Now signalling is a sequence of one or more successful hub state changes.

```
value
78. signalling: HΣ × HΣ → H → H
79. signalling(hσ_init,hσ_end)(h) ≡
79.     let sema = obs_Semaphore(h) in
79.     let sg = sema(hσ_init,hσ_end) in
79.     signal_sequence(sg)(h) end end
79. pre (hσ_init,hσ_end) ∈ dom obs_Semaphore(h)
```

83

```
79. signal_sequence(⟨⟩)(h) ≡ h
79. signal_sequence(⟨(hσ,ti)⟩^sg)(h) ≡
79.     let hσ′ = change_HΣ(h)(hσ) in
79.     if hσ′ ≠ hσ then chaos
79.     else wait(ti); signal_sequence(sg)(h) end end
```

If a desired hub state change fails (**chaos**) then we do not define the outcome of signalling.

3.3 Rules and Regulations 84

85

Definition: Rule. *A rule stipulates a regulating principle. In the context of modelling domain rules we shall understand a domain rule as some text whose meaning is a predicate[536] over a pair of suitably chosen domain state[705]s. We may assume that a domain action[12] or a domain event[281] takes place in the first of these states and results in the second of these states. If the predicate is true then we say that the rule has been obeyed, otherwise that it has been violated.*
Usually a domain rule is paired with a possibly remedying regulation.
Definition: Regulation. *A regulation stipulates that an action[12] be taken in order to remedy a previous action which violated a rule[638]. That is, a regulation is some text which designates a possibly composite action[12], that is, a state-to-state change which ostensibly results in a state in which the rule, "attached" to the regulation, now holds.*

3.3.1 Vehicles 86

In preparation for examples of transportation rules and regulations we introduce vehicles.

80. Vehicles are further undefined quantities except that
    a) vehicles have unique identifiers,
    b) vehicles are either positioned
        i. at/in hubs
        ii. or on links, in some fractional (non-zero) distance from a hub toward the connecting hub.

81. From a net (sort) one can observe all the vehicles of the net.[7]

82. No two vehicles so observed have the same identifier.

87

```
type
80.        V
80a.       VI
80b.       VP   = HP | LP
80(b)i.    HP == atH(hi:HI)
80(b)ii.   LP == onL(li:LI,fhi:HI,f:F,thi:HI)
80(b)ii.   F = {|f:F•0<f<1|}
value
80a.       obs_VI: V → VI
80b.       obs_VP: V → VP
81.        obs_Vs: N → V-set
axiom
```

[7]Thus a concrete net type, in addition to hubs and links (now) also contains vehicles.

```
82.   ∀ v:V • v ∈ obs_Vs(n) ⇒
82.      ∃ onL(li,fhi,f,thi):VP • onL(li,fhi,f,thi)=obs_VP(v) ⇒
82.         ∃ l:L•l ∈ obs_Ls(n)∧li=obs_LI(l)∧{fhi,thi}=obs_HIs(l) ∨
82.      ∃ atH(hi):VP • atH(hi)=obs_VP(v) ⇒
82.         ∃ h:H•h ∈ obs_Hs(n)∧hi=obs_HI(h)
```

MORE TO COME

3.3.2 Traffic 88

83. By traffic we understand a continuous function from time to a pair of nets and position of vehicles.

84. By time we understand a dense set of points with dense and points being mathematical concepts [57, 221].

```
type
83.  TF = T → (sel_net:N × sel_veh_pos:(V ⇀ VP))
                                        m
84.  T
```

Wellformedness of Traffic 91

Expressing the wellformedness of traffic is not a simple matter. We shall approach this task in a number of "small steps".

89

- **Static Wellformedness**

85. We define a predicate over vehicle positions.

    a) Every vehicle in the traffic has a proper position on the net, either at a hub or along a link.

    b) No two vehicles of the traffic can occupy exactly the same link position. (That is, the link positions onL(li,hi,f,hi′) and onL(li,hi,f′,hi′) must have the two fractions (f, f') differ – be it ever so "minutely").

We first define two auxiliary functions:[8]

92

```
value
   obs_HIs: N → HI-set
   obs_HIs(n) ≡ {obs_HI(h)|h:H•h ∈ obs_Hs(n)}
   obs_LIs: N → LI-set
   obs_LIs(n) ≡ {obs_LI(h)|l:L•l ∈ obs_Ls(n)}
```

90

```
85.  proper_vehicle_positions: TF → Bool
85.  proper_vehicle_positions(tf) ≡
85.     ∀ t:T • t ∈ DOMAIN tf •
85.       let (n,vps) = tf(t) in
85a.        ∀ v:V•v ∈ dom vp•is_net_position(vps(v))(n)
85b.        ∀ v′:V•v′ ∈ dom vp ∧ v≠v′⇒diff_net_pos(vps(v),vps(v′))
85.       end
85a. is_net_position: VP → N → Bool
85a. is_net_position(vp)(n) ≡
85a.    case vp of
85a.       atH(hi) → hi ∈ obs_HIs(n),
85a.       onL(li,fhi,f,thi) → li ∈ obs_LIs(n)∧{fhi,thi}⊆obs_HIs(n)
85a.    end
85b. diff_net_pos: VP × VP → Bool
85b. diff_net_pos(vp,vp′) ≡
85b.    case (vp,vp′) of
85b.       (atH(hi),atH(hi)) → true,
85b.       (onL(li,fhi,f,thi),onL(li,fhi,f′,thi)) → f≠f′,
85b.       _ → true
85b.    end
```

- **Dynamic Wellformedness**

86. Vehicles, when moving, move monotonically, that is,

    a) if a vehicle, at some time, t, is at a link position onL(li,hi,f,hi′) where f is not infinitesimally close to 1, then that vehicle will, at some later time t', infinitesimally close to t, be at link position onL(li,hi,f′,hi′) where f' is infinitesimally close to f;

    b) if the vehicle, at some time, t, is at a link position onL(li,hi,f,hi′) where f is indeed infinitesimally close to 1, then that vehicle will, at some infinitesimally later time t', be at hub position atH(hi′);

    c) and if the vehicle, at some time, t, is at a hub position atHP(hi) then the vehicle will at some infinitesimally later time t' either be at hub position atHP(hi) or at some link position onL(li,hi,f,hi′) where f is infinitesimally close to 0.

```
value
86.  monotonic: TF → Bool
```

[8] They really ought to have been defined much earlier!

```
86.   monotonic(tf) ≡
86.     ∀ t,t′:T • {t,t′}⊆DOMAIN tf •
86.       let (n,vps) = tf(t),(n′,vps′)=tf(t′) in
86.       INFINITESIMALLY CLOSE (t,t′)∧t<t′⇒
86.         ∀ v:V•v ∈ dom vps ∩ dom vps′ •
86.           case (vps(v),vps′(v)) of
86a.              (onL(li,fhi,f,thi),onL(li,fhi,f′,thi)) →
86a.                  f<f′ ∧ INFINITESIMALLY CLOSE (f,f′),
86b.              (onL(li,fhi,f,thi),atH(thi)) →
86b.                  INFINITESIMALLY CLOSE (f,1),
86c.              (atH(hi),atH(hi)) → true,
86c.              (atH(hi),onL(li,hi,f,thi)) →
86c.                  INFINITESIMALLY CLOSE (0,f),
86.               _ → true
86.       end   end
```

93

87. If a vehicle is (has been) moving along a link l_i and is now,

- at time t, at position $\mathsf{onL}(l_i, h_j, f, h_k)$, that is, moving from h_j to h_k,
- then it cannot at a subsequent, infinitesimally close time, t', be at a position
- $\mathsf{onL}(l_i, h_k, f', h_j)$, that is, moving in the opposite direction, h_k to h_j.

94

value

```
87.   God_does_not_play_dice[9]: TF → Bool
87.   God_does_not_play_dice(tf) ≡
87.     ∀ t,t′:T • {t,t′}⊆DOMAIN tf ∧ t<t′ ∧ INFINITESIMALLY CLOSE (t,t′)⇒
87.       let (n,vps) = tf(t),(n′,vps′)=tf(t′) in
87.       ∀ v:V • v ∈ dom vps ∩ dom vps′ ⇒
87.         case (vps(v),vps′(v)) of
87.           (onL(li,fhi,_,thi),onL(li,thi,_,fhi))→false,
87.           _ → true
87.       end end
```

95

88. If a vehicle is (has been) moving along and has,

- at time t, been at some position p, and
- at time t', later than t, is at some position p',

[9] Albert Einstein: "I, at any rate, am convinced that He does not throw dice." Letter to Max Born (4 December 1926); The Born-Einstein Letters (translated by Irene Born) (Walker and Company, New York, 1971) ISBN 0-8027-0326-7. Reflects Einstein's view of Quantum Mechanics at the time.

- then it must at all times t'' between t and t' have been somewhere on the net.

value

```
88.   no_ghost_vehicles: TF → Bool
88.   no_ghost_vehicles(tf) ≡
88.     ∀ t,t′:T • {t,t′}⊆DOMAIN tf ∧ t<t′ ⇒
88.       let (n,vps) = tf(t),(n′,vps′)=tf(t′) in
88.       ∀ v:V•v ∈ dom vps ∩ dom vps′ ⇒
88.         ∀ t″:T • t<t″<t′ ⇒
88.           let (n″,vps″) = tf(t″) in v ∈ dom vps″ end
88.       end
```

3.3.3 Traffic Rules (I of II) 96

89. A vehicle must not move from a hub, h_i, into a link ℓ (from hub (identified by) h_i to hub (identified by) h_j) which is closed in direction (h_i, h_j), that is, where (h_i, h_j) is not in the current state of link.

rule:

```
89.  ∀ tf:TF,t:T • t ∈ DOMAIN(tf) ⇒
89.    let (n,tp) = tf(t) in
89.    ∀ v:V • v ∈ dom tp ⇒
89.      case tp(v) of
89.        atH(hi) →
89.          let t′:T • t′>t ∧ t′ ∈ DOMAIN(tr′) ∧ INFINITESIMALLY_CLOSE(t,t′) in
89.          let (n′,tp′) = tf(t′) in
89.          ∃ li:LI,hi′:HI,f:F,hi″:HI •
89.            hi′=hi ∧ INFINITIEIMALLY_CLOSE(f,0) ∧
89.            tp′(v) = onL(li,hi′,f′,hi″) ∧(hi,hi″) ∉ obs_LΣ(getL(li,n′))
89.        _ → ...
89.    end end end end
```

We shall give another rule after the next section.

3.3.4 Another Traffic Regulator 97

We present an abstraction of a more conventional traffic signal than modelled in Items 74 on page 34 to 77 on page 35.

90. A traffic signal now simply shows an entry permit: either **red**, **yellow** or **green** at the hub when "leaving" any link, i.e., at the entry to a hub from any link.

```
type
90. EP == red | yellow | green
90. HΣ = LI →m EP
axiom
90. ∀ h:H • obs_LIs(h)=dom obs_HΣ(h)
```

We leave it to the reader to express a constraint over hub state spaces as to how there must be hub states such that entry from any link is possible.

3.3.5 Traffic Rules (II of II) 98

91. Vehicles must not enter a hub if entry permission is not **green**.

```
rule:
91.  ∀ tf:TF,t:T : t ∈ DOMAIN(tf) ⇒
91.    let (n,vps) = tf(t) in
91.    ∀ v:V • v ∈ dom vps ⇒
91.      case vps(v) of
91.        onL(li,hi,f,hi′) →
91.          INFINITESIMALLY_CLOSE(f,1) ∧
91.          let hσ = obs_HΣ(getH(hi′,n)),
91.              t′:T • t′>t ∧ INFINITESIMALLY_CLOSE(t,t′) in
91.          let (n′,vps′) = vps(t′) in
91.          hσ(li) ≠ green ∧ vps′(v) ≠ atH(hi′)  assert: vps′(v) = onL(li,hi,f,hi′)
91.          end end
91.        _ → ...
91.    end end
```

3.4 Scripts 99

Definition: Scripts. *A script is plan of action. By a domain script we shall, more specifically, understand the structured, almost, if not outright, formally expressed, wording of a set of rules and regulations*[640].
See also *license*[424] and *contract*[181]. Definitions follow.

3.4.1 Routes as Scripts 100

Paths

92. A path is a triple:

    a) a hub identifier, h_i, a link identifier, l_j, and another hub identifier, h_k, distinct from h_i,

    b) such that there is a link ℓ with identifier l_j in a net n such that $\{h_i, h_k\}$ are the hub identifiers that can be observed from ℓ.

```
type
92.  Pth = HI × LI × HI
axiom
92a.  ∀ (hi,li,hi′):Pth • ∃ n:N,l:L • l ∈ obs_Ls(n) ⇒
92b.      obs_LI(l)=li ∧ obs_HIs(l)={hi,hi′}
```

101

93. From a net one can extract all its paths:

    a) if l is a link of the net,

    b) l_j its identifier,

    c) $\{h_i, h_k\}$ the identifiers of its connected hubs,

    d) then (h_i, l_j, h_k) and (h_k, l_j, h_j) are paths of the net.

```
value
93.  paths: N → Pth-set
93a.  paths(n) ≡
93d.    {(hi,lj,hk),(hk,lj,hi)|l:L,lj:LI,hi,hk:HI•l ∈ obs_Ls(n) ∧
93b.                          lj=obs_LI(l) ∧
93c.                          {hi,hk}=obs_HIs(l)}
```

102

94. From a net descriptor one can (likewise) extract all its paths:

    a) Let h_i, h_k be any two distinct hub identifiers of the net descriptor (definition set),

    b) such that they both map into a link identifier l_j,

    c) then (h_i, l_j, h_k) and (h_k, l_j, h_j) are paths of the net.

```
value
93.  paths: ND → Pth-set
93.  paths(nd) ≡
94a.    {(hi,lj,hk),(hk,lj,hi)|hi,hk:HI,lj:LI • hi≠hk ∧ {hi,hk}⊆dom nd ⇒
94b.                          lj ∈ dom nd(hi)∩ dom nd(hk)}
```

Routes 103

95. A route of a net is a sequence of zero, one or more paths such that
    a) all paths of a route are paths of the net and
    b) adjacent paths in the sequence “share” hub identifiers.

```
type
95.   R = Pth*
axiom
95.   ∀ r:R, ∃ n:N •
95a.    elems r ⊆ paths(n) ∧
95b.    ∀ i:Nat • {i,i+1}⊆inds r ⇒
95b.      let (_,_,hi)=r(i), (hi′,_,_)=r(i+1) in hi=hi′ end
```

104

96. From a net, n, we can generate the possibly infinite set of finite and possibly infinite routes:
    a) $<>$ is a path (***basis clause 1***);
    b) if p is a path of n then $< p >$ is a path of n (***basis clause 2***);
    c) if r and r' are non-empty routes of n
       i. and the last h_i of r is the same as the first h_j of r'
       ii. then the concatenation of r and r' is a route

       (***induction clause***).
    d) Only such routes which can be formed by a (finite, respectively infinite) application of basis clauses Items 96a and 96b and induction clause Item 96c are routes (***extremal clause***).

105

```
value
96.      routes: N|ND → R-infset
96.      routes(nond) ≡
96a.       let rs = {⟨⟩} ∪
96b.             {⟨p⟩|p:Pth•p ∈ paths(nond)} ∪
96(c)ii.          {r⌢r′|r,r′:R • r ∈ rs ∧ r′ ∈ rs ∧
96(c)i.               ∃ hi,hi′,hi″,hi‴:H,li:LI •
96(c)i.               r=r″⌢⟨(hi,li,hi′)⟩∧r′=⟨(hi″,li′,hi‴)⟩⌢r‴ ∧
96(c)i.               hi′=hi″} in
96d.       rs end
```

108

3.4.2 Bus Timetables as Scripts 106

Buses

97. Buses are vehicles,
98. with bus identifiers being the same as vehicle identifiers.

```
type
97.   B
98.   BI ⊆ VI
```

Bus Stops

99. A link bus stop indicates the link (by its identifier), the from and to hub identifiers, and the fraction “down the link” from the from to the to hub identifiers.

```
type
99.   BS = mkL_BS(sel_fhi:HI,sel_li:LI,sel_f:F,sel_thi:HI)
```

Bus Routes 107

100. A bus stop list is a sequence of two or more bus stops, *bsl*.
101. A bus route, *br*, is a pair of a net route, r, and a bus stop list , *bsl*, such that route r is a route of n and such that *bsl* is embedded in r. If
     a) there exists an index list, *il*, of ascending indices of the route r and of the length of *bsl*
     b) such that the ith path of r
     c) share from and to hub identifiers and link identifier with the $il(i)$th bus stop of *bsl*

     then *bsl* is embedded in r.
102. We must allow for two or more stops along a bus route to be adjacent on the same link — in which case the corresponding fractions must likewise be ascending.

```
value
  n:N
type
100.   BSL = BS*
101.   BR = {|(r,bsl):(R×BSL)•wf_BR(r,bsl)|}
```

```
value
101.   wf_BR: BR → Bool
101.   wf_BR(r,bsl) ≡ ∃ n:N,r:R•r ∈ routes(n) ∧ is_embedded_in(r,bsl)

101a.  is_embedded_in: BR → Bool
101a.  is_embedded_in(r,bsl) ≡
101b.    ∃ il:Nat* • len il=len bsl∧inds il⊆inds r∧ascending(il) ⇒
101c.      ∀ i:Nat • i ∈ inds il ⇒
101c.        let (hi,lj,hk) = r(il(i)),(hi′,lj′,f,hk′) = bsl(i) in
101c.        hi=hi′ ∧ lj=lj′ ∧ hk=hk′ end ∧
102.       ∀ i:Nat • {i,i+1}⊆inds il ⇒
102.         let (hi,lj,f,hk)=bsl(i),(hi′,lj′,f′,hk′)=bsl(i+1) in
102.         hi=hi′ ∧ lj=lj′ ∧ hk=hk′ ⇒ f<f′ end

       ascending: Nat* → Bool, ascending(il) ≡ ∀ i:Nat•{i,i+1}⊆inds il ⇒ il(i)≤il(i+1)
```

The $\leq$ of the ascending predicate allows for more than one stop along the same route

112

Bus Schedule 109

103. A timed bus stop is a pair of a time and a bus stop.

104. A timed bus stop list is a sequence of timed bus stops.

105. A bus schedule is a pair of a route and a timed bus stop list such that

   - there is a net of which the routes is indeed a route,
   - the bus stop list of the timed bus stop list is embedded in the route, and
   - ‘later” listed bus stops register later times.

106. SimpleBusSchedules remove routes from BusRoutes.

110

```
type
103.  TBS :: sel_T:T  sel_bs:BS
104.  TBSL = TBS*
105.  BusSched = {|(r,tbsl):(R×TBSL)•wf_BusSched(r,tbsl)|}
value
105.  wf_BusSched: BusSched → Bool
105.  wf_BusSched(r,tbsl) ≡
105.    ∃ n:N•r ∈ routes(n)
105.    ∧ let bsl:SBS = ⟨sel_BS(tbsl(i))|i:[1..len tbsl]⟩ in is_embedded_in(r,bsl) end
105.    ∧ ∀ i:Nat•{i,i+1}⊆inds tbsl ⇒ sel_T(tbsl(i))<sel_T(tbsl(i+1))
type
106.  SBS = {|bsl:BS*•∃ n:N,r:R•r ∈ routes(n)∧is_embedded_in(r,bsl)|}
```

113

Timetable 111

The concept of a bus line captures all those bus schedules which ply the same bus route but at different times. A timetable is made up from distinctly named bus lines.

107. A bus line has a unique bus line name.

108. We say that two bus schedules are the same if they are based on the same route and if they differ only in their times.

109. Each of the different bus routes of a bus line has a unique bus number.

110. A route bus schedule pairs a route with simple bus schedules for each of a number of busses (identified by their bus number).

111. A bus timetable (listing, map) maps bus line names to route bus schedules.

112. A timetable is a pair, a net and a table.

113. A well-formed timetable must satisfy same bus schedules within each bus line

114. All bus numbers are distinct across bus lines.

```
type
107.  BLNm
value
108.  same_bus_schedule: BusSched × BusSched → Bool
108.  same_bus_schedule((r1,btl1),(r2,btl2)) ≡
108.    r1 = r2 ∧ len btl1 = btl2 ∧
108.    ⟨sel_BS(btl1(i))|i:[1..len btl1]⟩=⟨sel_BS(btl2(i))|i:[1..len btl2]⟩
type
109.  BNo
110.  RBS :: sel_R:R  sel_btbl:(BNo →m SBS)
111.  TBL = BLNm →m RBS
112.  TT′ = ND × TBL
113.  TT  = {|tt:TT′•wf_TT(tt)|}

value
113.  wf_TT: TT′ → Bool
113.  wf_TT(_,tbl) ≡
113.    ∀ bln:BLNm•bln ∈ dom tbl ⇒
113.      ∀ bno,bno′:BNo • {bno,bno′}⊆dom sel_btbl(tbl(bln)) ⇒
113.        same_bus_schedule(sel_R(tbl(bln)),sel_btbl(tbl(bln))(bno),
113.                          sel_R(tbl(bln)),sel_btbl(tbl(bln))(bno′)) ∧
114.    ∀ bln′,bln″:BLNm • {bln′,bln″}⊆dom tbl ∧ bln′≠bln″ ⇒
114.      dom sel_btbl(tbl(bln′)) ∩ dom sel_btbl(tbl(bln″)) = {}
```

3.4.3 Route and Bus Timetable Denotations 114 121

What are routes and bus timetables scripting ?

Routes (list of connected link traversal designations) script that one may transport people or freight along the sequence of designated links.

Bus timetables script (at least) two things: the set of bus traffics on the net which satisfy the bus timetable, and information that potential and actual bus passengers may, within some measure of statistics (and probability), rely upon for their bus transport. 115

Here, we shall not develop the idea of bus timetables denoting certain traffics. Instead we refer to our previously sketched model of traffics (Sect. 3.3.2, Pages 37–40).

Route (designations) and bus timetables script potential and actual route travels, respectively script the dispatch of buses and their travelling.

Bus timetables can also be seen as a form of contracts between the bus operators offering the bus services and potential and actual passengers, with the contract promising timely transport. In the next section, Sect. 3.4.4, we shall sketch a language of bus service contracts and bus service actions implied by such contracts.

3.4.4 Licenses and Contracts 116

Definition: License. *A license is a* script[651] *specifically expressing a permission to act; is freedom of action; is a permission granted by competent authority to engage in a business or occupation or in an activity otherwise unlawful; a document, plate, or tag evidencing* 123 *a license granted; a grant by the holder of a copyright or patent to another of any of the rights embodied in the copyright or patent short of an assignment of all rights.*
Licenses appear more to have morally than legally binding poser. 117
Definition: Contract. *A contract is a special kind of* license[424] *specifically expressing a legally binding agreement between two or more parties — hence a document describing the conditions of the contract; a contract is business arrangement for the supply of goods or services at fixed prices, times and locations. In software development a contract specifies* 118 *what is to be developed: (1) a* domain description[243]*, (2) a* requirements prescription[615]*, or (3) a* software design[688]*; or a combination of these (1–2, 2–3, 1–3). A contract further specifies* 119 *how it might, or must be developed; criteria for acceptance of what has been developed; delivery dates for the developed items; who the "parties" to the contract are: the* client[116] *and the* developer[227]*, etc.* 120
For a comprehensive treatment of licenses and contracts we refer to [48, Chapter 10, Sect. 10.6 (Pages 309–326) [84]].

We shall illustrate fragments of a language for bus service contracts.

The background for the bus contract language is the following. In many large cities 124 around Europe the city or provincial government secures public transport in the form of bus services operated by many different private companies. Section 3.4.2 illustrated the concept of bus (service) timetables. The bus services implied by such a timetable, for a city area — with surrounding suburbs etc. — need not be implemented by just one company, but can be contracted, by the city government public transport office, to several companies,

each taking care of a subset of the timetable. Different bus operators then take care of non-overlapping parts and all take care of the full timetable. It may even be that extra buses need be scheduled, on the fly, in connection with major sports or concert or other events. Bus operators may experience vehicle breakdowns or bus driver shortages and may be forced to subcontract other, even otherwise competing bus operators to "step in" and alleviate the problem.

Contracts 122 Schematically we may represent a bus contract as follows:

> **Contract** cn **between contractee** ci **and contractor** cj:
>   **This contract contracts** cj **in the period** $[t,t']$ **to**
>   **perform the following services with respect to timetable** tt:
>   **operate bus lines** $\{blj_1,blj_2,...,blj_n\}$
>   **subject to the following occasional exceptions:**
>   **cancellation of bus tours:**
> $\{(blj_a,\{bno_{a_1},...,bno_{a_m}\}),...\}$ **subject to conditions** cbt
>   **insertion of bus tours on lines**
> $\{blj_\alpha,blj_\beta,...,blj_\gamma\}$ **subject to conditions** ibt
>   **subcontracting bus tours on lines**
> $\{blj_\delta,blj_\phi,...,blj_\omega\}$ **subject to conditions** scbt.

115. A bus contract has a header with the distinct names of a contractee and a contractor and a time interval.

116. A bus contract presents a timetable.

117. A bus contract presents a set of bus lines (by their identifiers) such that these are in the timetable.

118. And a bus contract may list one or more of three kinds of "exceptions":

    a) cancellation of one or more named bus tours on one or more bus lines subject to certain (specified) conditions;

    b) insertion of one or more extra bus tours on one or more bus lines subject to certain (specified) conditions;

    c) subcontracting one or more unspecified bus tours on one or more bus lines subject to certain (specified) conditions — to further unspecified contractors.

    We abstract the above quoted "one or more of three kinds of exceptions" as one possibly empty clause for each of these alternatives.

119. A bus contract now contains a header, a timetable, the subject bus lines and the exceptions,

120. such that
    a) line names mentioned in the contract are those of the bus lines of the timetable, and
    b) bus (tour) numbers are those of the appropriate bus lines in the timetable.
121. The calendar period is for at least one full day, midnight to midnight.
122. A named contract is a pair of a contract name and a contract.

125

type
```
115.   CNm, CId, D, T, CON
115.   CH = CId × CId × (D×D)
116.   CT = TT
117.   CLs = BLNm-set
118.   CE = (CA × IN × SC) × CON
118a.  CA = BLNm →m BNo-set
118b.  IN = BLNm →m BNo-set
118c.  SC = BLNm-set
119.   CO′ = CH × CT × CLs × CE
120.   CO = {|co:CO′•wf_CO(co)|}
122.   NCO = CNm × CO
```

126

value
```
120.  wf_CO: CO′ → Bool
120.  wf_CO((ce,cr,(d,d′)),(nd,tbl),cls,((blns,blns′,bls),con)) ≡
117.      ce ≠ cr ∧
120a.     cls ⊆ dom tbl ∧
120b.     ∀ bli,bli′:BLNm • bli ∈ dom blns ∧ bli′ ∈ dom blns′ ⇒
120a.         {bli,bli′} ⊆ dom tbl ∧
120b.         blns(bli) ∪ blns′(bli′) ⊆ dom sel_btbtl(tbl(bli)) ∧
120a.     bls ⊂ dom tbl ∧
121.      d < d′
```

Contractual Actions

127 An bus operator can now perform a number of actions according to a contract. We schematise these:

For contract cn **commence bus tour, line:** bli **and bus no.:** bno

For contract cn **cancel bus tour, line:** bli **and bus no.:** bno

For contract cn **insert extra bus tour, line:** bli **and bus no.:** bno

Subcontract with respect to contract cn **the following:**
  Contract cn′**: for the calendar period** [d,d′] **contractee** ci **contracts contractor** cj
    to perform the following services with respect to timetable tt:
      operate bus lines $\{blj_1,blj_2,...,blj_n\}$
      subject to the following occasional exceptions:
        cancellation of bus tours:
          $\{(blj_c,\{bno_{c_1},...,bno_{c_m}\}),...\}$ **subject to conditions** cbt
        insertion of bus tours on lines
          $\{(blj_i,\{bno_{i_1},...,bno_{i_n}\}),...\}$ **subject to conditions** ibt
        subcontracting bus tours on lines
          $\{blj_\delta,blj_\phi,...,blj_\omega\}$ **subject to conditions** scbt.

128

123. A bus operator action is either a commence, a cancellation, an insertion or a subcontracting action. All actions refer to the (name of) the contract with respect to which the action is contracted.
    a) A `commence` action designator states the bus line concerned and the bus number of that line.
    b) A cancellation action designator states the bus line concerned and the bus number of that line.
    c) An insertion action designator states the bus line concerned and the bus number of that line — for which an extra bus is to be inserted.[10]
    d) A subcontracting action designator, besides the name of the contract with respect to which the subcontract is a subcontract, state a named contract (whose contract name is unique).

129

type
```
123.   Act = Com | Can | Ins | Sub
123a.  Com == mkCom(sel_cn:CNm,sel_bli:BLNm,sel_bno:BNo)
123b.  Can == mkCan(sel_cn:CNm,sel_bli:BLNm,sel_bno:BNo)
123c.  Ins == mkIns(sel_cn:CNm,sel_bli:BLNm,sel_bno:BNo)
123d.  Sub == mkSub(sel_cn:CNm,sel_con:NCO)
```

[10]The insertion of buses in connection with either unscheduled or extraordinary (sports, concerts, etc.) events can be handled by special, initial contracts.

Wellformedness of Contractual Actions 130

124. In order to express wellformedness conditions, that is, pre-conditions, for the action designators we introduce a context which map contract names to contracts.

125. Wellformedness of a contract is now expressed with respect to a context.

type
124. CTX = CNm $\underset{m}{\overrightarrow{\quad}}$ CO
value
125. wf_Act: Act → CTX → **Bool**

131

- Let a defined cnm entry in ctx be a contract: ((ce,cr),(nd,tbl),cls,(blns,bls,bls′),(d,d′)).

126. If cmd is a commence command mkCom(cnm,bln,bno), then
    a) contract name cnm must be defined in context ctx;
    b) bus line name bln must be defined in the contract, that is, in cls, and
    c) bus number bno must be defined in the bus table part of table tbl.

126. wf_Act(mkCom(cnm,bln,bno))(ctx) ≡
126a. cnm ∈ **dom** ctx ∧
126. **let** ((ce,cr),(nd,tbl),cls,(blns,bls,bls′),(d,d′)) = ctx(cnm) **in**
126b. bln ∈ cls ∧
126c. bno ∈ **dom** sel_btbl(tbl(bln)) **end**

132

127. cancellation and insertion commands have the same static wellformedness conditions as have commence command.

127. wf_Act(mkCan(cnm,bln,bno))(ctx) ≡ wf_Act(mkCom(cnm,bln,bno))(ctx)
127. wf_Act(mkIns(cnm,bln,bno))(ctx) ≡ wf_Act(mkCom(cnm,bln,bno))(ctx)

133

128. If cmd is a subcontract command then

    Let the subcontract command and the cnm named contract in ctx be
    mkSub(cnm,nco:(cnm′,(ce′,cr′,(d″,d‴)),(nd′,tbl′),cls′,(blns′,bls″,bls‴)))
    respectively ((ce,cr,(d,d′)), (nd,tbl), cls, (blns,bls,bls′)).

    a) contract name cnm must be defined in context ctx;
    b) contract name cnm′ must not be defined in context ctx;

    c) the calendar period of the subcontract must be within that of the contract from which it derives;
    d) the net descriptors nd and nd′ must be identical;
    e) the tables tbl and tbl′ and must be identical and
    f) the set, cls′, of bus line names that are the scope of the subcontracting must be a subset of bls′.

134

128. wf_Act(mkSub(cnm,nco:(cnm′,co:((ce′,cr′,(d″,d‴)),(nd′,tbl′),cls′,(blns′,blns″,bls‴)))))(ctx)
128a. cnm ∈ **dom** ctx ∧
128. **let** co′ = ((ce,cr,(d,d′)),(nd,tbl),cls,(blns,blns′,bls′)) = ctx(cnm) **in**
128b. cnm′ ∉ **dom** tbl ∧
128c. d ≤ d″ ≤ d‴ ≤ d′ ∧
128d. nd′ = nd ∧
128e. tbl′ = tbl ∧
128f. cls′ ⊆ bls′ **end**

135 Wellformedness of contracts, wf_CO(co) and wf_CO(co′), secures other constraints.

We do not here bring any narrated or formalised description of the semantics of contracts and actions. First such a description would be rather lengthy. Secondly a specification would be more of a requirements prescription.

3.5 Management and Organisation 136

Definition: Management. *Management is about resource[620]s: their acquisition[11], scheduling[646] (over time), allocation[33] (over locations), deployment (in performing actions) and disposal ("retirement"). We distinguish between board-directed, strategic, tactical and operational actions.* 137 *Board-directed actions target mainly financial resources: obtaining new funds through conversion of goodwill into financial resources, acquiring and selling "competing" or "supplementary" business units. Strategic actions (see Item 716 on page 221) convert financial resources into production, service supplies and resources and vice-versa — and in this these actions schedule availability of such resources. Tactical actions (see Item 741 on page 223) mainly allocate resources. Operational actions order, monitor and control the deployment of resources in the performance of actions.* 138
Definition: Organisation. *Organisation is about the "grand scale", executive and strategic national, continental or global (world wide) (i) allocation of major resource (e.g., business) units, whether in a hierarchical, in a matrix, or in some other organigram-specified structure, (ii) as well as the clearly defined relations (which information, decisions and actions are transferred) between these units, and (iii) organisational dynamics.* 139
Definition: Management & Organisation. *The composite term management and organisation applies in connection with management[444] as outlined just above and with organisation[500] also outlined above. The term then emphasises the relations between the organisation and management of an enterprise.*

• • •

The borderlines within management actions and across organisation "layouts" are fuzzy.

3.5.1 Transport System Examples 140

We shall only present sketchy examples of management and organsation.[11] 144

- Executive actions: Deciding on major re-organisation of a transport net (for example introduction of toll roads or freeways, road pricing, major bridges across wide waters [potentially connecting two hitherto unconnected nets], and their management) are executive actions. So are decisions on merging or splitting transport from or into several transport services. Reorganising an enterprise from one characterised by a 141 "deep" hierarchy of management layers (a hierarchy which may very well exemplify highly centralised both administrative and functional monitoring and control) into a matrix of two "shallow" hierarchies, one which addresses tactical and operational management and one which addresses executive and strategic management — with the former (the operations) being replicated across geographical areas while the latter applies "globally" — such reorganisations reflect executive actions (but are carried out by strategic and tactical management). 142 148
- Strategic actions: Adding or removing transport links, or major reorganisation of bus timetables are strategic actions. Splitting a(n own) contract into what is still to be operated and subcontracting other parts, for definite, to other bus operators are also strategic actions.
- Tactical actions: Insertion and cancellation of bus services are tactical actions. Subcontracting some parts of a timetable demanded service, for a short while, to other bus operators could be considered tactical actions. 149
- Operational actions: Commencing and thus, in general, allocating drivers to and sending these off on bus services are operational actions. So are announcing insertion of new (unscheduled) and cancellation of scheduled routes.

[11]Two remarks: (1) From an albeit superficial study of curricula of a number of business schools it seems, to this author, that the decomposition in *management and organisation* and into *executive, strategic, tactical and operational actions* is not quite the way the *financial, market, sales, product and production* (business administration) aspects of enterprises are looked upon in these schools. (2) We have, in [30], studied issues of management and organisation, and we shall elsewhere study these from the point of view of the signatures of $\mathcal{E}$xecutive, $\mathcal{S}$trategic, $\mathcal{T}$actical and $\mathcal{O}$perational functions as they apply to and results in one or more of the resource types: $\mathbb{F}$inance, $\mathbb{R}$esource, spatial $\mathbb{L}$ocation and $\mathbb{T}$emporal notions of "business environments" ($\rho : ENV$ which binds resource names to $\mathbb{SCHED}$ules) and "business states" ($\sigma : \Sigma$ which binds resource names to resource values) — and where $\mathbb{SCHED}$ules binds resource names to time intervals and [al]locations.

3.6 Human Behaviour 143

Definition: Human Behaviour. *By human behaviour we shall here understand the way a human follows the enterprise rules and regulations[640] as well as interacts with a machine[436]: dutifully honouring specified (machine dialogue[230] or) protocol[561]s, or negligently so, or sloppily not quite so, or even criminally not so! Human behaviour is a facet[285] of the domain[239]. We shall thus model human behaviour also in terms of it failing to react properly, i.e., humans as non-deterministic agent[24]s!*

3.7 Towards Theories of Domain Facets 145

3.7.1 A Theory of Intrinsics 146

3.7.2 Theories of Support Technologies 147

An Example Traffic (tf:TF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) vehicles (tn:TN).

Conventional optical sensors sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic (stf:sTF). Hence the collection of all optical sensors, for any given net, is a partial function from intrinsic (itf) to sampled train traffics (stf).

We need to express quality criteria that any optical sensor technology should satisfy — relative to a necessary and sufficient description of a closeness predicate.

> For all intrinsic traffics, itf, and for all optical sensor technologies, og, the following must hold: Let stf be the traffic sampled by the optical gates. For all time points, t, in the sampled traffic, those time points must also be in the intrinsic traffic, and, for all trains, tn, in the intrinsic traffic at that time, the train must be observed by the optical gates, and the actual position of the train and the sampled position must somehow be checkable to be close, or identical to one another.

Since hubs change state with time, n:N, the net needs to be part of any model of traffic.

```
type
   T, TN
   P = HP | LP
   NetTraffic :: net:N × trf:(V →ₘ P)
   iTF = T → NetTraffic
   sTF = T →ₘ NetTraffic
   oG = iTF ~→ sTF
value
   [close] c: NetTraffic × TN × NetTraffic ~→ Bool
axiom
   ∀ itt:iTF, og:OG • let stt = og(itt) in
      ∀ t:T • t ∈ dom stt •
```

```
    t ∈ $\mathbb{DOM}$ itt ∧ ∀ Tn:TN • tn ∈ dom trf(itt(t))
        ⇒ tn ∈ dom trf(stt(t)) ∧ c(itt(t),tn,stt(t)) end
```

$\mathbb{DOM}$ is not an RSL operator. It is a mathematical way of expressing the definition set of a general function. Hence it is not a computable function.

Checkability is an issue of testing the optical sensors when delivered for conformance to the closeness predicate, i.e., to the axiom.

General 150 The formal requirements can be narrated:Let Θ_i and Θ_a designate the spaces of intrinsic and actual-world configurations (contexts and states). For each intrinsic configuration model — that we know is support technology assisted — there exists a support technology solution, that is, a total function from all intrinsic configurations to corresponding actual configurations. If we are not convinced that there is such a function then there is little hope that we can trust this technology

```
type
   $\Theta_i$, $\Theta_a$
   ST = $\Theta_i \to \Theta_a$
axiom
   ∀ sts:ST-set, st:ST • st ∈ sts ⇒ ∀ $\theta_i$:$\Theta_i$, ∃ $\theta_a$:$\Theta_a$ • st($\theta_i$) = $\theta_a$
```

3.7.3 A Theory of Rules & Regulations 151

There are, abstractly speaking, usually three kinds of languages involved wrt. (i.e., when expressing) rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules, respectively regulations; and one, Stimulus, exists for describing the form of the [always current] domain action stimuli. 152

A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \to \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, sy_rul:Rule, stands for, i.e., has as its semantics, its meaning, rul:RUL, a predicate over current and next configurations, $(\Theta \times \Theta) \to$ **Bool**, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express: If the predicate holds then the stimulus will result in a valid next configuration. 153

```
type
   Stimulus, Rule, Θ
   STI = Θ → Θ
   RUL = (Θ × Θ) → Bool
value
   meaning: Stimulus → STI
   meaning: Rule → RUL
```

```
   valid: Stimulus × Rule → Θ → Bool
   valid(sy_sti,sy_rul)($\theta$) ≡ meaning(sy_rul)($\theta$,(meaning(sy_sti))($\theta$))

   valid: Stimulus × RUL → Θ → Bool
   valid(sy_sti,se_rul)($\theta$) ≡ se_rul($\theta$,(meaning(sy_sti))($\theta$))
```

154 A syntactic regulation, sy_reg:Reg (related to a specific rule), stands for, i.e., has as its semantics, its meaning, a semantic regulation, se_reg:REG, which is a pair. This pair consists of a predicate, pre_reg:Pre_REG, where Pre_REG = $(\Theta \times \Theta) \to$ **Bool**, and a domain configuration-changing function, act_reg:Act_REG, where Act_REG = $\Theta \to \Theta$, that is, both
155 involving current and next domain configurations. The two kinds of functions express: If the predicate holds, then the action can be applied.

The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function.
156

```
type
   Reg
   Rul_and_Reg = Rule × Reg
   REG = Pre_REG × Act_REG
   Pre_REG = Θ × Θ → Bool
   Act_REG = Θ → Θ
value
   interpret: Reg → REG
```

157 The idea is now the following: Any action of the system, i.e., the application of any stimulus, may be an action in accordance with the rules, or it may not. Rules therefore
158 express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort.

More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where θ' = (meaning(sy_sti))(θ). Let θ' (= (meaning(sy_sti))(θ)) violate the rule, i.e., ∼valid(sy_sti,sy_rul)(θ), then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,θ' then the action part, act_reg, of the meaning of the regulation, sy_reg, must be
159 applied, act_reg(θ'), to remedy the situation.

```
axiom
   ∀ (sy_rul,sy_reg):Rul_and_Regs •
      let se_rul = meaning(sy_rul),
            (pre_reg,act_reg) = meaning(sy_reg) in
      ∀ sy_sti:Stimulus, $\theta$:Θ •
```

```
    ∼valid(sy_sti,se_rul)(θ)
        ⇒ let θ′ = (meaning(sy_sti))(θ) in
            pre_reg(θ,θ′)
            ⇒ ∃ nθ:Θ • act_reg(θ′)=nθ ∧ se_rul(θ,nθ)
  end end
```

160 It may be that the regulation predicate fails to detect applicability of regulations actions. That is, the interpretation of a rule differs, in that respect, from the interpretation of a regulation. Such is life in the domain, i.e., in actual reality

3.7.4 A Theory of Management & Organisation 161

3.7.5 A Theory of Human Behaviour 162

Commensurate with the above, humans interpret rules and regulations differently, and not always "consistently" — in the sense of repeatedly applying the same interpretations.

163 Our final specification pattern is therefore:

```
type
    Action = Θ $\xrightarrow{\sim}$ Θ-infset
value
    hum_int: Rule → Θ → RUL-infset
    action: Stimulus → Θ → Θ
    hum_beha: Stimulus × Rules → Action → Θ $\xrightarrow{\sim}$ Θ-infset
    hum_beha(sy_sti,sy_rul)(α)(θ) as θset
      post
        θset = α(θ) ∧ action(sy_sti)(θ) ∈ θset
        ∧ ∀ θ′:Θ•θ′ ∈ θset ⇒
          ∃ se_rul:RUL•se_rul ∈ hum_int(sy_rul)(θ)⇒se_rul(θ,θ′)
```

164 The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not

4 An Ontology of Requirements Constructions 165

Definition: Requirements. *A condition or capability needed by a user to solve a problem or achieve an objective [133].*

166 **Definition: Machine.** *By the machine we understand the hardware[331] plus software[685] that implements some requirements[605], i.e., a computing system[151].*

Definition: Requirements Unit. *By a requirements unit[618] we mean a single sentence which expresses an "isolated" requirements. (We omit charaterising "single sentence" and "isolated".)*

Definition: Requirements Prescription. *By a requirements[605] prescription[540] we mean just that: the prescription of some requirements. Sometimes, by requirements prescription, we mean a relatively complete and consistent specification of all requirements, and sometimes just a requirements unit[618].*

167 **Definition: Requirements Engineering.** *The engineering of the development of a requirements prescription[615], from identification of requirements[605] stake-holders, via requirements acquisition[606], requirements analysis[607], and requirements prescription[615] to requirements validation[800] and requirements verification[807].*

We shall just focus on *requirements prescription*[615], that is, the modelling of *requirements*[605].

4.1 Business Process Re-engineering 168

Definition: Business Process. *By a business process we shall understand a behaviour[79] of an enterprise, a business, an institution, a factory. A business process reflects the ways in which a business conducts its affairs, and is a facet[285] of the domain[239]. Other facets of an enterprise are those of its intrinsics[399], support technology[725], rules and regulations[640], management and organisation[445] (a facet closely related to business processes), and human behaviour[345].*

169 **Definition: Business Process Engineering.** *By business process engineering[100] we shall understand the design[221], the determination, of business process[99]es. In doing business process engineering one is basically designing, i.e., prescribing entirely new business processes.*

170 **Definition: Business Process Re-engineering.** *By business process reengineering[101] we shall understand the re-design[221], the change, of business process[99]es. In doing business process re-engineering one is basically carrying out change management[109].*

4.1.1 The Kinds of Requirements 171

We distinguish between three kinds of requirements: (Sect. 4.2) the **domain requirements** are those requirements which can be expressed solely using terms of the domain; (Sect. 4.4) the **machine requirements** are those requirements which can be expressed solely using terms of the machine, and (Sect. 4.3) the **interface requirements** are those requirements which must use terms from both the domain and the machine in order to be expressed.

4.1.2 Goals Versus Requirements 172

Whereas a domain description presents a domain **as it is**, a requirements prescription presents a domain **as it would be** if some required **machine** was implemented (from these requirements). The **machine** is the **hardware** plus **software** to be designed from the requirements. That is, the *machine* is what the requirements are about. 173

We make a distinction between **goals** and **requirements.** Goals are what we expect satisfied by the software implemented from the requirements. But goals could also be of the system for which the software is required. First we exemplify the latter, then the former.

Goals of a Toll Road System 174

- A goal for a toll road system may be 178
  - to decrease the travel time between certain hubs and
  - to lower the number of traffic accidents between certain hubs,

Goals of Toll Road System Software 175

- The goal of the toll road system software is to help automate
  - the recording of vehicles entering, passing and leaving the toll road system
  - and collecting the fees for doing so.

Goals are usually expressed in terms of properties. Requirements can then be proved to satisfy the $\mathcal{G}$oals: $\mathcal{D}, \mathcal{R} \models \ \mathcal{G}$. [148, Lamsweerde] focus on goals. 180

Arguing Goal-satisfaction of a Toll Road System 176

- By endowing links and hubs with average traversal times for both ordinary road and for toll road links and hubs 181
  - one can calculate traversal times between hubs
  - and thus argue that the toll road system satisfies [significantly] "quicker" traversal times.
- By endowing links and hubs with traffic accident statistics (real, respectively estimated)
  - for both ordinary road and for toll road links and hubs
  - one can calculate estimated traffic accident statistics between all hubs
  - and thus argue that the combined ordinary road plus toll road system satisfies [significantly] lower traffic fatalities. 182

Arguing Goal-satisfaction of Toll Road System Software 177

- By recording
  - tickets issued and collected at toll booths and
  - toll road hubs and links entered and left
  - as per the requirements specification brought in forthcoming examples (Sects. 4.2.1–4.2.4),
- we can eventually argue that
  - the requirements of the forthcoming examples (Sects. 4.2.1–4.2.4)
  - help satisfy the goal of the example **??** on page ??.

We shall assume that the (goal and) requirements engineer elicit both $\mathcal{G}$oals and $\mathcal{R}$equirements from requirements stake-holders.

$\mathcal{D}, \mathcal{R} \models \mathcal{G}$ The $\mathcal{G}$oals can be argued to hold by reasoning over the $\mathcal{R}$equirements and the $\mathcal{D}$omain.

But we shall focus only on domain and interface requirements such as "derived" from domain descriptions.

4.1.3 Re-engineered Nets 179

The nets defined in Sect. 3 could be of any topology. They could consist of two or more nets that were not linked to one another; they could consist of connected nets or nets that were acyclic; etc.; and the nets were not specifically road, rail, sea lane or air lane nets. We shall now consider a special kind of road nets: basically the road nets we have in mind are linear sequences of pairs of links of opposite direction link "states", where these links, let us call them toll road links, are connected to toll road hubs; where, in addition, these toll road hubs are linked, via toll plazas (i.e., "special" hubs) to toll road hubs by means of on/off links.

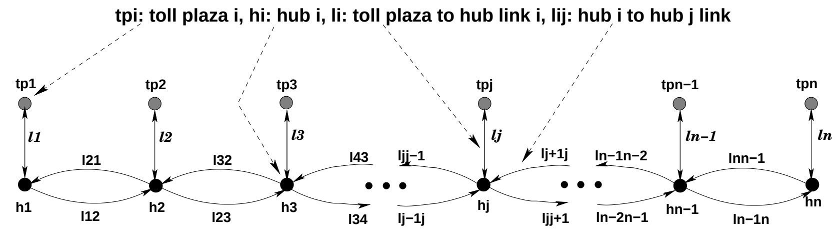

Figure 3: A Toll Road System

We do not consider the general nets that are (possibly) connected to the toll plazas. The pragmatics behind these nets is the following: Drivers enter and leave the toll road nets at toll road plazas; collect tickets from toll road plaza ticket-issuing booths when entering the toll road net and present these at toll road plaza ticket-collection booths and pay according to some function of the time and length (from entry to exit plaza) driven on the toll road net when leaving the net; drivers are otherwise free to "circle" the toll road net as they see fit: multiple times "up and down" the net, circling toll road hubs, etc. Our sketch centers 183 around a toll road net with toll booth plazas. The BPR focuses first on entities, actions, events and behaviours (Sect. 2), then on the six domain facets (Sect. 3). 184

189

129. **Re-engineered Entities:** We shall focus on a linear sequence of toll road intersections (i.e., hubs) connected by pairs of one-way (opposite direction) toll roads (i.e., links). Each toll road intersection is connected by a two way road to a toll plaza. Each toll plaza contains a pair of sets of entry and exit toll booths. (Sect. 4.2.2 brings more details.) 185

130. **Re-engineered Actions:** Cars enter and leave the toll road net through one of the toll plazas. Upon entering, car drivers receive, from the entry booth, a plastic/paper/electronic ticket which they place in a special holder in the front window. Cars arriving at intermediate toll road intersections choose, on their own, to turn either "up" the toll road or "down" the toll road — with that choice being registered by the electronic ticket. Cars arriving at a toll road intersection may choose to "circle" around that intersection one or more times — with that choice being registered by the electronic ticket. Upon leaving, car drivers "return" their electronic ticket to the exit booth and pay the amount "asked" for. 186

191

131. **Re-engineered Events:** A car entering the toll road net at a toll both plaza entry booth constitutes an event. A car leaving the toll road net at a toll both plaza entry booth constitutes an event. A car entering a toll road hub constitutes an event. A car entering a toll road link constitutes an event. 187

193

132. **Re-engineered Behaviours:** The journey of a car,from entering the toll road net at a toll booth plaza, via repeated visits to toll road intersections interleaved with repeated visits to toll road links to leaving the toll road net at a toll booth plaza, constitutes a behaviour — withreceipt of tickets, return of tickets and payment of fees being part of these behaviours. Notice that a toll road visitor is allowed to cruise "up" and "down" the linear toll road net – while (probably) paying for that pleasure (through the recordings of "repeated" hub and link entries). 188

133. **Re-engineered Intrinsics:** Toll plazas and abstracted booths are added to domain intrinsics.

134. **Re-engineered Support Technologies:** There is a definite need for domain-describing the failure-prone toll plaza entry and exit booths.

135. **Re-engineered Rules and Regulations:** Rules for entering and leaving toll booth entry and exit booths must be described as must related regulations. Rules and regulations for driving around the toll road net must be likewise be described.

136. **Re-engineered Scripts:** No need.

137. **Re-engineered Management and Organisation:** There is a definite need for domain describing the management and possibly distributed organisation of toll booth plazas.

138. **Re-engineered Human Behaviour:** Humans, in this case car drivers, may not change their behaviour in the spectrum from diligent and accurate via sloppy and delinquent to outright traffic-law breaking – so we see no need for any "re-engineering".

4.2 Domain Requirements 190

Definition: Domain Requirements. *By domain requirements[605] we understand such requirements (save those of business process reengineering[101]) which can be expressed sôlely by using professional terms of the domain[239].*

Definition: Domain Requirements Facet. *By domain requirements[258] facets we understand such domain requirements that basically arise from either of the following operations on domain description[243]s (cum requirements prescription[615]s): domain projection[255], domain determination[245], domain extension[249], domain instantiation[253] and domain fitting[251].*

4.2.1 Projection 192

Definition: Projection. *By projection we shall here, in a somewhat narrow sense, mean a technique that applies to domain description[243]s and yields requirements prescription[615]s. Basically projection "reduces" a domain description by "removing" (or, but rarely, hiding[337]) entities[272], function[310]s, event[281]s and behaviour[79]s from the domain description. If the domain description is an informal one, say in English, it may have expressed that certain entities, functions, events and behaviours might be in (some instantiations of) the domain. If not "projected away" the similar, i.e., informal requirements prescription will express that these entities, functions, events and behaviours shall be in the domain and hence will be in the environment of the machine[436] being requirements prescribed.*

Keep the following parts (items) of the domain:

- from Item 1 on page 13 to and including Item 11 on page 15,
- from Item 51a on page 23 to and including Item 52c on page 24,
- from Item 56 on page 28 to and including Item 72 on page 31 and
- from Item 80 on page 36 to and including Item 91 on page 41.

That is, omit these parts:

- Sect. 2.1.5,
- Sects. 2.3–2.4,
- Sects. 2.5.2–2.5.3,
- Sect. 3.1.7 and
- Sects. 3.2–3.6.

and keep these:

- N, H, L,
- obs_Hs,
- obs_Ls,
- HI, LI,
- obs_HI,
- obs_LI,
- obs_LIs,
- obs_HIs,
- PLAN, LHIM,
- wf_PLAN,
- ND, wf_ND,
- LΣ, LΩ,
- obs_LΣ, obs_LΣ,
- HΣ, HΩ,
- obs_HΣ, obs_HΣ,
- V, VI, VP,
- obs_VI, obs_VP,
- TF, T and
- wf_TF.

4.2.2 Instantiation 194

Definition: Instantiation. *'To represent (an abstraction) by a concrete instance[384]' [213]. Domain instantiation is a domain requirements facet[259]. It is an operation performed on a domain description[243] (cum requirements prescription[615]). Where, in a domain description certain entities and function[310]s are left undefined, domain instantiation means that these entities or functions are now instantiated into constant value[802]s.*

Example 195 The following instantiation prescription only covers the static aspects of the toll road net, i.e., simple entities. That is, the states of hubs and links will first be dealt with in Sect. 4.2.3.

139. A toll road net (a subnet of a larger previously described net) consists of a pair: toll road links and toll road to plaza hubs and links.
    a) The toll road links component is a linear sequence of one or more pairs of toll road links.
    b) The toll road to plaza hubs and links component is a linear sequence of two or more triples of a plaza, a (plaza to toll road hub) link and a toll road hub. 196
    c) The wellformedness of toll road nets are expressed next.
        i. The length of the toll road links sequence is one less than the length of the toll road to plaza hubs and links sequence. The idea is that the toll road links at position i connect the toll road hubs at positions i and $i+1$ of the toll road to plaza hubs and links sequence — i being the indexes of the toll road links sequence.
        ii. All links have distinct link identifiers.
        iii. All hubs and plazas have distinct hub identifiers.
        iv. From the links in the pairs of links, (l_i, l'_i), of position i in the toll road links component one observes exactly the same two element set of hub identifiers,
        v. and these are the identifiers of the hubs at positions i and $i+1$ of the toll road to plaza hubs and links sequence.
        vi. The plaza to toll road hub links are indeed connected to these plazas and hubs; and
        vii. the plaza and toll road hubs are connected only to the links as mentioned above. 197
    d) A toll road plaza is like a hub, with an observable hub identifier (and equipped with ticket-issuing tool booths and ticket-collection and payment toll booths).

```
type
139.    TRN′ = TRLs × PHLs
139.    TRN = {|trn:TRN′•wf_TRN(trn)|}
139a.   TRLs = (L × L)*
139b.   PHLs = (PZ × L × H)*
```

198

```
value
139c.      wf_TRN: TRN′ → Bool
139c.      wf_TRN(trn:(trls,phls)) ≡
139(c)i.       len trls +1 = len phls ∧
139(c)ii.       card xtr_Hs(trn) = card xtr_HIs(trn) ∧
139(c)iii.       card xtr_Ls(trn) = card xtr_LIs(trn)
139(c)iv.       ∀ i:Nat•i ∈ inds trls ⇒
139(c)iv.          let (l,l′)=trsl(i),(p,l″,hi)=phls(i),(_,l‴,hj)=phls(i+1) in
139(c)iv.          obs_HIs(l) = obs_HIs(l′) =
139(c)v.          {obs_HI(hi),obs_HI(hj)}  ∧
139(c)vii.         case i of
139(c)vii.            1 → obs_LIs(hi) = xtr_LIs({l,l′,l″}),
139(c)vii.            len trsl − 1 → obs_LIs(hj) = xtr_LIs({l,l′,l‴}),
139(c)vii.            _ → let (l⁗,l′′′′′)=trsl(i) in obs_LIs(hi)=xtr_LIs({l,l′,l″,l⁗,l′′′′′}) end
139(c)vii.         end end ∧
139(c)vii.       ∀ i:Nat•i ∈ inds phls ⇒
139(c)vii.          let (p,l,h)=phls(i) in obs_HIs(l)=xtr_HIs({p,h}) ∧
139(c)vii.          obs_LIs(p) = {obs_LI(l)} end
```

199

```
type
139d.  PZ
value
139d.  obs_HI: PZ → HI

       xtr_Hs: TRN → H-set
       xtr_Hs(_,phls) ≡ {pz,h|(pz,l,h):(PZ×L×H)•(pz,l,h)∈ elems phls}
       xtr_Ls: TRN → L-set
       xtr_Ls(trls,phls) ≡
            {l,l′|l,l′:L•(l,l′)∈ elems trls} ∪ {l|(pz,l,h):(PZ×L×H)•(pz,l,h)∈ elems phls}

       xtr_HIs: TRN → HI-set,   xtr_HIs(trn) ≡ {obs_HI(h)|h:(H|PZ)•h ∈ xtr_Hs(trn)}
       xtr_LIs: TRN → LI-set,    xtr_LIs(trn) ≡ {obs_LI(l)|l:L•l ∈ xtr_Ls(trn)}
       xtr_HIs: H-set → HI-set,  xtr_HIs(hs) = {obs_LI(h)|h:H•h ∈ hs}
       xtr_LIs: L-set → LI-set,   xtr_LIs(ls) = {obs_LI(l)|l:L•l ∈ ls}
```

Abstraction: From Concrete Toll Road Nets to Abstract Nets 200

140. From concrete toll road nets, trn:TRN, one can abstract the nets, n:N, of Items 1–11.
    a) the abstract net contains the hubs of the concrete net,
    b) and the links likewise.

```
value
140.  abs_N: TRN → N
140.  abs_N(trn) as n
140a.     obs_Hs(n) = xtr_Hs(trn) ∧
140b.     obs_Ls(n) = xtr_Ls(trn)
```

Theorem 201

141. One can prove the following theorem: If trn satisfies wf_TRN(trn) then abs_N(trn) satisfies Axioms 7–8 (Page 14).

```
141. ∀ trn:TRN • wf_TRN(trn) ⊨
             abs_N(trn) satisfies axioms 7.–10.
```

4.2.3 Determination 202

Definition: Determination. *Domain determination is a domain requirements facet[259]. It is an operation performed on a domain description[243] cum requirements prescription[615]. Any nondeterminism[482] expressed by either of these specifications which is not desirable for some required software design must be made deterministic (by this requirements engineer[612] performed operation).*

Example 203 We shall focus on making more specific the rather generically defined nets, hubs and links. There are no traffic signals within the toll road net and pairs of toll road links are "one way, opposite direction" links.

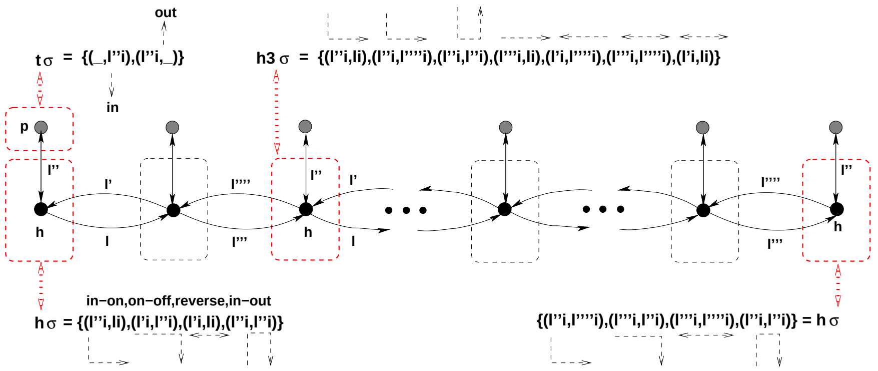
out
tσ = {(_,l''i),(l''i,_)}
in
h3 σ = {(l''i,li),(l''i,l''''i),(l''i,l''i),(l'''i,li),(l'i,l''''i),(l'''i,l''''i),(l'i,li)}
p
l''
l'
l''''
l''
l'
h
l
l'''
h
l
l''''
l''
h
l'''
in−on,on−off,reverse,in−out
hσ = {(l''i,li),(l'i,l''i),(l'i,li),(l''i,l''i)}
{(l''i,l''''i),(l'''i,l''i),(l'''i,l''''i),(l''i,l''i)} = hσ

Figure 4: Four example hub states: plaza, end hubs, "middle" hub

204

142. Pairs of toll road links, l, l', connecting adjacent hubs hj, hk, of identifiers hj_i, hk_i, respectively, always and only allow traffic in opposite directions, that is, are always in respective states $\{(hj_i, hk_i)\}$ and $\{(hk_i, hj_i)\}$.

143. Hub, h, states, $h\sigma$, are constant and allow traffic onto connected links not closed for traffic in directions from hub h.

144. Plazas allow traffic only onto connected plaza to hub links of the toll road net. (Whatever other links, "outside" the toll road net, the plazas may be connected to is covered in the last line of the axiom below.)

205

```
axiom
  ∀ (trls,phls):TRN •
     ∀ i:Nat • i ∈ inds trls
        let (l,l′) = trls(i), (p,l″,h) = phls(i) in
        case i of
           1 → obs_HΣ(h) = {(obs_LI(l″),obs_LI(l)),
                            (obs_LI(l′),obs_LI(l″)),(obs_LI(l′),obs_LI(l)),
                            (obs_LI(l″),obs_LI(l″))},
           _ → let (l‴,l⁗) = trls(i−1) in
                 obs_HΣ(h) = {(obs_LI(l″),obs_LI(l)),
                             (obs_LI(l″),obs_LI(l⁗)),(obs_LI(l″),obs_LI(l″)),
                             (obs_LI(l″),obs_LI(l″)),(obs_LI(l‴),obs_LI(l)),
                             (obs_LI(l′),obs_LI(l⁗)),(obs_LI(l‴),obs_LI(l⁗)),
                             (obs_LI(l′),obs_LI(l))} end end end ∧
     let (l‴,l⁗) = trls(len trsl), (p,l″,h) = phls(1 + len trsl) in
     obs_HΣ(h) = {(obs_LI(l″),obs_LI(l⁗)),
                  (obs_LI(l‴),obs_LI(l″)),(obs_LI(l‴),obs_LI(l⁗)),
                  (obs_LI(l″),obs_LI(l″))} end ∧
     ∀ (p,l″,_):(PZ×L×H)•(p,l″,_) ∈ elems phls ⇒
        let lis = obs_LIs(p) assert: obs_LI(l″) ∈ lis in
        obs_HΣ(p) = {(li,obs_LI(l″)),(obs_LI(l″),li)|li:LI•li ∈ lis} end
```

206

In the last line of the wellformedness axiom above we express that the plaza maybe connected to many links not in the toll road net and that the plaza is open for all traffic from these into the net (via l″), from l″ to these and that traffic may even reverse at the plazas, that is, decide to not enter the toll road net after having just visited the plaza.

4.2.4 Extension 207

Definition: Extension. *Domain extension is a domain requirements facet[259]. It is an operation performed on a domain description[243] or a requirements prescription[615]. It effectively extends a domain description[243] by entities, functions, events and/or behaviours conceptually possible, but not necessarily humanly or technologically feasible in the domain (as it was).*

208 Figure 5 on the facing page abstracts some of the extensions to nets: the plaza entry and exit booths.

209 The following is a prolonged example. It contains three kinds of formalisations: a `RAISE/CSP` model, a `Duration Calculus` model [235, 181] and a `Timed Automata` model [5, 181]. The narrative for all three models are given when narrating the `RAISE/CSP` model.

Intuition 210 A toll road system is delimited by toll plazas with entry and exit booths with their gates. To get access, from outside, to the roads within the toll road system, a car must pass through an entry booth and its entry gate. To leave the roads within the toll road system a car must pass through an exit booth and its exit gate. Cars collect tickets upon entry and return these tickets upon exit and pay a fee for having driven on the toll roads. The gates help ensure that cars have collected tickets and have paid their dues.

211

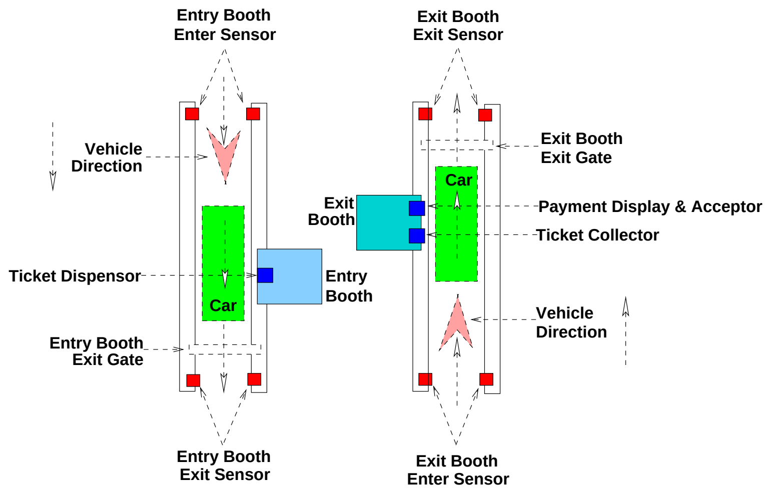

Figure 5: Entry and Exit Tool Booths

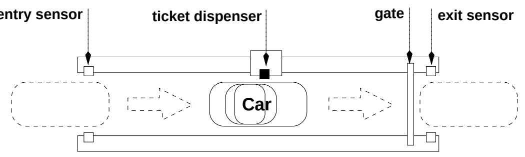

Figure 6: A toll plaza entry booth

Descriptions 212

• A RAISE/CSP Model

We use the CSP property [32, 131] of RSL.

Toll Booth Plazas With respect to toll road systems we focus on just their plazas: that is, where cars enter and leave the systems. The below description is grossly simplified: instead of plazas having one or more entry and one or more exit booths (both with gates), we just assume one (pair: booth/gate) of each. 213

145. A toll plaza consists of a one pair of an entry booth and and entry gate and one pair of an exit booth and an exit gate.
146. Entry booths consist of an entry sensor, a ticket dispenser and an exit sensor.
147. Exit booths consist of an entry sensor, a ticket collector, a payment display and a payment component.

type
145. PZ = (EB×G) × (XB×G)
146. EB = ...
147. XB = ...

214 **Cars** :

148. There are vehicles.
149. Vehicles have unique vehicle identifications.

type
148. V
149. VId
value
149. obs_VId: V → VId
axiom
149. $\forall$ v,v′:V • v≠v′ ⇒ obs_VId(v) ≠ obs_VId(v′)

215 **Entry Booths** :
The description now given is an idealisation. It assumes that everything works: that the vehicles behave as expected and that the electro-mechanics of booths and gates do likewise.

150. An entry_sensor registers whether a car is entering the entry booth or not,
    a) that is, for the duration of the car passing the entry_sensor that sensor senses the car identification cid
    b) otherwise it senses "nothing".

216

151. A ticket_dispenser
    a) either holds a ticket or does not hold a ticket, i.e., no_ticket;
    b) normally it does not hold a ticket;
    c) the ticket_dispenser holds a ticket soon after a car has passed the entry_sensor;
    d) the passing car collects the ticket –
    e) after which the ticket_dispenser no longer holds a ticket.
152. An exit_sensor
    a) registers the identification of a car leaving the toll booth
    b) otherwise it senses "nothing".

217 **Gates** :

153. A gate
    a) is either closed or open;
    b) it is normally closed;
    c) if a car is entering it is secured set to close (as a security measure);
    d) once a car has collected a ticket it is set to open;
    e) and once a car has passed the exit_sensor it is again set to close.

218

The Entry Plaza System :

```
type
    C, CI
    G = open | close
    TK == Ticket | no_ticket
value
    obs_CI: (C|Ticket) → CI
channel
    entry_sensor:CI
    ticket_dispenser:Ticket
    exit_sensor:CI
    gate_ch:G
value
    vs:V-set
    eb:EB,xb:XB,eg,xg:G
```

219

```
    system: G × EB × V-set × XB × G
    system(eg,eb,vs,xb,xg) ≡
        ‖{car(obs_CI(c),c)|c:C•c ∈ cs} ‖ entry_booth(eb) ‖ entry_gate(eg) ‖ ...

    car: CI × C → out entry_sensor,exit_sensor
                  in ticket_dispenser  Unit
    car(ci,c) ≡
         entry_sensor ! ci ;
         let ticket = ticket_dispenser ? assert: ticket ≠ no_ticket in
         ticket_dispenser ! no_ticket ;
         exit_sensor ! ci ;
         car(add(ticket,c)) end
```

220

```
    entry_booth: Unit → in entry_sensor, exit_sensor
                        out ticket_dispenser
                        out gate_ch  Unit
    entry_booth(b) ≡
         gate_ch ! close ;
         let ci = entry_sensor ? in
         ticket_dispenser ! make_ticket(cid) ;
         let res = ticket_dispenser ? in assert: res = no_ticket ;
         gate_ch ! open ;
         let ci′ = exit_sensor ? in assert: ci′ = ci ;
         gate_ch ! close ;
         entry_booth(add_Ticket(ticket,b)) end end end
```

221

```
    entry_gate: G → in gate  Unit
    entry_gate(g) ≡
         case gate_ch ? of
              close → exit_gate(close) assert: g = open,
              open → exit_gate(open) assert: g = close
```

```
         end

    add_Ticket: Ticket × C ~→ C
         pre add_Ticket(t,c): ∼has_Ticket(c)
         post: add_Ticket(t,c): has_Ticket(c)
```

222

```
    has_Ticket: (C|B) → Bool

    obs_Ticket: (C|B) ~→ Ticket
         pre obs_Ticket(cb): has_Ticket(cb)

    rem_Ticket: (C ~→ C) | (B ~→ B)
         pre rem_Ticket(cb): has_Ticket(cb)
         post rem_Ticket(cb): ∼has_Ticket(cb)
```

In the next section, "A Duration Calculus Model", we shall start refining the descriptions given above. We do so in order to handle failures of vehicles to behave as expected and of the electro-mechanics of booths and gates.

223

● **A Duration Calculus Model** We use the Duration Calculus [235, 181] extension to RSL. We abstract the channels of the RAISE/CSP model to now be Boolean-valued variables. 224

```
type
     ES = Bool [true=passing, false=not_passing]
     TD = Bool [true=ticket, false=no_ticket]
     G  = Bool [true=open, false=closing⌈⌉closed⌈⌉opening]
     XS = Bool [true=car_has_just_passed, false=car_passing⌈⌉no-one_passing]
variable
     entry_sensor:ES := false ;
     ticket_dispenser:TD := false ;
     gate:G := false ;
     exit_sensor:XS := false ;
```

225

154. No matter its position, the gate must be closed within no more than δ_{eg} time units after the entry_sensor has registered that a car is entering the toll booth.
155. A ticket must be in the ticket_dispenser within δ_{et} time units after the entry_sensor has registered that a car is entering the toll booth.
156. The ticket is in the ticket_dispenser at most δ_{tdc} time units
157. The gate must be open within δ_{go} time units after a ticket has been collected.
158. The exit sensor is registering (i.e., is on) the identification of exiting cars and is not registering anything when no car is passing (i.e., is off).

226

154. $\sim(\lceil \text{entry\_sensor} \rceil \,;\, (\ell = \delta_{eg} \wedge \lceil \text{gate} \rceil))$
155. $\sim(\lceil \text{entry\_sensor} \rceil \,;\, (\ell = \delta_{et} \wedge \lceil \sim\text{ticket\_dispenser} \rceil))$
156. $\Box(\lceil \sim\text{ticket\_dispenser} \rceil \Rightarrow \ell < \delta_{tdc})$
157. $\sim(\lceil \text{ticket\_dispenser} \rceil \,;\, (\lceil \sim\text{ticket\_dispenser} \wedge \sim\text{gate} \rceil \wedge \ell \geq \delta_{go}))$
158. $\Box(\lceil \text{gate=closing} \rceil \Rightarrow \lceil \sim \text{exit\_sensor} \rceil)$

227

• A `Timed Automata` Model A timed automaton [5, 181] for a configuration of an entry gate, its entry booth and a car is shown in Fig. 7. Figure 8 on the following page shows the a car, an exit booth and its exit gate interactions. They are more-or-less "derived" from the example of Sect. 7.5 of [5, Alur & Dill, 1994] (Pages 42–45). The right half of the car timed automaton of Fig. 7 is to be thought of as the same as the left half of the car timed automaton of Fig. 8 on the following page, cf. the vertical dotted ($\vdots$) line. 228

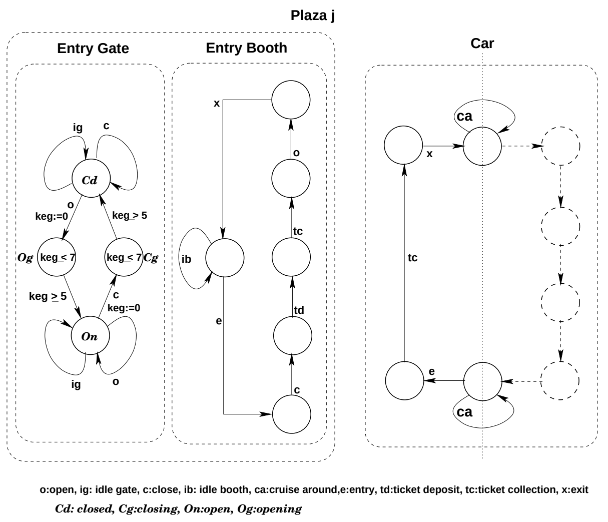
Plaza j
Entry Gate
Entry Booth
Car
ig
c
Cd
o
keg:=0
keg > 5
Og
keg < 7
keg < 7
Cg
keg > 5
c
keg:=0
On
ig
o
x
o
tc
ib
td
e
c
ca
x
tc
e
ca
o:open, ig: idle gate, c:close, ib: idle booth, ca:cruise around,e:entry, td:ticket deposit, tc:ticket collection, x:exit
Cd: closed, Cg:closing, On:open, Og:opening

Figure 7: A `timed automata` model of gate, entry booth and car interactions

229

value
eg,xg:G, eb:EB, xb:XB, vs:V-**set**

System: G×EV×V-**set**×XB×G → **Unit**
System(eg,eb,vs,xb,xg) ≡
Entry_Gate(eg) ∥ Entry_Booth(eb) ∥
∥{Car(obs_CId(c),c)|ci:C,v:C•c ∈ cs} ∥
Exit_Booth(xb) ∥ Exit_Gate(xg)

230

4.2.5 Fitting 231

Definition: Fitting. *By domain requirements fitting we understand an operation which takes n domain requirements prescriptions, d_{r_i} ($i = \{1..n\}$), claimed to share m independent sets of tightly related*

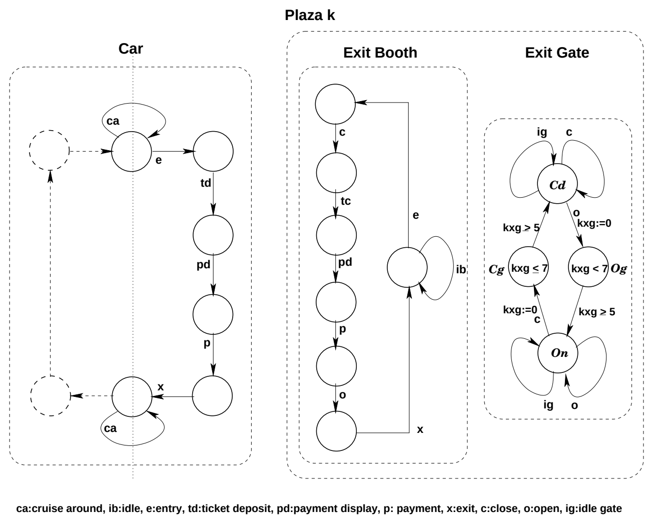
Plaza k
Car
Exit Booth
Exit Gate
ca
e
td
pd
p
x
ca
c
tc
pd
p
o
e
ib
x
ig
c
Cd
o
kxg:=0
kxg > 5
Cg
kxg < 7
kxg < 7
Og
kxg:=0
c
kxg > 5
On
ig
o
ca:cruise around, ib:idle, e:entry, td:ticket deposit, pd:payment display, p: payment, x:exit, c:close, o:open, ig:idle gate

Figure 8: A `timed automata` model of car, exit booth and gate interactions

sets of simple entities, actions, events and/or behaviours and map these into $n{+}m$ domain requirements prescriptions, δ_{r_j} ($j = \{1..n{+}m\}$), where m of these, $\delta_{r_{n+k}}$ ($k = \{1..m\}$) capture the m shared phenomena and concepts and the other n prescriptions, δ_{r_ℓ} ($\ell = \{1..n\}$), are like the n "input" domain requirements prescriptions, d_{r_i} ($i = \{1..n\}$), except that they now,(instead of the "more-or-less" shared prescriptions, that are now consolidated in $\delta_{r_{n+k}}$)prescribe interfaces between δ_{r_i} and $\delta_{r_{n+k}}$ for $i : \{1..n\}$.

Examples 232

TO BE WRITTEN

234

4.3 Interface Requirements 233

Definition: Interface Requirements. *Interface requirements are those requirements[605] which can on be expressed using professional terms from both the domain[239] and the machine[436]. Thus, by interface requirements we understand the expression of expectations as to which software-software, or software-hardware interface[393] places (i.e., channel[110]s), input[382]s and output[502]s (including the semiotics[658] of these input/outputs) there shall be in some contemplated computing system[151]. Interface requirements can often, usefully, be classified in terms of shared data initialisation requirements[671], shared data refreshment requirements[673], computational data+control requirements[146], man-machine dialogue requirements[447], man-machine physiological requirements[448] and machine-machine dialogue requirements[437]. Interface requirements constitute one requirements facet[285]. Other requirements facets are: business process reengineering[101], domain requirements[258] and machine requirements[438].*

4.3.1 But First: On Shared Phenomena and Concepts 235

Definition: Shared Phenomenon or Concept. *A shared phenomenon (or concept) is a phenomenon (respectively a concept) which is present in some domain[239] (say in the form of facts, knowledge[407] or information[373]) and which is also represented in the machine[436] (say in the form of some entity[272], simple, action, event or behaviour). A phenomenon of a domain, when shared, becomes a concept of the machine.*
We shall give some examples – but they are just illustrative. Proper narration and formalisation is left to the reader !

4.3.2 Shared Simple Entities 236

Definition: Shared Simple Entity. *By a shared simple entity we mean a simple entitywhich both occurs in the domain[239] (as a phenomenon or a concept) and in themachine[436]. Simple entities that are shared between the domain and the machine must initially be input to the machine. Dynamically arising simple entities must likewise be inputand all such machine entities must have their attributes updated, when need arise. Requirements for shared simple entitiesthus entail requirements for their representation and for their human/machine and/or machine/machine transfer dialogue.*

Example 237 Main shared entities are those of hubs and links. Representations of hubs and links "within" the machine necessarily abstracts many of the properties of hubs and links; some (such) attributes may not be represented altogether.

As for human input, some man/machine dialogue based around a set of visual display unit screens with fields for the input of hub, respectively link attributes can then be devised. Etc.

4.3.3 Shared Actions 238

Definition: Shared Action. *By a shared action we mean an action that can only be partly computed by the machine[436]. That is, the machine[436], in order to complete an action, may have to inquire with the domain[239] (in order, say, to extract some measurable, time-varying simple entity attribute value) in order to proceed in its computation.*

Example 239 In order for a car **driver** to leave an **exit toll booth** the following component actions must take place: (a) the **driver** inserts the electronic pass into the exit toll booth; (b) the **exit toll booth** scans and accepts the ticket andcalculates the fee for the car journey from entry booth via the toll road net to the exit booth; (c) **exit toll booth** alerts the driver as to the cost and is requested to pay this amount; (d) once the **driver** has paid (e) the **exit booth toll** gate is raised. Actions (a,d) are **driver** actions, (b,c,e) are **machine** actions.

4.3.4 Shared Events 240

Definition: Shared Event. *By a shared event we mean an event whose occurrence in the domain[239] need be communicated to the machine[436] and, vice-versa, an event whose occurrence in the machine[436] need be communicated to the domain[239].*

Examples 241 The arrival of a car at a toll plaza entry booth is an event that must be communicated to the machine so that the entry booth may issue a proper pass (ticket). Similarly for the arrival of a car at a toll plaza exit booth is an event that must be communicated to the machine so that the machine may request the return of the pass and compute the fee. The end of that computation is an event that is communicated to the driver (in the domain) requesting that person to pay a certain fee after which the exit gate is opened.

4.3.5 Shared Behaviours 242

Definition: Shared Behaviour. *By a shared behaviour we mean a behaviour many of whose actions and events occur both in the domain[239] and in the machine[436] (in some encoded form, and in the same squence).*

Example 243 A typical toll road net use behaviour is as follows: Entry at some toll plaza: receipt of electronic ticket, placement of ticket in special ticket "pocket" in front window, the raising of the entry booth toll gate; drive up to [first] toll road hub (with electronic registration of time of occurrence), drive down a selected link (with electronic registration of time of occurrence of entry to and exit from link), then a repeated number of zero, one or more toll road hub and link visits – some of which may be "repeats" – ending with a drive down from a toll road hub to a toll plaza with the return of the electronic ticket, etc. – cf. Sect, 4.3.4.

4.4 Machine Requirements 244

Definition: Machine Requirements. *Machine requirements are those requirements[605] which, in principle, can be expressed without using professional domain terms (for which these requirements are established).*
Thus, by *machine*[436] *requirements*[605], we understand *requirements*[605] put specifically to, i.e., expected specifically from, the *machine*[436]. We normally analyse machine requirements into *performance requirements*[521], *dependability requirements*[218], *maintenance requirements*[443], *platform requirements*[527] and *documentation requirements*[238].

4.4.1 An Enumeration of Classes of Machine Requirements 245

We shall in these lecture notes not go into any detail about machine requirements. But we shall classify machine requirements into a long list of specific kinds of machine requirements.

- Performance
  - Storage
  - Time
  - Software Size
- Dependability
  - Accessability
  - Availability
  - Reliability
  - Robustness
  - Safety
  - Security
- Maintenance
  - Adaptive
  - Corrective
  - Perfective
  - Preventive
- Platforms
  - Development
  - Demonstration
  - Execution
  - Maintenance
- Documentation
- Other

5 Conclusion 246

We discuss a number of issues.

5.1 What Have We Omitted

Our coverage of domain and requirements engineering has focused on modelling techniques for domain and requirements facets. We have omitted the important software engineering tasks of **stakeholder identification and liaison**, **domain** and, to some extents also **requirements**, especially **goal acquisition and analysis**, **terminologisation**, and techniques for **domain and requirements and goal validation and [goal] verification** ($\mathcal{D}, \mathcal{R} \models \mathcal{G}$).

We refer, instead, to [32, Vol.3, Part IV (Chaps. 9, 12–14) and Part V (Chaps. 18, 20–23)].

5.2 Domain Descriptions Are Not Normative 247

A description of, for example, "the" domain of the *New York Stock Exchange* would describe the set of rules and regulations governing the submission of sell offers and buy bids as well as rules and regulations for clearing ('matching') sell offers and buy bids. These rules and regulations appears to be quite different from those of the *Tokyo Stock Exchange* [217]. A normative description of stock exchanges would abstract these rules so as to be rather un-informative. And, anyway, rules and regulations changes and business process re-engineering changes entities, actions, events and behaviours. For any given software development one may thus have to rewrite parts of existing domain descriptions, or construct an entirely new such description.

5.3 "Requirements Always Change" 248

This claim is often used as a hidden excuse for not doing a proper, professional job of requirements prescription, let alone "deriving" them, as we advocate, from domain descriptions. Instead we now make the following counterclaims [1] "domains are far more stable than requirements" and [2] "requirements changes arise more as a result of business process re-engineering than as a result of changing stakeholder ideas". 249

Closer studies of a number of domain descriptions, for example of a *financial service industry*, reveals that the domain in terms of which an "ever expanding" variety of financial products are offered, are, in effect, based on a small set of very basic domain functions which have been offered for well-nigh centuries !

We thus claim that thoroughly developed domain descriptions and thoroughly "derived" requirements prescriptions tend to stabilise the requirements re-design, but never alleviate it.

5.4 What Can Be Described and Prescribed 250

The issue of *"what can be described"* has been a constant challenge to philosophers. In [204, 1919] Bertrand Russell covers his first *Theory of Descriptions*, and in [203, Philosophy of Mathematics] a revision, as *The Philosophy of Logical Atomism*. The issue is not that straightforward. In [40, 41] we try to broach the topic from the point of view of the kind of domain engineering presented in this paper.

Our approach is simple; perhaps too simple ! We can describe what can be observed. We do so, first by 251
postulating types of observable phenomena and of derived concepts; then by the introduction of *observer* functions and by axioms over these, that is, over values of postulated types and observers. To this we add defined functions; usually described by pre/post-conditions. The narratives refer to the "real" phenomena whereas the formalisations refer to related phenomenological concepts. The narrative/formalisation problem is that one can 'describe' phenomena without always knowing how to formalise them.

5.5 What Have We Achieved – and What Not 252

Section 1.2.3 made some claims. We think we have substantiated them all, albeit ever so briefly.

Each of the domain facets (intrinsics, support technologies, rules and regulations, scripts [licenses and contracts], management and organisation and human behaviour) and each of the requirements facets (projection, instantiation, determination, extension and fitting) provide rich grounds for both specification methodology studies and and for more theoretical studies [35, ICTAC 2007].

5.6 Relation to Other Work 253

The most obvious 'other' work is that of [139, Problem Frames]. In [139] Jackson, like is done here, departs radically from conventional requirements engineering. In his approach understandings of the domain, the requirements and possible software designs are arrived at, not hierarchically, but in parallel, interacting streams of decomposition. Thus the 'Problem Frame' development approach iterates between concerns of 254
domains, requirements and software design. "Ideally" our approach pursues domain engineering prior to requirements engineering, and, the latter, prior to software design. But see next. 255

The recent book [148, Axel van Lamsweerde] appears to represent the most definitive work on Requirements Engineering today. Much of its requirements and goal acquisition and analysis techniques carries over to main aspects of domain acquisition and analysis techniques and the goal-related techniques of [148] apply to determining which projections, instantiation, determination and extension operations to perform on domain descriptions.

5.7 "Ideal" Versus Real Developments 256

The term 'ideal' has been used in connection with 'ideal development' from domain to requirements. We now discuss that usage. Ideally software development could proceed from developing domain descriptions via "deriving" requirements prescriptions to software design, each phase involving extensive formal specifications, verifications (formal testing, model checking and theorem proving) and validation. 257

More realistically less comprehensive domain description development (D) may alternate with both requirements development (R) work and with software design (S) – in some controlled, contained iterated and "spiralling" manner and such that it is at all times clear which development step is what: $\mathcal{D}$, $\mathcal{R}$ or $\mathcal{S}$!

5.8 Description Languages 258

We have used the RSL specification language, [110, 32], for the formalisations of this report, but any of the model-oriented approaches and languages offered by Alloy [137], B, Event B [3], RAISE [112], VDM [107] and Z [233], should work as well. 259

No single one of the above-mentioned formal specification languages, however, suffices. Often one has to carefully combine the above with elements of Petri Nets [199], CSP [128], MSC [136], Statecharts [120], and/or some temporal logic, for example either DC [235] or TLA+ [147]. Research into how such diverse textual and diagrammatic languages can be combined is ongoing [9].

5.9 Entailments 260

$\mathcal{D}, \mathcal{R} \models \mathcal{G}$[*] From the $\mathcal{D}$omain and the $\mathcal{R}$equirements we can reason that the $\mathcal{G}$oals are met.

$\mathcal{D}, \mathcal{S} \models \mathcal{R}$[*] In a proof of correctness of $\mathcal{S}$oftware design with respect to $\mathcal{R}$equirements prescriptions one often has to refer to assumptions about the $\mathcal{D}$omain. [*] Formalising our understandings of the $\mathcal{D}$omain, the $\mathcal{R}$equirements and the $\mathcal{S}$oftware design enables proofs that the software is right and the formalisation of the "derivation" of $\mathcal{R}$equirements from $\mathcal{D}$omain specifications help ensure that it is the right software [58].

5.10 Domain Versus Ontology Engineering 261

In the information science community an ontology is a "formal, explicit specification of a shared conceptualisation". Most of the information science ontology work seems aimed primarily at axiomatisations of properties of entities. Apart from that there are many issues of "ontological engineering" that are similar to the triptych kind of domain engineering; but then, we claim, that domain engineering goes well beyond ontological engineering and makes free use of whatever formal specification languages are needed, cf. Sect. 6.1.

6 Bibliographical Notes 262

6.1 Description Languages

Besides using as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements, we have "paired" such narrative elements with corresponding enumerated clauses of a formal specification language. We have been using the `RAISE` Specification Language, `RSL`, [112], in our formal texts. But any of the model-oriented approaches and languages offered by

- `Alloy` [137],
- `CafeOBJ` [109],
- `Event B` [3],
- `VDM` [107] and
- `Z` [233],

should work as well. 263

No single one of the above-mentioned formal specification languages, however, suffices. Often one has to carefully combine the above with elements of

- `Petri Nets` [199],
- `CSP: Communicating Sequential Processes` [128],
- `MSC: Message Sequence Charts` [136],
- `Statecharts` [120],
- and some temporal logic, for example
  - `DC: Duration Calculus` [235]
  - or `TLA+` [147].

Research into how such diverse textual and diagrammatic languages can be meaningfully and proof-theoretically combined is ongoing [9]. And even then !

6.2 References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass., USA, 1996. 2nd edition.

[2] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.

[3] 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.

[4] J.-R. Abrial and L. Mussat. *Event B Reference Manual (Editor: Thierry Lecomte)*, June 2001. Report of EU IST Project Matisse IST-1999-11435.

[5] 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).

[6] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS. Springer–Verlag, 1998.

[7] D. Andrews and W. Henhapl. Pascal. In *[53]*, chapter 7, pages 175–252. Prentice-Hall, 1982.

[8] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.

[9] K. Araki et al., editors. *IFM 1999–2009: Integrated Formal Methods*, volume 1945, 2335, 2999, 3771, 4591, 5423 (only some are listed) of *Lecture Notes in Computer Science*. Springer, 1999–2009.

[10] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison Wesley, US, 1996.

[11] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Heidelberg, Germany, 1998.

[12] J. W. Backus and P. Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–1, 1963.

[13] H. P. Barendregt. *The Lambda Caculus — Its Syntax and Semantics*. North-Holland Publ.Co., Amsterdam, 1981.

[14] H. P. Barendregt. Introduction to Lambda Calculus. *Niew Archief Voor Wiskunde*, 4:337–372, 1984.

[15] H. P. Barendregt. *The Lambda Calculus*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, revised edition, 1991.

[16] H. Barringer, J. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[17] H. Bekič, P. Lucas, K. Walk, and M. Others. Formal Definition of PL/I, ULD Version I. Technical report, IBM Laboratory, Vienna, 1966.

[18] H. Bekič, P. Lucas, K. Walk, and M. Others. Formal Definition of PL/I, ULD Version II. Technical report, IBM Laboratory, Vienna, 1968.

[19] H. Bekič, P. Lucas, K. Walk, and M. Others. Formal Definition of PL/I, ULD Version III. IBM Laboratory, Vienna, 1969.

[20] C. Berge. *Théorie des Graphes et ses Applications*. Collection Universitaire de Mathematiques. Dunod, Paris, 1958. See [21].

[21] C. Berge. *Graphs*, volume 6 of *Mathematical Library*. North-Holland Publ. Co., second revised edition of part 1 of the 1973 english version edition, 1985. See [20].

[22] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, September 1996.

[23] G. Birtwistle, O.-J.Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA* `begin`. Studentlitteratur, Lund, Sweden, 1974.

[24] D. Bjørner. Programming in the Meta-Language: A Tutorial. In D. Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language, [52]*, LNCS, pages 24–217. Springer–Verlag, 1978.

[25] D. Bjørner. Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In D. Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language, [52]*, LNCS, pages 337–374. Springer–Verlag, 1978.

[26] D. Bjørner. The Vienna Development Method: Software Abstraction and Program Synthesis. In *Mathematical Studies of Information Processing*, volume 75 of *LNCS*. Springer–Verlag, 1979. Proceedings of Conference at Research Institute for Mathematical Sciences (RIMS), University of Kyoto, August 1978.

[27] D. Bjørner, editor. *Abstract Software Specifications*, volume 86 of *LNCS*. Springer, 1980.

[28] D. Bjørner. Application of Formal Models. In *Data Bases*. INFOTECH Proceedings, October 1980.

[29] D. Bjørner. Formalization of Data Base Models. In D. Bjørner, editor, *Abstract Software Specification, [27]*, volume 86 of *LNCS*, pages 144–215. Springer–Verlag, 1980.

[30] D. Bjørner. Domain Modelling: Resource Management Strategics, Tactics & Operations, Decision Support and Algorithmic Software. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millenial Perspectives in Computer Science*, Cornerstones of Computing (Ed.: Richard Bird and Tony Hoare), pages 23–40, Houndmills, Basingstoke, Hampshire, RG21 6XS, UK, 2000. Palgrave (St. Martin's Press). An Oxford University and Microsoft Symposium in Honour of Sir Anthony Hoare, September 13–14, 1999.

[31] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [37, 42].

[32] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; ol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[33] 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. See [38, 43].

[34] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [39, 44].

[35] D. Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[36] 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.

[37] D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.

[38] D. Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.

[39] D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.

[40] D. Bjørner. An Emerging Domain Science – A Rôle for Stanisław Leśniewski's Mereology and Bertrand Russell's Philosophy of Logical Atomism. *Higher-order and Symbolic Computation*, 2009.

[41] D. Bjørner. On Mereologies in Computing Science. In *Festschrift for Tony Hoare*, History of Computing (ed. Bill Roscoe), London, UK, 2009. Springer.

[42] D. Bjørner. **Chinese:** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[43] D. Bjørner. **Chinese:** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[44] D. Bjørner. **Chinese:** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[45] D. Bjørner. Domain Engineering. In *BCS FACS Seminars*, Lecture Notes in Computer Science, the BCS FAC Series (eds. Paul Boca and Jonathan Bowen), pages 1–42, London, UK, 2010. Springer.

[46] D. Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (2), May 2010.

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

[48] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph (# 4), March 2009. This Research Monograph[12] contains the following chapters: [83, 85, 86, 87, 88, 89, 90, 91, 92, 84]. Bjørner will post this 507 page book (with 77 fine photos of "all things Japanese", in full colours, taken by Dines in 2006) to you provided you e-mail your name and address and post international reply postage coupons (http://en.wikipedia.org/wiki/International_reply_coupon) to DINES BJØRNER, FREDSVEJ 11, DK-2840 HOLTE, DENMARK in the total amount of: Denmark 60.50 Kr., Europe 126.00 Kr., elsewhere 209.00 Kr.

[49] D. Bjørner. From Domains to Requirements — On a Triptych of Software Development. *Submitted for publication*, Submitted 8 January, 2010.

[50] 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.

[51] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987*. North-Holland, 1988. 625 pages.

[52] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978. This was the first monograph on $Meta\text{-}IV$. [24, 25, 26].

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

[54] D. Bjørner and H. H. Løvengreen. Formal Semantics of Data Bases. In *8th Int'l. Very Large Data Base Conf.*, Mexico City, Sept. 8-10 1982.

[55] D. Bjørner and H. H. Løvengreen. Formalization of Data Models. In *Formal Specification and Software Development, [53]*, chapter 12, pages 379–442. Prentice-Hall, 1982.

[56] D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer, 1980.

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

[58] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ., USA, 1981.

[59] R. Bruni and J. Meseguer. Generalized Rewrite Theories. In Jos C. M. Baeten and Jan Karel Lenstra and Joachim Parrow and Gerhard J. Woeginger, editor, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2003.

[60] D. Cansell and D. Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003.

[61] D. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier Science Publishers (North-Holland), 1990.

[12]http://www.imm.dtu.dk/ db/jaistmono.pdf

[62] C.C.I.T.T. The Specification of CHILL. Technical Report Recommendation Z200, International Telegraph and Telephone Consultative Committee, Geneva, Switzerland, 1980.

[63] E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications With Objective Caml*. Project Cristal, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France, 2004. Preliminary translation of the book Développement d'applications avec Objective Caml [64].

[64] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. Éditions O'Reilly, Paris, France, Avril 2000. ISBN 2-84177-121-0.

[65] J. Cheng. *A Logic for Partial Functions*. PhD thesis, Department of Computer Science, University of Manchester, 1986. UMCS-86-7-1.

[66] A. Church. *Introduction to Mathematical Logic*. The Princeton University Press, Princeton, New Jersey, USA, 1956. Reprint Edition, October 28, 1996, ISBN 0691029067.

[67] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[68] E. F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, 1970.

[69] G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, Cambridge, UK, 1998. ISBN 0-521-57183-9 (hardcover), 0-521-57681-4 (paperback).

[70] D. Crystal. *The Cambridge Encyclopedia of Language*. Cambridge University Press, 1987, 1988.

[71] O.-J. Dahl, E. Dijkstra, and C. Hoare. *Structured Programming*. Academic Press, 1972.

[72] O.-J. Dahl and C. Hoare. Hierarchical program structures. In *[71]*, pages 197–220. Academic Press, 1972.

[73] O.-J. Dahl and K. Nygaard. SIMULA – an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[74] O. Danvy. A Rational Deconstruction of Landin's SECD Machine. Research RS 03–33, BRICS: Basic Research in Computer Science, Dept. of Comp.Sci., University of Århus, Ny Munkegade, Bldg. 540, DK-8000 Århus C, Denmark, October 2003. ISSN 0909 0878. E–mail: BRICS@brics.dk.

[75] C. Date. *An Introduction to Database Systems, I*. The Systems Programming Series. Addison Wesley, 1981.

[76] C. Date. *An Introduction to Database Systems, II*. The Systems Programming Series. Addison Wesley, 1983.

[77] C. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley Professional, November 8, 1996. 4th Edition, ISBN: 0201964260.

[78] J. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.

[79] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. Elsevier, 1990.

[80] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing - Vol. 6. World Scientific Publishing Co., Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, July 1998. 196pp, ISBN 981-02-3513-5, US$30.

[81] R. Diaconescu, K. Futatsugi, and K. Ogata. CafeOBJ: Logical Foundations and Methodology. *Computing and Informatics*, 22(1–2), 2003.

[82] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[83] Dines Bjørner. *[48] Chap. 1: On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38. JAIST Press, March 2009.

[84] Dines Bjørner. *[48] Chap. 10: Towards a Family of Script Languages – – Licenses and Contracts – Incomplete Sketch*, pages 283–328. JAIST Press, March 2009.

[85] Dines Bjørner. *[48] Chap. 2: Possible Collaborative Domain Projects – A Management Brief*, pages 39–56. JAIST Press, March 2009.

[86] Dines Bjørner. *[48] Chap. 3: The Rôle of Domain Engineering in Software Development*, pages 57–72. JAIST Press, March 2009.

[87] Dines Bjørner. *[48] Chap. 4: Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106. JAIST Press, March 2009.

[88] Dines Bjørner. *[48] Chap. 5: The Triptych Process Model – Process Assessment and Improvement*, pages 107–138. JAIST Press, March 2009.

[89] Dines Bjørner. *[48] Chap. 6: Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm*, pages 139–175. JAIST Press, March 2009.

[90] Dines Bjørner. *[48] Chap. 7: Documents – A Rough Sketch Domain Analysis*, pages 179–200. JAIST Press, March 2009.

[91] Dines Bjørner. *[48] Chap. 8: Public Government – A Rough Sketch Domain Analysis*, pages 201–222. JAIST Press, March 2009.

[92] Dines Bjørner. *[48] Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.

[93] O. Dommergaard. The design of a virtual machine for Ada. In *[27]*, pages 463–605. Springer, 1980.

[94] O. Dommergaard and S. Bodilsen. A formal definition of P-code. Technical report, Dept. of Comp. Sci., Techn. Univ. of Denmark, 1980.

[95] D. J. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 244–261. VDM-Europe, Springer-Verlag, 1990.

[96] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and M. B, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice Hall, 1991.

[97] R. K. Dybvig. *The Scheme Programming Language*. The MIT Press, Cambridge, Mass., USA, 2003. 3rd Edition.

[98] A. Ershov. On the essence of translation. *Computer Software and System Programming*, 3(5):332–346, 1977.

[99] A. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

[100] A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[101] A. Ershov. On Futamura projections. *BIT (Japan)*, 12(14):4–5, 1982. (In Japanese).

[102] A. Ershov. On mixed computation: Informal account of the strict and polyvariant computational schemes. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming. NATO ASI Series F: Computer and System Sciences, vol. 14*, pages 107–120. Springer-Verlag, 1985.

[103] A. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*. Ohmsha Ltd. and Springer-Verlag, 1988.

[104] A. Ershov and V. Grushetsky. An implementation-oriented method for describing algorithmic languages. In B. Gilchrist, editor, *Information Processing 77, Toronto, Canada*, pages 117–122. North-Holland, 1977.

[105] A. Ershov and V. Itkin. Correctness of mixed computation in Algol-like programs. In J. Gruska, editor, *Mathematical Foundations of Computer Science, Tatranská Lomnica, Czechoslovakia. (Lecture Notes in Computer Science, vol. 53)*, pages 59–77. Springer-Verlag, 1977.

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

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

[108] FOLDOC: The free online dictionary of computing. Electronically, on the Web: `http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?ISWIM`, 2004.

[109] 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.

[110] 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.

[111] C. W. George and A. E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003.

[112] 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.

[113] J. Gosling and F. Yellin. *The Java Language Specification*. Addison-Wesley & Sun Microsystems. ACM Press Books, 1996. 864 pp, ISBN 0-10-63451-1.

[114] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[115] O. Grillmeyer. *Exploring Computer Science with Scheme*. Springer-Verlag, New York, USA, 1998.

[116] C. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.

[117] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[118] P. Haff, editor. *The Formal Definition of CHILL*. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1981.

[119] M. R. Hansen and H. Rischel. *Functional Programming in Standard ML*. Addison Wesley, 1997.

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

[121] D. Harel and R. Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[122] F. Harrary. *Graph Theory*. Addison Wesley Publishing Co., 1972.

[123] E. Hehner. *The Logic of Programming*. Prentice-Hall, 1984.

[124] E. Hehner. *a Practical Theory of Programming*. Springer-Verlag, 2nd edition, 1993. On the net: http://www.cs.toronto.edu/˜hehner/aPToP/.

[125] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Microsoft •net Development Series. Addison-Wesley, 75 Arlington Street, Suite 300, Boston, MA 02116, USA, (617) 848-6000, 30 October 2003. 672 page, ISBN 0321154916.

[126] M. C. Henson, S. Reeves, and J. P. Bowen. Z Logic and its Consequences. *Computing and Informatics*, 22(1–2), 2003.

[127] J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, Ithaca, N.Y., USA, June 1962. ASIN 0801401879.

[128] 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).

[129] C. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

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

[131] T. Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.-com/cspbook.pdf`, 2004. Second edition of [130]. See also `http://www.usingcsp.com/`.

[132] C. J. Hogger. *Essentials of Logic Programming*. Graduate Texts in Computer Science, no.1, 310 pages. Clarendon Press, December 1990. .

[133] IEEE CS. IEEE Standard Glossay of Software Engineering Terminology, 1990. IEEE Std.610.12.

[134] Inmos Ltd. Specification of instruction set & Specification of floating point unit instructions. In *Transputer Instruction Set – A compiler writer's guide*, pages 127–161. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1988.

[135] B. B. S. Institution. Specification for computer programming language Pascal. Technical Report BS6192, BSI, 1982.

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

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

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

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

[140] K. Jensen and N. Wirth. *Pascal User Manual and Report*, volume 18 of *LNCS*. Springer–Verlag, 1976.

[141] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. C.A.R.Hoare Series in Computer Science. Prentice Hall International, 1993.

[142] B. Kernighan and D. Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1989.

[143] D. Knuth. *The Art of Computer Programming, Vol.1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., USA, 1968.

[144] D. Knuth. *The Art of Computer Programming, Vol.2.: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., USA, 1969.

[145] D. Knuth. *The Art of Computer Programming, Vol.3: Searching & Sorting*. Addison-Wesley, Reading, Mass., USA, 1973.

[146] I. Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery (Eds.: J. Worrall and E. G. Zahar)*. Cambridge University Press, The Edinburgh Building, Shaftesbury Road, Cambridge CB2 2RU, England, 2 September 1976. ISBN: 0521290384. Published in 1963-64 in four parts in the British Journal for Philosophy of Science. (Originally Lakatos' name was Imre Lipschitz.).

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

[148] A. Lamsweerde. *Requirements Engineering: from system goals to UML models to software specifications*. Wiley, 2009.

[149] P. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses, 1997. In O. Danvy, editor, ACM SIGPLAN Workshop on Continuations, Number NS-96-13 in BRICS Notes Series, 1997.

[150] P. J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation (in 2 parts). *Communications of the ACM*, 8(2-3):89–101 and 158–165, Feb.-March 1965.

[151] P. J. Landin. A Generalization of Jumps and Labels. Technical report, Univac Sys. Prgr. Res. Grp., N.Y., 1965.

[152] P. J. Landin. Getting Rid of Labels. Technical report, Univac Sys. Prgr. Res. Grp., N.Y., 1965.

[153] J. Lee. *Computer Semantics*. Van Nostrand Reinhold Co., 1972.

[154] J. Lee and W. Delmore. The Vienna Definition Language, a generalization of instruction definitions. In *SIGPLAN Symp. on Programming Language Definitions, San Francisco*, Aug. 1969.

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

[156] X. Leroy and P. Weis. *Manuel de Référence du langage Caml*. InterEditions, Paris, France, 1993. ISBN 2-7296-0492-8.

[157] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley & Sun Microsystems. ACM Press Books, 1996. 496 pp, ISBN 0-10-63452-X.

[158] J. Lipson. *Elements of Algebra and Algebraic Computing*. Addison-Wesley, Reading, Mass., 1981.

[159] W. Little, H. Fowler, J. Coulson, and C. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.

[160] J. Lloyd. *Foundation of Logic Programming*. Springer-Verlag, 1984.

[161] P. Lucas. Formal Semantics of Programming Languages: VDL. *IBM Journal of Devt. and Res.*, 25(5):549–561, 1981.

[162] P. Lucas and K. Walk. On the Formal Description of PL/I. *Annual Review Automatic Programming Part 3*, 6(3), 1969.

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

[164] ANSI X3.23-1974. The Cobol programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1974.

[165] ANSI X3.53-1976. The PL/I programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1976.

[166] ANSI X3.9-1966. The Fortran programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1966.

[167] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*. North-Holland Publ.Co., Amsterdam, 1963.

[168] J. McCarthy. Artificial Intellingence. Electronically, on the Web: `http://www-formal.stanford.edu/jmc/`, 2004.

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

[170] S. Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003.

[171] J. Meseguer. Software Specification and Verification in Rewriting Logic. NATO Advanced Study Institute, 2003.

[172] Microsoft Corporation. *MCAD/MCSD Self-Paced Training Kit: Developing Web Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET*. Microsoft Corporation, Redmond, WA, USA, 2002. 800 pages.

[173] Microsoft Corporation. *MCAD/MCSD Self-Paced Training Kit: Developing Windows-Based Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET*. Microsoft Corporation, Redmond, WA, USA, 2002.

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

[175] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., USA and London, England, 1990.

[176] C. C. Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.

[177] T. Mossakowski, A. E. Haxthausen, D. Sanella, and A. Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003.

[178] J. F. Nilsson. Some Foundational Issues in Ontological Engineering, October 30 – Novewmber 1 2002. Lecture slides for a PhD Course in *Representation Formalisms for Ontologies*, Copenhagen, Denmark.

[179] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[180] Object Management Group. *OMG Unified Modelling Language Specification*. OMG/UML, http://www.omg.org/uml/, version 1.5 edition, March 2003. www.omg.org/cgi-bin/doc?formal/03-03-01.

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

[182] O. Ore. *Graphs and their Uses* . The Mathematical Association of America, 1963.

[183] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[184] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[185] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[186] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 14(5), May 1972.

[187] D. L. Parnas. *Software Fundamentals: Collected Papers, Eds.: David M. Weiss and Daniel M. Hoffmann*. Addison–Wesley Publ. Co., April 9 2001. 688 pages. ISBN 0201703696. Amazon price (August 2001) US $ 49.95.

[188] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Software Engineering*, 12(2):251–257, February 1986.

[189] D. L. Parnas, P. C. Clements, and D. M. Weiss. Enhancing reusability with information hiding. In *Tutorial: Software Reusability (Ed.: Peter Freeman)*, pages 83–90. IEEE Press, 1986.

[190] L. Paulson. Isabelle: The next 700 theorem provers. In P. Oddifreddi, editor, *Logic in Computer Science*, pages 361–386. Academic Press, 1990.

[191] C. Petzold. *Programming Windows with C# (Core Reference)* . Microsoft Corporation, Redmond, WA, USA, 2001. 1200 pages.

[192] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Comp. Sci. Dept., Aarhus Univ., Denmark; DAIMI-FN-19, 1981. Definitive version of this seminal report is (to be) published in a special issue of the *Journal of Logic and Algebraic Programming* (eds. Jan Bergstra and John Tucker) devoted to a workshop on SOS: Structural Operational Semantics, a Satellite Event of CONCUR 2004, August 30, 2004, London, United Kingdom. Look also for Gordon Plotkin's introductory paper for that issue: The Origins of Structural Operational Semantics.

[193] G. D. Plotkin. Structural operational semantics. Lecture notes, Aarhus University, DAIMI FN-19. Reprinted 1991, 1981. See [195, 194].

[194] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July-December 2004. See [193, 195].

[195] G. D. Plotkin. A structural approach operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July-December 2004. Widely disseminated since 1981 as [193]. See also [194].

[196] B. Randell. On Failures and Faults. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Formal Methods Europe, Springer–Verlag, 2003. Invite Paper.

[197] W. Reisig. On Gurevich's Theorem for Sequential Algorithms. *Acta Informatica*, 2003.

[198] W. Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [60, 81, 177, 111, 170, 126] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

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

[200] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.

[201] J. C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.

[202] A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. Now available on the net: http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/-68b.pdf.

[203] 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.

[204] B. Russell. *Introduction to Mathematical Philosophy*. George Allen and Unwin, London, 1919.

[205] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.

[206] S. Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.

[207] P. Sestoft. *Java Precisely*. The MIT Press, 25 July 2002. 100 pages (sic !), ISBN 0262692767.

[208] P. M. Simons. *Foundations of Logic and Linguistics: Problems and their Solutions*, chapter Leśniewski's Logic and its Relation to Classical and Free Logics. Plenum Press, New York, 1985. Georg Dorn and P. Weingartner (Eds.).

[209] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Jan. 1988.

[210] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

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

[212] J. Srzednicki and Z. Stachniak. *Leśniewksi's Systems Protothetic*. . Dordrecht, 1998.

[213] Staff of Merriam Webster. Online Dictionary: `http://www.m-w.com/home.htm`, 2004. Merriam–Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.

[214] J. Stein, editor. *The Random House American Everyday Disctionary*. Random House, New York, N.Y., USA, 1949, 1961.

[215] B. Stroustrup. *C++ Programming Language*. Addison-Wesley Publishing Company, 1986.

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

[217] T. Tamai. Social Impact of Information System Failures. *Computer, IEEE Computer Society Journal*, 42(6):58–65, June 2009.

[218] R. Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.

[219] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 2nd edition, 29 March 1999. 512 pages, ISBN 0201342758.

[220] D. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In J. Jouannaud, editor, *Functional Programming Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[221] 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.

[222] R. van Glabbeek and P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. Electronically, on the Web: `http://theory.stanford.edu/~rvg/abstraction/abstraction.html`, Centrum voor Wiskunde en Informatica, Postbus 94079, 1090 GB Amsterdam, The Netherlands, January 1996.

[223] A. van Wijngaarden. Report on the algorithmic language Algol 68. *Acta Informatica*, 5:1–236, 1975.

[224] B. Venners. *Inside the Java 2.0 Virtual Machine (Enterprise Computing)*. McGraw-Hill; ISBN: 0071350934, October 1999.

[225] D. Watt, B. Wichmann, and W. Findlay. *Ada: Language and Methodology*. Intl. Ser. in Comp. Sc. Prentice-Hall International, 1986.

[226] P. Weis and X. Leroy. *Le langage Caml*. Dunod, Paris, France, 1999. ISBN 2-10-004383-8, Second edition.

[227] Wikipedia. Polymorphism. In *Internet*. Published: http://en.wikipedia.org/wiki/Polymorphism_(computer_science), 2005.

[228] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.

[229] N. Wirth. The Programming Language PASCAL. *Acta Informatica*, 1(1):35–63, 1971.

[230] N. Wirth. *Systematic Programming*. Prentice-Hall, 1973.

[231] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[232] N. Wirth and C. Hoare. A Contribution to the Development of ALGOL. *Communications of the ACM*, 9(6):413–432, 1966.

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

[234] E. N. Zalta. Logic. In *The Stanford Encyclopedia of Philosophy*. Published: http://plato.stanford.edu/, Winter 2003.

[235] 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.

A An RSL Primer 264

This is an ultra-short introduction to the RAISE Specification Language, RSL. Examples follow and expand on the examples of earlier sections.

A.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

A.1.1 Type Expressions

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

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. 265

Basic Types

```
type
   [1] Bool
   [2] Int
   [3] Nat
   [4] Real
   [5] Char
   [6] Text
```

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″, ...

266

Example 1 .... **Basic Net Attributes:**

- For safe, uncluttered traffic, hubs and links can 'carry' a maximum of vehicles.
- Links have lengths. (We ignore hub (traversal) lengths.)
- One can calculate whether a link is a two-way link.

267

```
type
   MAX = Nat
   LEN = Real
   is_Two_Way_Link = Bool
value
   obs_Max: (H|L) → MAX
```

obs_Len: L $\rightarrow$ LEN
is_two_way_link: L $\rightarrow$ is_Two_Way_Link
is_two_way_link(l) $\equiv \exists$ lσ:LΣ • l$\sigma \in$ obs_HΣ(l)$\wedge$**card** lσ=2

. . . . **End of Example 1**

Composite Types 268

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully "taken apart".

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc. 270

Let A, B and C be any type names or type expressions, then:

Composite Type Expressions

[7] A-**set**
[8] A-**infset**
[9] A $\times$ B $\times$ ... $\times$ C
[10] A*
[11] A$^\omega$
[12] A $\overrightarrow{m}$ B
[13] A $\rightarrow$ 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

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 $\times$ C $\times$ ... $\times$ 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. 272
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.

269

Example 2 . . . . **Composite Net Type Expressions:**

The type clauses of function signatures:

value
  f: A $\rightarrow$ B

often have the type expressions *A* and/or *B* be composite type expressions:

value
  obs_HIs: L $\rightarrow$ HI-**set**
  obs_LIs: H $\rightarrow$ LI-**set**
  obs_HΣ: H $\rightarrow$ HT-**set**
  set_HΣ: H $\times$ HΣ $\rightarrow$ H

Right-hand sides of type definitions often have composite type expressions:

type
  N = H-**set** $\times$ L-**set**
  HT = LI $\times$ HI $\times$ LI
  LT$'$ = HI $\times$ LI $\times$ HI

. . . . **End of Example 2**

A.1.2 Type Definitions 271

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

Type Definition

type
  A = Type_expr
schematic examples:
  A1 = B1-**set**, A2 = B1-**infset**
  A3 = B2 $\times$ C1 $\times$ D1
  B1 = E*, B2 = E$^\omega$
  C1 = F $\overrightarrow{m}$ G
  D1 = H $\rightarrow$ J, D2 = H $\xrightarrow{\sim}$ J
  K = L | M

Example 3 . . . . **Composite Net Types:**

There are many ways in which nets can be concretely modelled:

- **Sorts + Observers + Axioms:** First we show an example of type definitions without right-hand side, that is, of sort definitions.

  From a net one can observe many things.

Of the things we focus on are the hubs and the links.

A net contains two or more hubs and one or more links. Possibly other entities and net attributes may also be observable, but we shall not consider those here.

type
  [sorts] N_α, H, L, HI, LI
value
  obs_Hs: $N_\alpha \to$ H-**set**
  obs_Ls: $N_\alpha \to$ L-**set**
axiom
  $\forall$ n:N_α • **card** obs_Hs(n)>0 $\Rightarrow$ **card** obs_Ls(n)$\geq$1 $\wedge$ ...

273

- **Cartesians + Wellformedness:** A net can be considered as a Cartesian of sets of two or more hubs and sets of one or more links.

type
  [sorts] H, L
  N_β = H-**set** $\times$ L-**set**
value
  wf_N_β: $N_\beta \to$ **Bool**
  wf_N_β(hs,ls) $\equiv$ **card** hs>1 $\Rightarrow$ **card** ls>0 ....
  inject_N_β: $N_\alpha \xrightarrow{\sim} N_\beta$ **pre**: wf_N_β(hs,ls)
  inject_N_β(n_α) $\equiv$ (obs_Hs(n_α),obs_Ls(n_α))

274

- **Cartesians + Maps + Wellformedness:** Or a net can be described

  a as a triple of b-c-d:

  b hubs (modelled as a map from hub identfiers to hubs),

  c links (modelled as a map from link identfiers to links), and

  d a graph from hub h_i identifiers h_{i_i} to maps from identfiers l_{ij_i} of hub h_i connected links l_{ij} to the identfiers h_{j_i} of link connected hubs h_j.

275

type
  [sorts] H, HI, L, LI
  [a] N_γ = HUBS $\times$ LINKS $\times$ GRAPH
  [b] HUBS = HI $\xrightarrow[m]{}$ H
  [c] LINKS = LI $\xrightarrow[m]{}$ L
  [d] GRAPH = HI $\xrightarrow[m]{}$ (LI −m> HI)

– [b,c] *hs:HUBS* and *ls:LINKS* are maps from hub (link) identifiers to hubs (links) where one can still observe these identfiers from these hubs (link).

- Example 12 on page 112 defines the well-formedness predicates for the above map types.

. . . . **End of Example 3**

276

Variety of Type Definitions

[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) |}

277

where a form of [2–3] is provided by combining the types:

Record 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
  $\forall$ a1:A_1, a2:A_2, ..., ai:Ai •
    s_a1(mk_id_1(a1,a2,...,ai))=a1 $\wedge$ s_a2(mk_id_1(a1,a2,...,ai))=a2 $\wedge$
    ... $\wedge$ s_ai(mk_id_1(a1,a2,...,ai))=ai $\wedge$
  $\forall$ a:A • **let** mk_id_1(a1′,a2′,...,ai′) = a **in**
    a1′ = s_a1(a) $\wedge$ a2′ = s_a2(a) $\wedge$ ... $\wedge$ ai′ = s_ai(a) **end**

278

Example 4 . . . . **Net Record Types: Insert Links:**

19. To a net one can insert a new link in either of three ways:

a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;

b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;

c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.

d) From the inserted link one must be able to observe identifier of respective hubs.

20. From a net one can remove a link.[13] The removal command specifies a link identifier.

279

type
```
  19     Insert == Ins(s_ins:Ins)
  19     Ins = 2xHubs | 1x1nH | 2nHs
  19a    2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
  19b    1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
  19c    2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)
  20     Remove == Rmv(s_li:LI)
```
axiom
```
  19d  ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_LIs(l)={hi′,hi″} ∧
       ∀ 1old1newH(hi,l,h):Ins • obs_LIs(l)={hi,obs_HI(h)} ∧
       ∀ 2newH(h′,l,h″):Ins • obs_LIs(l)={obs_HI(h′),obs_HI(h″)}
```

RSL Explanation

- 19: The type clause **type** Ins = 2xHubs | 1x1nH | 2nHs introduces the type name Ins and defines it to be the union (|) type of values of either of three types: 2xHubs, 1x1nH and 2nHs.
  - 19a): The type clause **type** 2xHubs == 2oldH(s_hi1:HI, s_l:L, s_hi2:HI) defines the type 2xHubs to be the type of values of record type 2oldH(s_hi1:HI,s_l:L,s_hi2:HI), that is, Cartesian-like, or "tree"-like values with record (root) name 2oldH and with three sub-values, like branches of a tree, of types HI, L and HI. Given a value, cmd, of type 2xHubs, applying the selectors s_hi1, s_l and s_hi2 to cmd yield the corresponding sub-values.
  - 19b): Reading of this type clause is left as exercise to the reader.
  - 19c): Reading of this type clause is left as exercise to the reader.
  - 19d): The axiom **axiom** has three predicate clauses, one for each category of Insert commands.

[13] – provided that what remains is still a proper net

◊ The first clause: ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_HIs(l) = {hi′, hi″} reads as follows:
  - For all record structures, 2oldH(hi′,l,hi″), that is, values of type Insert (which in this case is the same as of type 2xHubs),
  - that is values which can be expressed as a record with root name 2oldH and with three sub-values ("freely") named hi′, l and hi″
  - (where these are bound to be of type HI, L and HI by the definition of 2xHubs),
  - the two hub identifiers hi′ and hi″ must be different,
  - and the hub identifiers observed from the new link, l, must be the two argument hub identifiers hi′ and hi″.

◊ Reading of the second predicate clause is left as exercise to the reader.

◊ Reading of the third predicate clause is left as exercise to the reader.

The three types 2xHubs, 1x1nH and 2nHs are disjoint: no value in one of them is the same value as in any of the other merely due to the fact that the record names, 2oldH, 1oldH1newH and 2newH, are distinct. This is no matter what the "bodies" of their record structure is, and they are here also distinct: (s_hi1:HI,s_l:L,s_hi2:HI), (s_hi:HI,s_l:L,s_h:H), respectively (s_h1:H,s_l:L,s_h2:H).

- 20; The type clause **type** Remove == Rmv(s_li:LI)
  - (as for Items 19b) and 19c))
  - defines a type of record values, say rmv,
  - with record name Rmv and with a single sub-value, say li of type LI
  - where li can be selected from by rmv selector s_li.

End of RSL Explanation

Example 17 on page 123 presents the semantics functions for *int_Insert* and *int_Remove*.

. . . . **End of Example 4**

Subtypes

280

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:

Subtypes

type
  A = {| b:B • $\mathcal{P}$(b) |}

281

Example 5 .... **Net Subtypes:**
In Example 3 on page 92 we gave three examples. For the first we gave an example, **Sorts + Observers + Axioms**, "purely" in terms of sets, see *Sorts — Abstract Types* below. For the second and third we gave concrete types in terms of Cartesians and Maps. 282

- In the **Sorts + Observers + Axioms** part of Example 3
  - a net was defined as a sort, and so were its hubs, links, hub identifiers and link identifiers;
  - axioms – making use of appropriate observer functions - make up the wellformedness condition on such nets.

  We now redefine this as follows:

283

```
type
   [sorts] N′, H, L, HI, LI
                N = {|n:N′ • wf_N(n)|}
value
   wf_N: N′ → Bool
   wf_N(n) ≡
      ∀ n:N • card obs_Hs(n)≥0 ∧ card obs_Ls(n)≥0 ∧
      axioms 2.–3., 5.–6., and 10., (Page 13)
```

284

- In the **Cartesians + Wellformedness** part of Example 3
  - a net was a Cartesian of a set of hubs and a set of links
  - with the wellformedness that there were at least two hubs and at least one link
  - and that these were connected appropriately (treated as ...).

  We now redefine this as follows:

```
type
   N′ = H-set × L-set
   N = {|n:N′ • wf_N(n)|}
```

285

- In the **Cartesians + Maps + Wellformedness** part of Example 3
  - a net was a triple of hubs, links and a graph,
  - each with their wellformednes predicates.

We now redefine this as follows:

286

```
type
   [sorts] L, H, LI, HI
   N′ = HUBS × LINKS × GRAPH
   N = {|(hs,ls,g):N′ • wf_HUBS(hs)∧wf_LINKS(ls)∧wf_GRAPH(g)(hs,ls)|}
   HUBS′ = HI →m H
   HUBS = {|hs:HUBS′ • wf_HUBS(hs)|}
   LINKS′ = LI → L
   LINKS = {|ls:LINKS′ • wf_LINKS(ls)|}
   GRAPH′ = HI →m (LI →m HI)
   GRAPH = {|g:GRAPH′ • wf_GRAPH(g)|}
value
   wf_GRAPH: GRAPH′ → (HUBS × LINKS) → Bool
   wf_GRAPH(g)(hs,ls) ≡ wf_N(hs,ls,g)
```

Example 12 on page 112 presents a definition of *wf_GRAPH*.
.... **End of Example 5**

Sorts — Abstract Types 287

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

Sorts

```
type
   A, B, ..., C
```

288

Example 6 .... **Net Sorts:**
In formula lines of Examples 3–5 we have indicated those **type** clauses which define *sorts*, by bracketed [sorts] literals.
.... **End of Example 6**

A.2 Concrete RSL Types: Values and Operations 289

A.2.1 Arithmetic

Arithmetic

```
type
   Nat, Int, Real
value
   +,−,∗: Nat×Nat→Nat | Int×Int→Int | Real×Real→Real
   /: Nat×Nat ~→ Nat | Int×Int ~→ Int | Real×Real ~→ Real
   <,≤,=,≠,≥,> (Nat|Int|Real) × (Nat|Int|Real) → Bool
```

294

290

A.2.2 Set Expressions

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

Set Enumerations

$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \subseteq$ **A-set**
$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \subseteq$ **A-infset**

291

Example 7 .... **Set Expressions over Nets:**
We now consider hubs to abstract cities, towns, villages, etcetera. Thus with hubs we can associate sets of citizens.

Let c:C stand for a citizen value c being an element in the type C of all such. Let g:G stand for any (group) of citizens, respectively the type of all such. Let s:S stand for any set of groups, respectively the type of all such. Two otherwise distinct groups are related to one another if they share at least one citizen, the liaisons. A network nw:NW is a set of groups such that for every group in the network one can always find another group with which it shares liaisons. 292
Solely using the set data type and the concept of subtypes, we can model the above:

```
type
   C
   G′ = C-set,  G = {| g:G′ • g≠{} |}
   S = G-set
   L′ = C-set,  L = {| ℓ:L′ • ℓ≠{} |}
   NW′ = S,  NW = {| s:S • wf_S(s) |}
value
   wf_S: S → Bool
   wf_S(s) ≡ ∀ g:G • g ∈ s ⇒ ∃ g′:G •  g′ ∈ s ∧ share(g,g′)
   share: G×G → Bool
   share(g,g′) ≡ g≠g′ ∧ g ∩ g′ ≠ {}
   liaisons: G×G → L
   liaisons(g,g′) = g ∩ g′ pre share(g,g′)
```

296

293

Annotations: L stands for proper liaisons (of at least one liaison). G′, L′ and N′ are the "raw" types which are constrained to G, L and N. {| binding:type_expr • bool_expr |} is the general form of the subtype expression. For G and L we state the constraints "in-line", i.e., as direct part of the subtype expression. For NW we state the constraints by referring to a separately defined predicate. wf_S(s) expresses — through the auxiliary predicate — that s contains at least two groups and that any such two groups share at least one citizen. liaisons is a "truly" auxiliary function in that we have yet to "find an active need" for this function!

The idea is that citizens can be associated with more than one city, town, village, etc. (primary home, summer and/or winter house, working place, etc.). A group is now a set of citizens related by some "interest" (Rotary club membership, political party "grassroots", religion, et.). The reader is invited to define, for example, such functions as:The set of groups (or networks) which are represented in all hubs [or in only one hub]. The set of hubs whose citizens partake in no groups [respectively networks]. The group [network] with the largest coverage in terms of number of hubs in which that group [network] is represented.
.
.... **End of Example 7**

Set Comprehension 295
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.

Set Comprehension

```
type
   A, B
   P = A → Bool
   Q = A ~→ B
value
   comprehend: A-infset × P × Q → B-infset
   comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

Example 8 .... **Set Comprehensions:**
Item 52 on page 23, the *wf_N(hs,ls,g)* wellformedness predicate definition, includes:

```
type
51a.  PLAN = HI ⇀m LHIM
51b.  LHIM = LI ⇀m HI-set
value
52c.   no_junk: PLAN → Bool
52c.   no_junk(plan) ≡ dom plan = ∪{rng(plan(hi))|hi:HI•hi ∈ dom plan}
```

It expresses the distributed union of sets (**rng** *(plan(li))* of hub identifiers (for each of the *hi* indexed maps from (definition set, **dom**) link identiers to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **dom** *plan*).

. . . . **End of Example 8**

A.2.3 Cartesian Expressions 297

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

Cartesian Enumerations

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

298

Example 9 . . . . **Cartesian Net Types:**

So far we have abstracted hubs and links as sorts. That is, we have not defined their types concretely. Instead we have postulated some attributes such as: observable hub identifiers of hubs and sets of observable link identifiers of links connected to hubs. We now claim the following further attributes of hubs and links. 299

- Concrete links have
  - link identifiers,
  - link names – where two or more connected links may have the same link name,
  - two (unordered) hub identifiers,
  - lenghts,
  - locations – where we do not presently defined what we main by locations,
  - etcetera
- Concrete hubs have
  - hub identifiers,
  - unique hub names,
  - a set of one or more observable link identifiers,
  - locations,
  - etcetera.

300

```
type
   LN, HN, LEN, LOC
   cL = LI × LN × (HI × HI) × LOC × ...
   cH = HI × HN × LI-set × LOC × ...
```

. . . . **End of Example 9**

A.2.4 List Expressions 301

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

List Enumerations

$\{\langle\rangle, \langle e\rangle, \ldots, \langle e1,e2,\ldots,en\rangle, \ldots\} \subseteq A^*$
$\{\langle\rangle, \langle e\rangle, \ldots, \langle e1,e2,\ldots,en\rangle, \ldots, \langle e1,e2,\ldots,en,\ldots\ \rangle, \ldots\} \subseteq A^\omega$

$\langle\ a\_i\ ..\ a\_j\ \rangle$

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension 302

The last line below expresses list comprehension.

List Comprehension

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

303

Example 10 . . . . **Routes in Nets:**

- A phenomenological (i.e., a physical) route of a net is a sequence of one or more adjacent links of that net.
- A conceptual route is a sequence of one or more link identifiers.
- An abstract route is a conceptual route

– for which there is a phenomenological route of the net for which the link identifiers of the abstract route map one-to-one onto links of the phenomenological route.

304

```
type
  N, H, L, HI, LI
  PR′ = L*
  PR = {| pr:PR′ • ∃ n:N • wf_PR(pr)(n)|}
  CR = LI*
  AR′ = LI*
  AR = {| ar:AR′ • ∃ n:N • wf_AR(ar)(n) |}
value
  wf_PR: PR′ → N → Bool
  wf_PR(pr)(n) ≡
    ∀ i:Nat • {i,i+1}⊆inds pr ⇒
      obs_HIs(l(i)) ∩ obs_HIs(l(i+1)) ≠ {}
  wf_AR′: AR′ → N → Bool
  wf_AR(ar)(n) ≡
    ∃ pr:PR • pr ∈ routes(n) ∧ wf_PR(pr)(n) ∧ len pr=len ar ∧
       ∀ i:Nat • i ∈ inds ar ⇒ obs_LI(pr(i))=ar(i)
```

305

- A single link is a phenomenological route.
- If r and r' are phenomenological routes
  - such that the last link r
  - and the first link of r'
  - share observable hub identifiers,

  then the concatenation $r\frown r'$ is a route.

  This inductive definition implies a recursive set comprehension.
- A circular phenomenological route is a phenomenological route whose first and last links are distinct but share hub identifiers.
- A looped phenomenological route is a phenomenological route where two distinctly positions (i.e., indexed) links share hub identifiers.

306

```
value
  routes: N → PR-infset
  routes(n) ≡
    let prs = {⟨l⟩|l:L•obs_Ls(n)} ∪
```

```
                    ∪ {pr⌢pr′|pr,pr′:PR•{pr,pr′}⊆prs∧obs_HIs(r(len pr))∩obs_HIs(pr′(1))≠{}}
    prs end

  is_circular: PR → Bool
  is_circular(pr) ≡ obs_HIs(pr(1))∩obs_HIs(pr(len pr))≠{}

  is_looped: PR → Bool
  is_looped(pr) ≡ ∃ i,j:Nat • i≠j∧{i,j}⊆index pr ⇒ obs_HIs(pr(i))∩obs_HIs(pr(j))≠{}
```

307

- Straight routes are Phenomenological routes without loops.
- Phenomenological routes with no loops can be constructed from phenomenological routes by removing suffix routes whose first link give rise to looping.

```
value
  straight_routes: N → PR-set
  straight_routes(n) ≡
    let prs = routes(n) in {straight_route(pr)|pr:PR•ps ∈ prs} end

  straight_route: PR → PR
  straight_route(pr) ≡
    ⟨pr(i)|i:Nat•i:[1..len pr] ∧ pr(i)∉ elems⟨pr(j)|j:Nat•j:[1..i]⟩⟩
```

. . . . **End of Example 10**

A.2.5 Map Expressions 308

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:

Map Enumerations

```
type
  T1, T2
  M = T1 →ₘ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  {[ ], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn],...} ⊆ M
```

Map Comprehension 309

The last line below expresses map comprehension:

Map Comprehension

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

310

Example 11 ....**Concrete Net Type Construction:**

- We Define a function *con[struct]_*N_γ (of the **Cartesians + Maps + Wellformedness** part of Example 3.
  - The base of the construction is the fully abstract sort definition of N_α in the **Sorts + Observers + Axioms** part of Example 3 – where the sorts of hub and link identifiers are taken from earlier examples.
  - The target of the construction is the N_γ of the **Cartesians + Maps + Wellformedness** part of Example 3.
  - First we recall the ssential types of that N_γ.

311

```
type
   N_γ = HUBS × LINKS × GRAPH
   HUBS = HI →m H
   LINKS = LI →m L
   GRAPH = HI →m (LI →m HI)
value
   con_N_γ: N_α → N_γ
   con_N_γ(n_α) ≡
      let hubs = [obs_HI(h) ↦ h | h:H • h ∈ obs_Hs(n_α)],
          links = [obs_LI(h) ↦ l | l:L • l ∈ obs_Ls(n_α)],
          graph = [obs_HI(h) ↦ [obs_LI(l) ↦ ι(obs_HIs(l)\{obs_HI(h)})
                                          | l:L • l ∈ obs_Ls(n_α)∧li ∈ obs_LIs(h)]
                             | H:h • h ∈ obs_Hs(n_α)] in
          (hubs.links,graph) end

   ι: A-set ~→ A [A could be LI-set]
   ι(as) ≡ if card as=1 then let {a}=as in a else chaos end end
```

312

theorem:
n_α **satisfies axioms** 2.–3., 5.–6., and 10. (Page 13) ⇒ wf_N_γ(con_N_γ(n_α))

.... **End of Example 11**

A.2.6 Set Operations 313

Set Operator Signatures

Set Operations

```
value
   21 ∈: A × A-infset → Bool
   22 ∉: A × A-infset → Bool
   23 ∪: A-infset × A-infset → A-infset
   24 ∪: (A-infset)-infset → A-infset
   25 ∩: A-infset × A-infset → A-infset
   26 ∩: (A-infset)-infset → A-infset
   27 \: A-infset × A-infset → A-infset
   28 ⊂: A-infset × A-infset → Bool
   29 ⊆: A-infset × A-infset → Bool
   30 =: A-infset × A-infset → Bool
   31 ≠: A-infset × A-infset → Bool
   32 card: A-infset ~→ Nat
```

Set Examples 314

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
```

Informal Explication 315

21. ∈: The membership operator expresses that an element is a member of a set.
22. ∉: The non-membership operator expresses that an element is not a member of a set.
23. ∪: 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.
24. ∪: 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.
25. ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
26. ∩: 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. 316
27. \: 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.
28. ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
29. ⊂: 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.
30. =: The equal operator expresses that the two operand sets are identical.
31. ≠: The non-equal operator expresses that the two operand sets are *not* identical.
32. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions 317

The operations can be defined as follows (≡ is the definition symbol):

Set Operation Definitions

```
value
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
```

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

A.2.7 Cartesian Operations 318

Cartesian Operations

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

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

A.2.8 List Operations 319

List Operator Signatures

List Operations

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ⁀: A^∗ × A^ω → A^ω
  =: A^ω × A^ω → Bool
  ≠: A^ω × A^ω → Bool
```

List Operation Examples 320

List 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⟩
```

Informal Explication 321

- **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. 322
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The non-equal operator expresses that the two operand lists are *not* identical. 323

The operations can also be defined as follows:

List Operator "Definitions"

```
value
   is_finite_list: A^ω → Bool

   len q ≡
      case is_finite_list(q) of
```

```
         true → if q = ⟨⟩ then 0 else 1 + len tl q end,
         false → chaos end

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

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

324

```
   q(i) ≡
      case (q,i) of
         (⟨⟩,1) → chaos,
         (_,1) → let a:A,q′:Q • q=⟨a⟩⌢q′ in a end
         _ → 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″)
```

A.2.9 Map Operations 325

Map Operator Signatures and Map Operation Examples

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

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

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

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

326

$\cup$: M $\times$ M $\rightarrow$ M [merge $\cup$]
  $[a\mapsto b, a'\mapsto b', a''\mapsto b''] \cup [a'''\mapsto b'''] = [a\mapsto b, a'\mapsto b', a''\mapsto b'', a'''\mapsto b''']$

$\backslash$: M $\times$ **A-infset** $\rightarrow$ M [restriction by]
  $[a\mapsto b, a'\mapsto b', a''\mapsto b'']\backslash\{a\} = [a'\mapsto b', a''\mapsto b'']$

/: M $\times$ **A-infset** $\rightarrow$ M [restriction to]
  $[a\mapsto b, a'\mapsto b', a''\mapsto b'']/\{a', a''\} = [a'\mapsto b', a''\mapsto b'']$

$=,\neq$: M $\times$ M $\rightarrow$ **Bool**

$^\circ$: (A $\xrightarrow[m]{}$ B) $\times$ (B $\xrightarrow[m]{}$ C) $\rightarrow$ (A $\xrightarrow[m]{}$ C) [composition]
  $[a\mapsto b, a'\mapsto b'] \circ [b\mapsto c, b'\mapsto c', b''\mapsto c''] = [a\mapsto c, a'\mapsto c']$

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- $\cup$: 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.
- /: 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.
- $\neq$: The non-equal operator expresses that the two operand maps are *not* identical.
- $^\circ$: 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.

Example 12 .... **Miscellaneous Net Expressions: Maps:**
Example 3 on page 92 left out defining the well-formedness of the map types:

```
type
    GRAPH = HI →m (LI →m HI-set)
    HUBS = HI →m H
    LINKS = LI →m L
    N_γ = HUBS × LINKS × GRAPH
value
    wf_HUBS: H-set → Bool
    wf_HUBS(hubs) ≡ ∀ hi:HI • hi ∈ dom hubs ⇒ obs_HI(hubs(hi))=hi
    wf_LINKS: L-set → Bool
    wf_LINKS(links) ≡ ∀ li:LI • li ∈ dom links ⇒ obs_LI(links(li))=li
    wf_N_γ: N_γ → Bool
    wf_N_γ(hs,ls,g) ≡
        dom hs = dom g ∧
        ∪ {dom g(hi)|hi:HI • hi ∈ dom g} = dom links ∧
        ∪ {rng g(hi)|hi:HI • hi ∈ dom g} = dom g ∧
        ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒ (g(hi))(li)≠hi
        ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒
            ∃ hi′:HI • hi′ ∈ dom g ⇒ ∃ ! li:LI • li ∈ dom g(hi) ⇒ (g(hi))(li) = hi′ ∧ (g(hi′))(li) = hi
```

327

- *Hubs* record the same hubs as do the net corresponding *GRAPH* (**dom** *hs* = **dom** *g* ∧).
- *GRAPH* record the same links as do the net corresponding *LINKS* (∪ {**dom** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *links*).
- The target (or range) hub identifiers of graphs are the same as the domain of the graph (∪ {**rng** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *g*), that is none missing, no new ones !
- No links emanate from and are incident upon the same hub (∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)*≠*hi*).
- If there is a link from one hub to another in the *GRAPH*, then the same link also connects the other hub to the former (∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ ∃ *hi′:HI* • *hi′* ∈ **dom** *g* ⇒ ∃ ! *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)* = *hi′* ∧ *(g(hi′))(li)* = *hi*).

.... **End of Example 12**

Map Operation "Redefinitions"

328

The map operations can also be defined as follows:

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

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

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

329

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

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

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

A.3 The RSL Predicate Calculus 330

333

A.3.1 Propositional Expressions

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

334

Propositional Expressions

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*.

A.3.2 Simple Predicate Expressions 331

335

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:

Simple Predicate Expressions

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.

A.3.3 Quantified Expressions 332

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:

Quantified Expressions

∀ 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.

Example 13 . . . . **Predicates Over Net Quantities:**

From earlier examples we show some predicates:

- Example 1: Right hand side of function definition *is_two_way_link(l)*:
  ∃ *lσ:LΣ* • *lσ* ∈ *obs_HΣ(l)*∧**card** *lσ*=2
- Example 3:
  - The **Sorts** + **Observers** + **Axioms** part:
    - ∗ Right hand side of the wellformedness function *wf_N(n)* definition:
      ∀ *n:N* • **card** *obs_Hs(n)*≥*2* ∧ **card** *obs_Ls(n)*≥*1* ∧ **axioms** 2.–3., 5.–6., and 10., (Page 13)
    - ∗ Right hand side of the wellformedness function *wf_N(hs,ls)* definition:
      card *hs*≥*2* ∧ **card** *ls*≥*1* ...
  - The **Cartesians** + **Maps** + **Wellformedness** part:
    - ∗ Right hand side of the *wf_HUBS* wellformedness function definition:
      ∀ *hi:HI* • *hi* ∈ **dom** *hubs* ⇒ *obs_HIhubs(hi)*=*hi*

* Right hand side of the *wf_LINKS* wellformedness function definition:
  $\forall$ *li:LI* • *li* $\in$ **dom** *links* $\Rightarrow$ *obs_LIlinks(li)=li*
* Right hand side of the *wf_N(hs,ls,g)* wellformedness function definition:
  [*c*] **dom** *hs* = **dom** *g* $\wedge$
  [*d*] $\cup$ {**dom** *g(hi)*|*hi:HI* • *hi* $\in$ **dom** *g*} = **dom** *links* $\wedge$
  [*e*] $\cup$ {**rng** *g(hi)*|*hi:HI* • *hi* $\in$ **dom** *g*} = **dom** *g* $\wedge$
  [*f*] $\forall$ *hi:HI* • *hi* $\in$ **dom** *g* $\Rightarrow$ $\forall$ *li:LI* • *li* $\in$ **dom** *g(hi)* $\Rightarrow$ *(g(hi))(li)*$\neq$*hi*
  [*g*] $\forall$ *hi:HI* • *hi* $\in$ **dom** *g* $\Rightarrow$ $\forall$ *li:LI* • *li* $\in$ **dom** *g(hi)* $\Rightarrow$
  $\exists$ *hi′:HI* • *hi′* $\in$ **dom** *g* $\Rightarrow$ $\exists$! *li:LI* • *li* $\in$ **dom** *g(hi)* $\Rightarrow$
  (g(hi))(li) = *hi′* $\wedge$ *(g(hi′))(li)* = *hi*

.... **End of Example 13**

A.4 λ-Calculus + Functions 336

A.4.1 The λ-Calculus Syntax

λ-Calculus Syntax

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

A.4.2 Free and Bound Variables 337

Free and Bound Variables 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).

340

A.4.3 Substitution 338

In RSL, the following rules for substitution apply:
Substitution of an expression N for all free free x in M is expressed: **subst**([N/x]M).

Substitution

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

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

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

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

A.4.4 α-Renaming and β-Reduction 339

α and β Conversions

- 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)

Example 14 .... **Network Traffic:**
We model traffic by introducing a number of model concepts. We simplify, without loosing the essence of this example, namely to show the use of λ–functions, by omitting consideration of dynamically changing nets. These are introduced next:

- Let us assume a net, *n:N*.

- There is a dense set, *T*, of times – for which we omit giving an appropriate definition.
- There is a sort, *V*, of vehicles.
- *TS* is a dense subset of *T*.
- For each *ts:TS* we can define a minimum and a maximum time. 341
- The $\mathcal{MIN}$ and $\mathcal{MAX}$ functions are meta-linguistic, that is, are not defined in our formal specification language RSL, but can be given a satisfactory meaning.
- At any moment some vehicles, *v:V*, have a *pos:Pos*ition on the net and *VP* records those.
- A *Pos*ition is either on a link or at a hub.
- An *onL*ink position can be designated by the link identifier, the identifiers of the from and to hubs, and the fraction, *f:F*, of the distance down the link from the from hub to the to hub.
- An *atH*ub position just designates the hub (by its identifier).
- Traffic, *tf:TF*, is now a continuous function from *T*ime to *NP* ("recordings").
- Modelling traffic in this way, in fact, in whichever way, entails a ("serious") number of well-formedness conditions. These are defined in *wf_TF* (omitted: ...).

342

```
value
  n:N
type
  T, V
  TS = T-infset
axiom
  ∀ ts:TS • ∃ tmin,tmax:T: tmin ∈ ts ∧ tmax ∈ ts ∧ ∀ t:T • t ∈ ts ⇒ tmin ≤ t ≤ tmax
  [that is: ts = {𝓜𝓘𝓝(ts)..𝓜𝓐𝓧(ts)}]
type
  VP = V →m Pos
  TF′ = T → VP,                       TF = {|tf:TF′•wf_TF(tf)(n)|}
  Pos = onL | atH
  onL == mkLPos(hi:HI,li:LI,f:F,hi:HI),    atH == mkHPos(hi:HI)
value
  wf_TF: TF→ N → Bool
  wf_TF(tf)(n) ≡ ...
  𝓓𝓞𝓜𝓐𝓘𝓝: TF → TS
  𝓜𝓘𝓝, 𝓜𝓐𝓧: TS → T
```

343

We have defined the continuous, composite entity of traffic. Now let us define an operation of inserting a vehicle in a traffic.

- To insert a vehicle, v, in a traffic, tf, is prescribable as follows:
  - the vehicle, v, must be designated;
  - a time point, t, "inside" the traffic tf must be stated;
  - a traffic, vtf, from time t of vehicle v must be stated;
  - as well as traffic, tf, into which vtf is to be "merged".
- The resulting traffic is referred to as tf'.

```
value
  insert_V: V × T × TF → TF → TF
  insert_V(v,t,vtf)(tf) as tf′
```

344

- The function *insert_V* is here defined in terms of a pair of pre/post conditions.
- The pre-condition can be prescribed as follows:
  - The insertion time t must be within to open interval of time points in the traffic tf to which insertion applies.
  - The vehicle v must not be among the vehicle positions of tf.
  - The vehicle must be the only vehicle "contained" in the "inserted" traffic vtf.

```
pre: 𝓜𝓘𝓝(𝓓𝓞𝓜𝓐𝓘𝓝(tf)≤t≤𝓜𝓐𝓧(𝓓𝓞𝓜𝓐𝓘𝓝(tf)) ∧
     ∀ t′:T • t′ ∈ 𝓓𝓞𝓜𝓐𝓘𝓝(tf) ⇒ v ∉ dom tf(t′) ∧
     𝓜𝓘𝓝(𝓓𝓞𝓜𝓐𝓘𝓝(vtf)) = t ∧
     ∀ t′:T•t′ ∈ 𝓓𝓞𝓜𝓐𝓘𝓝(vtf) ⇒ dom vtf(t′)={v}
```

345

- The post condition "defines" tf', the traffic resulting from merging vtf with tf:
  - Let ts be the time points of tf and vtf, a time interval.
  - The result traffic, tf', is defines as a λ-function.
  - For any t'' in the time interval
  - if t'' is less than t, the insertion time, then tf' is as tf;
  - if t'' is t or larger then tf' applied to t'', i.e., $tf'(t'')$
    - for any $v' : V$ different from v yields the same as $(tf(t))(v')$,
    - but for v it yields $(vtf(t))(v)$.

346

```
    post: tf′ = λt″•
              let ts = DOMAIN(tf) ∪ DOMAIN(vtf) in
              if MIN(ts) ≤ t″ ≤ MAX(ts)
                 then
                    ((t″<t) → tf(t″),
                    (t″≥t) → [v′↦ if v′≠v then (tf(t))(v′) else (vtf(t))(v) end
                                   |v′:V•v′ ∈ vehicles(tf)])
                 else chaos end
              end
    assumption: wf_TF(vtf)∧wf_TF(tf)
    theorem: wf_TF(tf′)
value
    vehicles: TF → V-set
    vehicles(tf) ≡ {v|t:T,v:V•t ∈ DOMAIN(tf)∧v ∈ dom tf(t)}
```

We leave it as an exercise for the reader to define functions for: removing a vehicle from a traffic, changing to course of a vehicle from an originally (or changed) vehicle traffic to another. etcetera.

. . . . **End of Example 14**

A.4.5 Function Signatures 347

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
    A, B, ..., C
value
    obs_B: A → B
    ...
    obs_C: A → C
```

These functions cannot be defined. Once a domain is presented in which sort A and sorts or types B, ... and C occurs these observer functions can be demonstrated. 348

Example 15 . . . . **Hub and Link Observers:**
Let a net with several hubs and links be presented. Now observer functions obs_Hs and obs_Ls can be demonstrated: one simply "walks" along the net, pointing out this hub and that link, one-by-one until all the net has been visited. 349

The observer functions obs_HI and obs_LI can be likewise demonstrated, for example: when a hub is "visited" its unique identification can be postulated (and "calculated") to be the unique

geographic position of the hub one which is not overlapped by any other hub (or link), and likewise for links.

. . . . **End of Example 15**

A.4.6 Function Definitions 350

Functions can be defined explicitly:

```
type
    A, B
value
    f: A → B [a total function]
    f(a_expr) ≡ b_expr

    g: A ~→ B [a partial function]
    g(a_expr) ≡ b_expr
    pre P(a_expr)
    P: A → Bool
```

351 a_expr, b_expr are A, respectively B valued expressions of any of the kinds illustrated in earlier and later sections of this primer.
Or functions can be defined implicitly:

```
value
    f: A→B
    f(a_expr) as b
    post P(a_expr,b)
    P: A×B→Bool

    g: A~→B
    g(a_expr) as b
    pre P′(a_expr)
    post P(a_expr,b)
    P′: A→Bool
```

where b is just an identifier.

The symbol $\stackrel{\sim}{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. 352 Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

Finally functions, f, g, ..., h, can be defined in terms of axioms over function identifiers, f, g, ..., h, and over identifiers of function arguments and results.

```
type
    A, B, ..., C, D
value
    f: A → B, g: B → C, ..., h: C → D
axiom
    ∀ a:A, b:B, ..., c:C, d:D
       P₁(f,g,...,h,a,b,...,c,d) ∧ ... ∧ Pₙ(f,g,...,h,a,b,...,c,d)
```

353 where $\mathcal{P}_1, \ldots, \mathcal{P}_m$ and $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ designate suitable predicate expressions.

Example 16 .... **Axioms over Hubs, Links and Their Observers:**
The axioms displayed in Items 7–10 on Page 14 of Sect. 2.1 demonstrates how a number of entities and observer functions are constrained (that is, partially defined) by function signatures.

.... **End of Example 16**

A.5 Other Applicative Expressions 354

A.5.1 Simple let Expressions

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

Let Expressions

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

is an "expanded" form of:

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

A.5.2 Recursive let Expressions 355

Recursive **let** expressions are written as:

Recursive let Expressions

```
let f = λa•E(f,a) in B(f,a) end
let f = (λg•λa•E(g,a))(f) in B(f.a) end
let f = F(f) in E(f,a) end where F ≡ λg•λa•E(g,a)
let f = YF in B(f,a) end where YF = F(YF)
```

We read f = **Y**F as "*f is a fix point of F*".

A.5.3 Non-deterministic let Clause 356

The non-deterministic **let** clause:

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

expresses the non-deterministic selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a). If no a:A • P(a) the clause evaluates to **chaos**.

359

A.5.4 Pattern and "Wild Card" let Expressions 357

Patterns and *wild cards* can be used:

Patterns

```
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
```

A.5.5 Conditionals 358

Various kinds of conditional expressions are offered by RSL:

Conditionals

```
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
```

Example 17 .... **Choice Pattern Case Expressions: Insert Links:**
We consider the meaning of the Insert operation designators.

33. The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.

34. We characterise the "is not at odds", i.e., is semantically well-formed, that is:

    - pre_int_Insert(op)(hs,ls),

    as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied. Let (hs,ls) be a value of type N. 360

35. If the command is of the form 2oldH(hi′,l,hi′) then

    $\star 1$ hi′ must be the identifier of a hub in hs,

    $\star s2$ l must not be in ls and its identifier must (also) not be observable in ls, and

    $\star 3$ hi′′ must be the identifier of a(nother) hub in hs.

36. If the command is of the form 1oldH1newH(hi,l,h) then

    $\star 1$ hi must be the identifier of a hub in hs,

    $\star 2$ l must not be in ls and its identifier must (also) not be observable in ls, and

    $\star 3$ h must not be in hs and its identifier must (also) not be observable in hs. 361

37. If the command is of the form 2newH(h′,l,h′′) then

    $\star 1$ h′ — left to the reader as an exercise (see formalisation !),

    $\star 2$ l — left to the reader as an exercise (see formalisation !), and

    $\star 3$ h′′ — left to the reader as an exercise (see formalisation !).

Conditions concerning the new link (second $\star$s, $\star 2$, in the above three cases) can be expressed independent of the insert command category. 362

363

```
value
   33  int_Insert: Insert → N ~→ N
   34′  pre_int_Insert: Ins → N → Bool
   34″  pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2            s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
      case op of
   35)      2oldH(hi′,l,hi″) → {hi′,hi″}∈ iohs(hs),
   36)      1oldH1newH(hi,l,h) →
                hi ∈ iohs(hs) ∧ h∉ hs ∧ obs_HI(h)∉ iohs(hs),
   37)      2newH(h′,l,h″) →
                {h′,h″}∩ hs={} ∧ {obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
      end
```

RSL Explanation

- 33: The value clause **value** int_Insert: Insert → N $\overset{\sim}{\to}$ N names a value, int_Insert, and defines its type to be Insert → N $\overset{\sim}{\to}$ N, that is, a partial function ($\overset{\sim}{\to}$) from Insert commands and nets (N) to nets.
  (int_Insert is thus a function. What that function calculates will be defined later.)

- 34′: The predicate pre_int_Insert: Insert → N → **Bool** function (which is used in connection with int_Insert to assert semantic well-formedness) applies to Insert commands and nets and yield truth value **true** if the command can be meaningfully performed on the net state.

- 34″: The action pre_int_Insert(op)(hs,ls) (that is, the effect of performing the function pre_int_Insert on an Insert command and a net state is defined by a case distinction over the category of the Insert command. But first we test the common property:

- $\star 2$: s_l(op)∉ls∧obs_LI(s_l(op))∉iols(ls), namely that the new link is not an existing net link and that its identifier is not already known.

  – 35): If the Insert command is of kind 2oldH(hi',l,hi") then {hi′,hi″}∈ iohs(hs), that is, then the two distinct argument hub identifiers must not be in the set of known hub identifiers, i.e., of the existing hubs hs.

  – 36): If the Insert command is of kind 1oldH1newH(hi,l,h) then ... exercise left as an exercises to the reader.

  – 37): If the Insert command is of kind 2newH(h',l,h") ... exercise left as an exercises to the reader. The set intersection operation is defined in Sect. A.2.6 on page 106 Item 25 on page 107.

End of RSL Explanation

38. Given a net, (hs,ls), and given a hub identifier, (hi), which can be observed from some hub in the net, xtr_H(hi)(hs,ls) extracts the hub with that identifier.

39. Given a net, (hs,ls), and given a link identifier, (li), which can be observed from some link in the net, xtr_L(li)(hs,ls) extracts the hub with that identifier.

```
value
38: xtr_H: HI → N ~→ H
38: xtr_H(hi)(hs,_) ≡ let h:H•h ∈ hs ∧ obs_HI(h)=hi in h end
                      pre hi ∈ iohs(hs)
39: xtr_L: HI → N ~→ H
39: xtr_L(li)(_,ls) ≡ let l:L•l ∈ ls ∧ obs_LI(l)=li in l end
                     pre li ∈ iols(ls)
```

RSL Explanation

- 38: Function application xtr_H(hi)(hs,_) yields the hub h, i.e. the value h of type H, such that (•) h is in hs and h has hub identifier hi.
- 38: The wild-card, _, expresses that the extraction (xtr_H) function does not need the L-**set** argument.
- 39: Left as an exercise for the reader.

End of RSL Explanation

364

40. When a new link is joined to an existing hub then the observable link identifiers of that hub must be updated to reflect the link identifier of the new link.
41. When an existing link is removed from a remaining hub then the observable link identifiers of that hub must be updated to reflect the removed link (identifier).

```
value
  aLI: H × LI → H, rLI: H × LI ~→ H
  40:  aLI(h,li) as h′
         pre li ∉ obs_LIs(h)
         post obs_LIs(h′) = {li} ∪ obs_LIs(h) ∧ non_l_eq(h,h′)
  41:  rLI(h′,li) as h
         pre li ∈ obs_LIs(h′) ∧ card obs_LIs(h′)≥2
         post obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′)
```

RSL Explanation

- 40: The add link identifier function aLI:
  - The function definition clause aLI(h,li) **as** h′ defines the application of aLI to a pair (h,li) to yield an update, h′ of h.
  - The pre-condition **pre** li ∉ obs_LIs(h) expresses that the link identifier li must not be observable h.

  - The post-condition **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except (\) that the argument link identifier is not in the resulting hub.
- 41: The remove link identifier function rLI:
  - The function definition clause rLI(h′,li) **as** h defines the application of rLI to a pair (h′,li) to yield an update, h of h′.
  - The pre-condition clause **pre** li ∈ obs_LIs(h′) ∧ **card** obs_LIs(h′)≥2 expresses that the link identifier li must not be observable h.
  - post-condition clause **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except that the argument link identifier is not in the resulting hub.

End of RSL Explanation

365

42. If the Insert command is of kind 2newH(h',l,h") then the updated net of hubs and links, has
    - the hubs hs joined, ∪, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
43. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has

    43.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.

    43.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.

    43.2 : The set of links augmented by the new link.
44. If the Insert command is of kind 2oldH(hi',l,hi") then

    44.1–.2 : the two connecting hubs are updated to reflect the new link,

    44.3 : and the resulting sets of hubs and links updated.

366

```
int_Insert(op)(hs,ls) ≡
⋆ᵢ  case op of
42       2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
43       1oldH1newH(hi,l,h) →
43.1         let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
43.2         (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
44       2oldH(hi′,l,hi″) →
```

```
44.1    let hs$\delta$ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
44.2                     aLI(xtr_H(hi″,hs),obs_LI(l))} in
44.3    (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hs$\delta$,ls ∪{l}) end
$\star_j$  end
$\star_k$  pre pre_int_Insert(op)(hs,ls)
```

RSL Explanation

- $\star_i$–$\star_j$: The clause **case** op **of** $p_1 \longrightarrow c_1$, $p_2 \longrightarrow c_2$, . . . $p_n \longrightarrow c_n$ **end** is a conditional clause.
- $\star_k$: The pre-condition expresses that the insert command is semantically well-formed — which means that those reference identifiers that are used are known and that the new link and hubs are not known in the net.
- $\star_i$ + 42: If op is of the form 2newH(h′,l,h″ then — the narrative explains the rest;

  else
- $\star_i$ + 43: If op is of the form 1oldH1newH(hi,l,h) then
  - 43.1: h′ is the known hub (identified by hi) updated to reflect the new link being connected to that hub,
  - 43.2: and the pair [(updated hs,updated ls)] reflects the new net: the hubs have the hub originally known by hi replaced by h′, and the links have been simple extended (∪) by the singleton set of the new link;

  else
- $\star_i$ + 44: 44: If op is of the form 2oldH(hi′,l,hi″) then
  - 44.1: the first element of the set of two hubs (hsδ) reflect one of the updated hubs,
  - 44.2: the second element of the set of two hubs (hsδ) reflect the other of the updated hubs,
  - 44.3: the set of two original hubs known by the argument hub identifiers are removed and replaced by the set hsδ;

  else — well, there is no need for a further 'else' part as the operator can only be of either of the three mutually exclusive forms !

End of RSL Explanation

367

45. The remove command is of the form Rmv(li) for some li.
46. We now sketch the meaning of removing a link:

a) The link identifier, li, is, by the pre_int_Remove pre-condition, that of a link, l, in the net.

b) That link connects to two hubs, let us refer to them as h′ and h′.

c) For each of these two hubs, say h, the following holds wrt. removal of their connecting link:

   i. If l is the only link connected to h then hub h is removed. This may mean that
      - either one
      - or two hubs

      are also removed when the link is removed.

   ii. If l is not the only link connected to h then the hub h is modified to reflect that it is no longer connected to l.

d) The resulting net is that of the pair of adjusted set of hubs and links.

368

```
value
  45  int_Remove: Rmv → N $\overset{\sim}{\rightarrow}$ N
  46  int_Remove(Rmv(li))(hs,ls) ≡
  46a)  let l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) in
  46b)  let {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} in
  46c)  let hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) in
  46d)  (hs\{h′,h″} ∪ hs′,ls\{l}) end end end
  46a)  pre li ∈ iols(ls)

  cond_rmv: LI × H × H-set → H-set
  cond_rmv(li,h,hs) ≡
  46(c)i)   if obs_HIs(h)={li} then {}
  46(c)ii)   else {sLI(li,h)} end
  pre li ∈ obs_HIs(h)
```

RSL Explanation

- 45: The int_Remove operation applies to a remove command Rmv(li) and a net (hs,ls) and yields a net — provided the remove command is semantically well-formed.
- 46: To Remove a link identifier by li from the net (hs,ls) can be formalised as follows:
  - 46a): obtain the link l from its identifier li and the set of links ls, and
  - 46a): obtain the identifiers, {hi′,hi″}, of the two distinct hubs to which link l is connected;
  - 46b): then obtain the hubs {h′,h″} with these identifiers;
  - 46c): now examine cond_rmv each of these hubs (see Lines 46(c)i)–46(c)ii)).

  - ◦ The examination function cond_rmv either yields an empty set or the singleton set of one modified hub (a link identifier has been removed).
  - ◦ 46c) The set, hs′, of zero, one or two modified hubs is yielded.
  - ◦ That set is joined to the result of removing the hubs {h′,h″}
  - ◦ and the set of links that result from removing l from ls.

  The conditional hub remove function cond_rmv
- – 46(c)i): either yields the empty set (of no hubs) if li is the only link identifier inh,
- – 46(c)ii): or yields a modification of h in which the link identifier li is no longer observable.

End of RSL Explanation

. . . . **End of Example 17**

A.5.6 Operator/Operand Expressions 369

Operator/Operand Expressions

$\langle$Expr$\rangle$ ::=
  $\langle$Prefix_Op$\rangle$ $\langle$Expr$\rangle$
  | $\langle$Expr$\rangle$ $\langle$Infix_Op$\rangle$ $\langle$Expr$\rangle$
  | $\langle$Expr$\rangle$ $\langle$Suffix_Op$\rangle$
  | ...
$\langle$Prefix_Op$\rangle$ ::=
  $- \mid \sim \mid \cup \mid \cap \mid$ **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
$\langle$Infix_Op$\rangle$ ::=
  $= \mid \neq \mid \equiv \mid + \mid - \mid * \mid \uparrow \mid / \mid < \mid \leq \mid \geq \mid > \mid \wedge \mid \vee \mid \Rightarrow$
  $\mid \in \mid \notin \mid \cup \mid \cap \mid \backslash \mid \subset \mid \subseteq \mid \supseteq \mid \supset \mid \hat{} \mid \dagger \mid \circ$
$\langle$Suffix_Op$\rangle$ ::= !

A.6 Imperative Constructs 370

A.6.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract, sorts and applicative constructs which, through stages of refinements, are turned into concrete types and imperative constructs.

Imperative constructs are thus inevitable in RSL.

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

- The **Unit** clause, in a sense, denotes “an underlying state”
  - – which we, for simplicity, can consider as
  - – a mapping from identifiers of declared variables into their values.
- Statements accept no arguments and, usually, operate on the state
  - – through “reading” the value(s) of declared variables and
  - – through “writing”, i.e., assigning values to such declared variables.
- Statement execution thus changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Affixing () as an “only” arguments to a function “means” that () is an argument of type **Unit**.

A.6.2 Variables and Assignment 371

Variables and Assignment

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

A.6.3 Statement Sequences and skip

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

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

A.6.4 Imperative Conditionals

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

A.6.5 Iterative Conditionals 372

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

A.6.6 Iterative Sequencing

8. **for** i **in** list • P(list(i)) **do** S(list(i)) **end**
9. **for** e **in** set • P(e) **do** S(e) **end**

A.7 Process Constructs 373

A.7.1 Process Channels

Let A, B and C stand for three types of (channel) messages and i:IIdx, j:JIdx for channel array indexes, then:

Process Channels

```
channel
   c:A
channel
   {k[i]|i:IIdx}:B
   {ch[i,j]i:IIdx,j:JIdx}:C
```

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B). 374

Example 18 ....**Modelling Connected Links and Hubs:**
Examples (18–21) of this section, i.e., Sect. A.7 are building up a model of one form of meaning of a transport net. We model the movement of vehicles around hubs and links. We think of each hub, each link and each vehicle to be a process. These processes communicate via channels. 375

- We assume a net, $n : N$, and a set, vs, of vehicles.
- Each vehicle can potentially interact
  - with each hub and
  - with each link.
- Array channel indices *(vi,hi):IVH* and *(vi,li):IVL* serve to effect these interactions.
- Each hub can interact with each of its connected links and indices *(hi,li):IHL* serves these interactions.

376

```
type
   N, V, VI
value
   n:N, vs:V-set
   obs_VI: V → VI
type
   H, L, HI, LI, M
   IVH = VI×HI, IVL = VI×LI, IHL = HI×LI
```

- We need some auxiliary quantities in order to be able to express subsequent channel declarations.
- Given that we assume a net, $n : N$ and a set of vehicles, $vs : VS$, we can now define the following (global) values:
  - the sets of hubs, hs, and links, ls of the net;
  - the set, $ivhs$, of indices between vehicles and hubs,
  - the set, $ivls$, of indices between vehicles and links, and
  - the set, $ihls$, of indices between hubs and links.

```
value
   hs:H-set = obs_Hs(n), ls:L-set = obs_Ls(n)
   his:HI-set = {obs_HI(h)|h:H•h ∈ hs}, lis:LI-set = {obs_LI(h)|l:L•l ∈ ls},
   ivhs:IVH-set = {(obs_VI(v),obs_HI(h))|v:V,h:H•v ∈ vs∧h ∈ hs}
   ivls:IVL-set = {(obs_VI(v),obs_LI(l))|v:V,l:L•v ∈ vs∧l ∈ ls}
   ihls:IHL-set = {(hi,li)|h:H,(hi,li):IHL• h ∈ hs∧hi=obs_HI(h)∧li ∈ obs_LIs(h)}
```

377

- We are now ready to declare the channels:
  - a set of channels, {vh[i]|i:IVH•i∈ivhs} between vehicles and all potentially traversable hubs;
  - a set of channels, {vh[i]|i:IVH•i∈ivhs} between vehicles and all potentially traversable links; and
  - a set of channels, {hl[i]|i:IHL•i∈ihls}, between hubs and connected links.

```
channel
   {vh[i] | i:IVH • i ∈ ivhs} : M
   {vl[i] | i:IVL • i ∈ ivls} : M
   {hl[i] | i:IHL • i ∈ ihls} : M
```

.... **End of Example 18**

A.7.2 Process Definitions 378

A process definition is a function definition. 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.

Processes P and Q are to interact, and to do so "ad infinitum". Processes R and S are to interact, and to do so "once", and then yielding B, respectively D values. 379

```
value
   P: Unit → in c out {k[i]|i:IIdx} Unit
   Q: i:KIdx →  out c in k[i] Unit

   P()  ≡ ... c ? ... k[i] ! e ... ; P()
   Q(i) ≡ ...  c ! e ... k[i] ? ... ; Q(i)
```

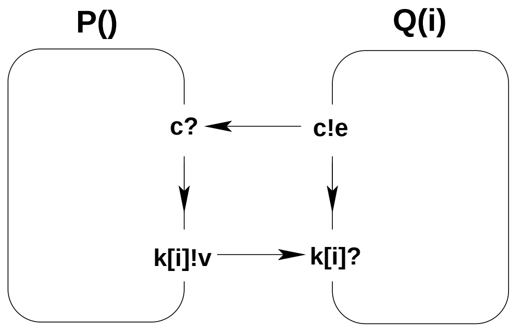

Figure 9: The P —— Q Process

380

Example 19 .... **Communicating Hubs, Links and Vehicles:**

- Hubs interact with links and vehicles:
  - with all immediately adjacent links,
  - and with potentially all vehicles.
- Links interact with hubs and vehicles:
  - with both adjacent hubs,
  - and with potentially all vehicles.

381

- Vehicles interact with hubs and links:
  - with potentially all hubs.
  - and with potentially all links.

```
value
   hub: hi:HI × h:H → in,out {hl[(hi,li)|li:LI•li ∈ obs_LIs(h)]}
                      in,out {vh[(vi,hi)|vi:VI•vi ∈ vis]}  Unit
   link: li:LI × l:L → in,out {hl[(hi,li)|hi:HI•hi ∈ obs_HIs(l)]}
                      in,out {vl[(vi,li)|vi:VI•vi ∈ vis]}  Unit
   vehicle: vi:VI → (Pos × Net) → v:V → in,out {vh[(vi,hi)|hi:HI•hi ∈ his]}  Unit
                      in,out {vl[(vi,li)|li:LI•li ∈ lis]}  Unit
```

.... **End of Example 19**

A.7.3 Process Composition 382

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 $\mathcal{P}$ and $\mathcal{Q}$ stand for process expressions, and let $\mathcal{P}_i$ stand for an indexed process expression, then:

$\mathcal{P} \parallel \mathcal{Q}$	Parallel composition
$\mathcal{P} \talloblong \mathcal{Q}$	Nondeterministic external choice (either/or)
$\mathcal{P} \sqcap \mathcal{Q}$	Nondeterministic internal choice (either/or)
$\mathcal{P} \between \mathcal{Q}$	Interlock parallel composition
$\mathcal{O}$ { $\mathcal{P}_i$ | i:Idx }	Distributed composition, $\mathcal{O} = \parallel, \talloblong, \sqcap, \between$

express the parallel ($\parallel$) of two processes, or the nondeterministic choice between two processes: either external ($\talloblong$) or internal ($\sqcap$). The interlock ($\between$) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

383

Example 20 .... **Modelling Transport Nets:**

- The net, with vehicles, potential or actual, is now considered a process.
- It is the parallel composition of
  - all hub processes,
  - all link processes and
  - all vehicle processes.

```
value
    net: N → V-set → Unit
    net(n)(vs) ≡
        ∥ {hub( obs_HI(h))(h)|h:H•h ∈ obs_Hs(n)} ∥
        ∥ {link( obs_LI(l))(l)|l:L•l ∈ obs_Ls(n)} ∥
        ∥ {vehicle(obs_VI(v))(obs_PN(v))(v)|v:V•v ∈ vs}

    obs_PN: V → (Pos×Net)
```

384

- We illustrate a schematic definition of simplified hub processes.
- The hub process alternates, internally non-deterministically, ⌈⌉, between three sub-processes
  - a sub-process which serves the link-hub connections,
  - a sub-process which serves thos vehicles which communicate that they somehow wish to enter or leave (or do something else with respect to) the hub, and
  - a sub-process which serves the hub itself — whatever that is !

```
    hub(hi)(h) ≡
            ⌈⌉{let m = hl[(hi,li)] ? in hub(hi)(ℰ_hℓ(li)(m)(h)) end|i:LI•li ∈ obs_LI(h)}
        ⌈⌉ ⌈⌉{let m = vh[(vi,hi)] ? in hub(vi)(ℰ_hv(vi)(m)(h)) end|vi:VI•vi ∈ vis}
        ⌈⌉ hub(hi)(ℰ_hown(h))
```

385

- The three auxiliary processes:
  - $\mathcal{E}_{h_\ell}$ update the hub with respect to (wrt.) connected link, *li*, information *m*,
  - $\mathcal{E}_{h_v}$ update the hub with wrt. vehicle, *vi*, information *m*,
  - $\mathcal{E}_{h_{own}}$ update the hub with wrt. whatever the hub so decides. An example could be signalling dependent on previous link-to-hub communicated information, say about traffic density.

```
    ℰ_hℓ:   LI → M → H → H
    ℰ_hv:   VI → M → H → H
    ℰ_hown: H → H
```

The reader is encouraged to sketch/define similarly schematic link and vehicle processes.
.
. . . . **End of Example 20**

A.7.4 Input/Output Events 386

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

```
    [1] c?, k[i]?                    input A value
    [2] c!e, k[i]!e                  output A value
value
    [3] P: ... → out c  ...,  P(...) ≡ ... c!e ...     offer an A value,
    [4] Q: ... → in c  ...,  Q(...) ≡ ... c? ...       accept an A value
    [5] S: ... → ...,  S(...) = P(...)∥Q(...)          synchronise and communicate
```

[5] expresses the willingness of a process to engage in an event that [1,3] "reads" an input, respectively [2,4] "writes" an output. If process P reaches the c!e "program point before" process Q 'reaches program point' c? then process P "waits" on Q — and vice versa. Once both processes have reached these respective program points they "synchronise while communicating the message vale e.

The process function definitions (i.e., their bodies) express possible [output/input] events.

Example 21 . . . . **Modelling Vehicle Movements:**

- Whereas hubs and links are modelled as basically static, passive, that is, inert, processes we shall consider vehicles to be "highly" dynamic, active processes. 387
- We assume that a vehicle possesses knowledge about the road net.
  - The road net is here abstracted as an awareness of
  - which links, by their link identifiers,
  - are connected to any given hub, designated by its hub identifier,
  - the length of the link,
  - and the hub to which the link is connected "at the other end", also by its hub identifier 388
- A vehicle is further modelled by its current position on the net in terms of either hub or link positions
  - designated by appropriate identifiers
  - and, when "on a link" "how far down the link", by a measure of a fraction of the total length of the link, the vehicle has progressed.

```
type
    Net = HI →m (LI →m HI)
    Pos = atH | onL
    atH == mk_atH(hi:HI)
    onL == mk_onL(fhi:HI,li:LI,f:F,thi:HI)
    F = {|f:Real•0≤f≤1|}
```

389

- We first assume that the vehicle is at a hub.
- There are now two possibilities (1–2] versus [4–8]).
  - Either the vehicle remains at that hub
    - [1] which is expressed by some non-deterministic *wait*
    - [2] followed by a resumption of being that vehicle at that location.
  - [3] Or the vehicle (driver) decides to "move on":
    - [5] Onto a link, *li*,
    - [4] among the links, *lis*, emanating from the hub,
    - [6] and towards a next hub, *hi′*.
  - [4,6] The *lis* and *hi′* quantities are obtained from the vehicles own knowledge of the net.
  - [7] The hub and the chosen link are notified by the vehicle of its leaving the hub and entering the link,
  - [8] whereupon the vehicle resumes its being a vehicle at the initial location on the chosen link.

390

- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
  M == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(mk_atH(hi),net)(v) ≡
  [1]  (wait ;
  [2]   vehicle(vi)(mk_atH(hi),net)(v))
  [3]  ⌈⌉
  [4]  (let lis=dom net(hi) in
  [5]   let li:LI•li ∈ lis in
  [6]   let hi′=(net(hi))(li) in
  [7]   (vh[ (vi,hi) ]!mk_H_L(hi,li)‖vl[ (vi,li) ]!mk_H_L(hi,li));
  [8]   vehicle(vi)(mk_onL(hi,li,0,hi′),net)(v)
  [9]   end end end)
```

391

- We then assume that the vehicle is on a link and at a certain distance "down", *f*, that link.

- There are now two possibilities ([1–2] versus [4–7]).
  - Either the vehicle remains at that hub
    - [1′] which is expressed by some non-deterministic *wait*
    - [2′] followed by a resumption of being that vehicle at that location.
  - [3′] Or the vehicle (driver) decides to "move on".
  - [4′] Either
    - [5′] The vehicle is at the very end of the link and signals the link and the hub of its leaving the link and entering the hub,
    - [6′] whereupon the vehicle resumes its being a vehicle at hub *h′*.
  - [7′] or the vehicle moves further down, some non-zero fraction down the link.
- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

392

```
type
  M == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  δ:Real = move(h,f) axiom 0<δ≪1
  vehicle(vi)( mk_onL(hi,li,f,hi′),net)(v) ≡
  [1′]  (wait ;
  [2′]   vehicle(vi)(mk_onL(hi,li,f,hi′),net)(v))
  [3′]  ⌈⌉
  [4′]  (case f of
  [5′]     1 → ((vl[ vi,hi′ ]!mk_L_H(li,hi′)‖vh[ vi,li ]!mk_L_H(li,hi′));
  [6′]           vehicle(vi)(mk_atH(hi′),net)(v)),
  [7′]     _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
  [8′]   end)
  move: H × F → F
```

. . . . **End of Example 21**

A.8 Simple RSL Specifications 393

Besides the above constructs RSL also possesses module-oriented scheme, class and object constructs. We shall not cover these here. An RSL specification is then simply a sequence of one or more clusters of zero, one or more sort and/or type definitions, zero, one or more variable declarations, zero, one or more channel declarations, zero, one or more value definitions (including functions) and zero, one or more and axioms. We can illustrate these specification components schematically:

394

Simple RSL Specifications

```
type
   A, B, C, D, E, F, G
   Hf = A-set, Hi = A-infset
   J = B×C×...×D
   Kf = E*, Ki = Eω
   L = F →m G
   Mt = J → Kf, Mp = J ~→ Ki
   N == alpha | beta | ... | omega
   O == mk_Hf(as:Hf)
         | mk_Kf(el:Kf) | ...
   P = Hf | Kf | L | ...
variable
   vhf:Hf := ⟨⟩
channel
   chf:F, chg:G, {chb[i]|i:A}:B
```

```
value
   va:A, vb:B, ..., ve:E
   f1:  A → B, f2:  C ~→ D
   f1(a) ≡ ℰf1(a)
   f2:  E → in|out chf F
   f2(e) ≡ ℰf2(e)
   f3:  Unit → in chf out chg  Unit
   ...
axiom
   𝒫i(f1,va),
   𝒫j(f2,vb),
   ...
   𝒫k(f3,ve)
```

395

The ordering of these clauses is immaterial. Intuitively the meaning of these definitions and declarations are the following.

The **type** clause introduces a number of user-defined type names; the type names are visible anywhere in the specification; and either denote sorts or concrete types.

The **variable** clause declares some variable names; a variable name denote some value of decalred type; the variable names are visible anywhere in the specification: assigned to ('written') or values 'read'.

The **channel** clause declares some channel names; either simple channels or arrays of channels of some type; the channel names are visible anywhere in the specification. 396

The **value** clause bind (constant) values to value names. These value names are visible anywhere in the specification. The specification

```
type                          value
   A                             a:A
```

non-deterministically binds a to a value of type A. Thuis includes, for example

```
type                          value
   A, B                          f: A → B
```

which non-deterministically binds f to a function value of type A→B. 397

The **axiom** clause is usually expressed as several "comma (,) separated" predicates:

$$\mathcal{P}_i(\overline{A_i}, \overline{f_i}, \overline{v_i}), \mathcal{P}_j(\overline{A_j}, \overline{f_j}, \overline{v_j}), \ldots, \mathcal{P}_k(\overline{A_k}, \overline{f_k}, \overline{v_k})$$

where $(\overline{A_k, f_\ell}, \overline{v\ell})$ is an abbreviation for A_{ℓ_1}, A_{ℓ_2}, ..., A_t, f_{ℓ_1}, f_{ℓ_2}, ..., f_{ℓ_f}, v_{ℓ_1}, v_{ℓ_2}, ..., v_{ℓ_v}. The indexed sort or type names, A and the indexed function names, d, are defined elsewhere in the specification. The index value names, v are usually names of bound 'variables' of universally or existentially quantified predicates of the indexed ("comma"-separated) $\mathcal{P}$. 398

Example 22 .... **A Neat Little "System":**

We present a self-contained specification of a simple system: The system models vehicles moving along a net, *vehicle*, the recording of vehicles entering links, *enter_sensor*, the recording of vehicles leaving links, *leave_sensor*, and the *road_pricing payment* of a vehicle having traversed (*entered* and *left*) a link. Note that vehicles only pay when completing a link traversal; that 'road pricing' only commences once a vehicle enters the first link after possibly having left an earlier link (and hub); and that no *road_pricing payment* is imposed on vehicles entering, staying-in (or at) and leaving hubs. 399

We assume the following: that each *link* is somehow associated with two pairs of *sensors:* a pair of *enter* and *leave sensors* at one end, and a pair of *enter* and *leave sensors* at the other end; and a *road pricing* process which records pairs of link enterings and leavings, first one, then, after any time interval, the other, with leavings leading to debiting of traversal fees; Our first specification define types, assume a net value, declares channels and state signatures of all processes. 400

- *ves* stand for vehicle entering (link) sensor channels,
- *vls* stand for vehicle leaving (link) sensor channels,
- *rp* stand for 'road pricing' channel
- *enter_sensor(hi,li)* stand for vehicle entering [sensor] process from hub *hi* to link *(*li).
- *leave_sensor(li,hi)* stand for vehicle leaving [sensor] process from link *li* to hub *(*hi).
- *road_pricing()* stand for the unique 'road pricing' process.
- *vehicle(vi)(...)* stand for the vehicle *vi* process.

401

```
type
   N, H, HI, LI, VI
   RPM == mk_Enter_L(vi:VI,li:LI) | mk_Leave_L(vi:VI,li:LI)
value
   n:N
channel
   {ves[obs_HI(h),li]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
   {vls[li,obs_HI(h)]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
   rp:RPM
type
   Fee, Bal
   LVS = LI →m VI-set, FEE = LI →m Fee, ACC = VI →m Bal
value
   link: (li:LI × L) → Unit
   enter_sensor: (hi:HI × li:LI) → in ves[hi,li],out rp  Unit
```

```
    leave_sensor: (li:LI × hi:HI) → in vls[li,hi],out rp Unit
    road_pricing: (LVS×FEE×ACC) → in rp Unit
```

402

To understand the sensor behaviours let us review the vehicle behaviour. In the *vehicle* behaviour defined in Example 21, in two parts, Page 137 and Page 138 we focus on the events [7] where the vehicle enters a link, respectively [5′] where the vehicle leaves a link. These are summarised in the schematic reproduction of the vehicle behaviour description. We redirect the interactions between vehicles and links to become interactions between vehicles and enter and leave sensors.

```
value
    δ:Real = move(h,f) axiom 0<δ≪1
    move: H × F → F
    vehicle: VI → (Pos × Net) → V → Unit                                    403
    vehicle(vi)(pos,net)(v) ≡
    [1] (wait ;
    [2]  vehicle(vi)(pos,net)(v))
    [3]  ⌈⌉
        case pos of
            mk_atH(hi) →
    [4−6]  (let lis=dom net(hi) in let li:LI•li ∈ lis in let hi′=(net(hi))(li) in
    [7]        ves[hi,li]!vi;
    [8]        vehicle(vi)(mk_onL(hi,li,0,hi′),net)(v)
    [9]        end end end)
            mk_onL(hi,li,f,hi′) →
    [4′]       (case f of
    [5′−6′]      1 → (vls[li,hi]!vi; vehicle(vi)(mk_atH(hi′),net)(v)),
    [7′]         _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
    [8′]       end)
        end
```

404

- As mentioned on Page 140 *link* behaviours are associated with two pairs of sensors:
  - a pair of *enter* and *leave sensors* at one end, and
  - a pair of *enter* and *leave sensors* at the other end;

```
value
    link(li)(l) ≡
        let {hi,hi′} = obs_HIs(l) in
        enter_sensor(hi,li) ‖ leave_sensor(li,hi) ‖
        enter_sensor(hi′,li) ‖ leave_sensor(li,hi′) end
    enter_sensor(hi,li) ≡
        let vi = ves[hi,li]? in rp!mk_Enter_LI(vi,li); enter_sensor(hi,li) end
    leave_sensor(li,hi) ≡
        let vi = ves[li,hi]? in rp!mk_Leave_LI(vi,li); enter_sensor(li,hi) end
```

405

- The *LVS* component of the *road_pricing* behaviour serves,
  - among other purposes that are not mentioned here,
  - to record whether the movement of a vehicles "originates" along a link or not.
- Otherwise we leave it to the reader to carefully read the formulas.

406

```
value
    payment: VI × LI → (ACC × FEE) → ACC
    payment(vi,li)(fee,acc) ≡
        let bal′ = if vi ∈ dom acc then add(acc(vi),fee(li)) else fee(li) end
        in acc † [vi ↦ bal′] end
    add: Fee × Bal → Bal [add fee to balance]
    road_pricing(lvs,fee,acc) ≡ in rp
        let m = rp? in
        case m of
            mk_Enter_LI(vi,li) →
                road_pricing(lvs†[li↦lvs(li)∪{vi}],fee,acc),
            mk_Leave_LI(vi,li) →
                let lvs′ = if vi ∈ lvs(li) then lvs†[li↦lvs(li)\{vi}] else lvs end,
                    acc′ = payment(vi,li)(fee,acc) in
                road_pricing(lvs′,fee,acc′)
        end end
```

. . . . **End of Example 22**

B Terminology 407

B.1 Term Table of Contents

In any *development project* it is important to define the terms before their first use, to maintain, including adjust, update and extend, such a glossary of term definitions, and to adhere to the definitions.

B.2 Terms

. . . . $\mathcal{A}$

1. **Abstract:** Something which focuses on essential properties. Abstract is a relation: something is abstract with respect to something else (which possesses — what is considered — inessential properties).

2. **Abstract algebra:** An *abstract*[1] *algebra*[26] is an algebra whose carrier elements and whose functions are defined by *postulates* (*axiom*[75]s, *laws*) which specify general properties, rather than values, of functions. (Abstract algebras are also referred to as *postulational*, or *axiom*[75]*atic algebras*. The axiomatic approach to the study of algebras forms the cornerstone of so-called modern algebra [158].)

3. **Abstraction:** 'The art of abstracting. The act of separating in thought; a mere idea; something visionary.'

4. **Abstract data type:** An *abstract*[1] *data*[193] *type*[782] is a set of values for which no external world or computer (i.e., data) representation is being defined, together with a set of abstractly defined functions over these data values.

5. **Abstraction function:** An *abstraction*[3] *function*[310] is a function which applies to *value*[802]s of a *concrete type*[157] and yields values of — what is said to be a corresponding — *abstract type*[7]. (Same as *retrieve function*[624].)

6. **Abstract syntax:** An *abstract*[1] *syntax*[733] is a set of rules, often in the form of an *axiom system*[77], or in the form of a set of *sort definition*[695]s, which defines a set of structures without prescribing a precise external world, or a computer (i.e., data) representation of those structures.

7. **Abstract type:** An *abstract*[1] *type*[782] is the same as an *abstract data type*[4], except that no functions over the data values have been specified.

8. **Accessibility:** We say that a *resource*[620] is accessible by another resource, if that other resource can make use of the former resource. (Accessibility is a *dependability requirement*[218]. Usually accessibility is considered a *machine*[436] property. As such, accessibility is (to be) expressed in a *machine requirements*[438] document.)

9. **Acceptor:** An acceptor is a device, like a *finite state automaton*[289] of a *pushdown automaton*[564], which, when given (i.e., presented with) character strings (or, in general, finite structures), purported to belong to a language, can recognise, i.e., can decide, whether these character strings belong to that language.

10. **Acquirer:** The legal entity, a person, an institution or a firm which orders some *development*[228] to take place. (Synonymous terms are *client*[116] and *customer*[192].)

11. **Acquisition:** The common term means purchase. Here we mean the collection of *knowledge*[407] (about a *domain*[239], about some *requirements*[605], or about some *software*[685]). This collection takes place in an interaction between the *developer*[227]s and representatives of the *client*[116] (*user*[796]s, etc.). (A synonym term is *elicitation*[265].)

12. **Action:** By an action we shall understand something which potentially changes a *state*[705], that is, *value*[802]s of *dynamic*[260] *attribute*[69]s of *simple entities*[681]. We consider *action*[12]s to be one of the four kinds of *entities*[272] that the `Triptych` "repeatedly" considers. The other three are: *simple entities*[681], *event*[281]s and *behaviour*[79]s. Consideration of these are included in the specification of all *domain facet*[250]s and all *requirements facet*[614]s.

13. **Activation stack:** See the *Comment* field of the *function activation*[311] entry.

14. **Active:** By active is understood a *phenomenon*[524] which, over *time*[761], changes *value*[802], and does so either by itself, *autonomous*[73]ly, or also because it is "instructed" (i.e., is "bid" (see *biddable*[85]), or "programmed" (see *programmable*[546]) to do so). (Contrast to *inert*[367] and *reactive*[578].)

15. **Actor:** By an actor we shall understand someone which carries out an *action*[12]. (A synonymous term for actor is *agent*[24].)

16. **Actual argument:** When a function is invoked it is usually applied to a list of values, the actual *argument*[52]s. (See also *formal parameter*[302].)

17. **Actuator:** By an actuator we shall understand an electronic, a mechanical, or an electromechanical device which carries out an *action*[12] that influences some physical *value*[802]. (Usually actuators, together with *sensor*[659]s, are placed in *reactive*[578] systems, and are linked to *controller*[183]s. Cf. *sensor*[659].)

18. **Acyclic:** Acyclicity is normally thought of as a property of graphs. (Hence see next entry: *acyclic graph*[19].)

19. **Acyclic graph:** An acyclic graph is usually thought of as a *directed graph*[232] in which there is no nonempty *path*[517], in the direction of the *arrow*[54]s, from any *node*[479] to itself. (Often acyclic graphs are called directed acyclic graphs, *DAG*s. An undirected graph which is acyclic is a *tree*[777].)

20. **Adaptive:** By adaptive we mean some thing that can adapt or arrange itself to a changing *context*[172], a changing *environment*[275].

21. **Adaptive maintenance:** By adaptive maintenance we mean an update, as here, of software, to fit (to adapt) to a changing environment. (Adaptive maintenance is required when new input/output media are attached to the existing software, or when a new, underlying database management system is to be used (instead of an older such), etc. We also refer to *corrective maintenance*[187], *perfective maintenance*[519], and *preventive maintenance*[541].)

22. **Address:** An address is the same as a *link*[425], a *pointer*[528] or a *reference*[587]: Something which refers to, i.e., designates something (typically something else). (By an address we shall here, in a narrow sense, understand the *location*[431], the place, or position in some *storage*[715] at which some *data*[193] is *store*[714]d or kept.)

23. **Ad hoc polymorphism:** See Comment field of *polymorphic*[529].

24. **Agent:** By an agent we mean the same as an *actor*[15] — a human or a machine (i.e., robot). (The two terms *actor*[15] and *agent*[24] are here considered to be synonymous.)

25. **AI:** Abbreviation for artificial intelligence. (We shall refrain from positing (including risking) a definition of the term AI. Instead we refer to John McCarthy's home page [168].)

26. **Algebra:** An algebra is here taken to just mean: A set of *value*[802]s, A, the *carrier* of the algebra, and a set of *function*[310]s, Φ, on these values such that the result values are within the set of values: $\Phi = \mathsf{A}^* \rightarrow \mathsf{A}$. (We make the distinction between *universal algebra*[790]s, *abstract algebra*[2]s and *concrete algebra*[155]s. See also *heterogeneous algebra*[336]s, *partial algebra*[515]s and *total algebra*[765]s.)

27. **Algebraic semantics:** By an algebraic semantics we understand a *semantics*[655] which denotes one, or a (finite or infinite) set of zero, one or more *algebra*[26]s. (Usually an algebraic semantics is expressed in terms of (i) *sort*[694] definitions, (ii) *function signature*[318]s and (iii) *axiom*[75]s.)

28. **Algebraic systems:** An algebraic system is an *algebra*[26]. (We use the term *system*[736] as an entity with two clearly separable parts: the *carrier*[106] of the algebra and the *function*[310]s of the algebra. We distinguish between *concrete algebra*[155]s, *abstract algebra*[2]s and *universal algebra*[790]s — here listed in order of increasing *abstraction*[3].)

29. **Algebraic type:** An algebraic type is here considered the same as a *sort*[694]. (That is, algebraic types are specified as are *algebraic systems*[28].)

30. **Algol:** Algol stands for Algorithmic Language. (`Algol 60` designed in the period 1958–1960 [12]. It became a reference standard for future language designs (Algol W [232], Algol 68 [223], Pascal [229, 129, 140] and others.)

31. **Algorithm:** The notion of an algorithm is so important that we will give a number of not necessarily complementary definitions, and will then discuss these.

   - By an algorithm we shall understand a precise prescription for carrying out an orderly, finite set of *operation*[493]s on a set of *data*[193] in order to calculate (*compute*[148]) a result. (This is a version of the classical definition. It is compatible with computability in the sense of *Turing machine*[781]s and *Lambda-calculus*[412]. Other terms for algorithm are: effective procedure, and abstract program.)
   - Let there be given a possibly infinite set of *state*[705]s, S, let there be given a possibly infinite set of initial states, I, where $I \subseteq S$, and let there be given a next state function $f : S \rightarrow S$. (C, where $C = (Q, I, f)$ is an initialised, *deterministic*[226] *transition*[772] system.) A sequence $s_0, s_1, \ldots, s_{i-1}, s_i, \ldots, s_m$ such that $f(s_{i-1}) = s_i$ is a *computation*[144]. An algorithm, A, is a C with final states O, i.e.: $A = (Q, I, f, O)$, where $O \subseteq S$, such that each computation ends with a state s_m in O. (This is basically Don Knuth's definition [143]. In that definition a state is a collection of identified data, i.e., a formalised representation of information, i.e., of computable data. Thus Knuth's definition is still Turing and Lambda-calculus "compatible".)
   - There is given the same definition as just above with the generalisation that a state is any association of variables to phenomena, whether the latter are representable "inside" the computer or not. (This is basically Yuri Gurevitch's definition of an algorithm [117, 197, 198]. As such this definition goes beyond Turing machine and Lambda-calculus "compatibility". That is, captures more!)

32. **Algorithmic:** Adjective form of *algorithm*[31].

33. **Allocate:** To apportion for a specific purpose or to particular persons or things, to distribute tasks among human and automated components. (We shall here use the term generally for the allocation of *resources* (see also *resource allocation*[621]), specifically for *storage*[715] to *assignable variable*[59]s. In the general sense, allocation, as the name implies, has some spatial qualities about it: allocation to spatial positions. In the special sense we can indeed talk of storage space.)

34. **Alphabet:** A finite collection of script symbols called the letters of the alphabet.

35. **Alpha-renaming:** By alpha-renaming (α-renaming) we mean the substitution of a *binding*[88] *identifier*[351], with another, the "new", identifier, in some *Lambda-expression*[414]

(statement or clause), such that all free occurrences of that binding identifier in that expression (statement or clause) are replaced by the new identifier, and such that that new identifier is not already bound in that expression (statement or clause). (Alpha-renaming is a concept of the *Lambda-calculus*[412].)

36. **Ambiguous:** A *sentence*[660] is ambiguous if it is open to more than one *interpretation*[397], i.e., has more than one *model*[460] and these models are not *isomorphic*[403].

37. **Analogic:** Equivalency or likeness of relations. Resemblance of relations or attributes as a ground of reasoning. Also: Presumptive reasoning based on the assumption that if things have some similar attributes, their other attributes will be similar [159].

38. **Analogue:** A representative in another class or group [159]. (Used in this technical note in the sense above, not in the sense of electrical engineering or control theory.)

39. **Analysis:** The resolution of anything complex into simple elements. A determination of proper components. The tracing of things to their sources; the discovery of general principles underlying concrete phenomena [159]. (In conventional mathematics analysis pertains to continuous phenomena, e.g. differential and integral calculi. Our analysis is more related to hybrid systems of both discrete and continuous phenomena, or often to just discrete ones.)

40. **Analytic:** Of, or pertaining to, or in accordance with *analysis*[39].

41. **Analytic grammar:** A *grammar*[325], i.e., a *syntax*[733] whose designated sentences (in general: Structures) can be subject to *analysis*[39], i.e., where the syntactic composition can be revealed through *analysis*[39].

42. **Anomaly:** Deviation from the normal.

43. **Anthropomorphic:** Attributing a human personality to anything impersonal or irrational [159]. (See *anthropomorphism*[44]. It seems to be a "disease" of programmers to attribute their programs with human properties: "The program does so-and-so; and after that, it then goes on to do such-and-such," etcetera. Programs, to recall, are, as are any description is, a mere syntactic, i.e., static text. As such they certainly can "do nothing". But they may prescribe that certain actions are effected by machine — when a machine interprets ("executes") the program text!)

44. **Anthropomorphism:** Ascription of a human form and attributes to the Deity, or of a human attribute or personality to anything impersonal or irrational [159]. (See *anthropomorphic*[43].)

45. **Application:** By an application we shall understand either of two rather different things: (i) the application of a function to an *argument*[52], and (ii) the use of software for some specific purpose (i.e., the application). (See next entry for variant (ii).)

46. **Application domain:** An area of activity which some *software*[685] is to support (or supports) or partially or fully automate (resp. automates). (We normally omit the prefix 'application' and just use the term *domain*[239].)

47. **Applicative:** The term applicative is used in connection with applicative programming. It is hence understood as programming where applying functions to *argument*[52]s is a main form of expression, and hence designates function application as a main form of operation. (Thus the terms applicative and *functional*[312] are here used synonymously.)

48. **Applicative programming:** See the term *applicative*[47] just above. (Thus the terms applicative programming and *functional programming*[313] are here used synonymously.)

49. **Applicative programming language:** Same as *functional programming language*[314].

50. **Arc:** Same as an *edge*[262]. (Used normally in connection with *graph*[327]s.)

51. **Architecture:** The structure and content of *software*[685] as perceived by their *user*[796]s and in the context of the *application domain*[46]. (The term architecture is here used in a rather narrow sense when compared with the more common use in civil engineering.)

52. **Argument:** A *value*[802] provided (possibly as part of an argument list) when invoking a function.

53. **Arity:** By the arity of a *function*[310] (i.e., an *operation*[493]) we understand the number (0, 1, or more) of *argument*[52]s that the function applies to. (Usually a function applies to an argument list, and the arity is therefore the length of this list.)

54. **Arrow:** A directed *edge*[262]. (*Branches* are arrows.)

55. **Artefact:** An artificial product [159]. (Anything designed or constructed by humans or machines, which is made by humans.)

56. **Artifact:** Same term as *artefact*[55].

57. **Artificial intelligence:** See *AI*[25].

58. **Assertion:** By an assertion we mean the act of stating positively usually in anticipation of denial or objection. (In the context of *specification*[698]s and *program*[545]s an assertion is usually in the form of a pair of *predicate*[536]s "attached" to the specification text, to the program text, and expressing properties that are believed to hold before any interpretation of the text; that is, "a before" and "an after", or, as we shall also call it: a **pre-** and a **post-**condition.)

59. **Assignable variable:** By an assignable variable we understand an entity of a program text which *denote*[216]s a *storage*[715] *location*[431] whose associated *value*[802] can be changed by an *assignment*[60]. (Usually, in the context of specifications and programs, assignable variables are declared.)

60. **Assignment:** By an assignment we mean an update to, a change of a *storage*[715] *location*[431]. (Usually, in the context of specifications and programs, assignments are prescribed by assignment statements.)

61. **Associative:** Property of a binary operator o: If for all values a, b and c, $(a\ o\ b)\ o\ c\ =\ a\ o\ (b\ o\ c)$, then o is said to be an associative operator. (Addition (+) and multiplication (*) of natural numbers are associative operators.)

62. **Asynchronous:** Not *synchronous*[731]. (In the context of computing we say that two or more *process*[544]es — some of which may represent the world external to the computing device — are asynchronous if occurrences of the *event*[281]s of these processes are not (a priori) coordinated.)

63. **Atomic:** In the context of *software engineering*[693] atomic means: A *phenomenon*[524] (a *concept*[152], a *simple entity*[681], a *value*[802]) which consists of no proper subparts, i.e., no proper sub*phenomena*[524], sub*concept*[152]s, sub*entities*[272] or sub*value*[802]s other than itself. When we consider a *phenomenon*[524], a *concept*[152], a *simple entity*[681], a *value*[802], to be atomic, then it is often a matter of choice, with the choice reflecting a level of *abstraction*[3].

64. **Atomic action:**

65. **Atomic behaviour:**

66. **Atomic entity:** Either an *atomic action*[64], an *atomic behaviour*[65], an *atomic event*[67] or an *atomic simple entity*[68]

67. **Atomic event:**

68. **Atomic simple entity:**

69. **Attribute:** We use the term attribute only in connection with values of composite type. An attribute is now whether a composite value possesses a certain property, or what value it has for a certain component part. (An example is that of database (e.g., SQL) relations (i.e., tabular data structures): Columns of a table (i.e., a relation) are usually labelled with a name designating the attribute (type) for values of that column. Another example is that, say, of a Cartesian: $\mathsf{A} = \mathsf{B}\times\mathsf{C}\times\mathsf{D}$. A can be said to have the attributes B, C, and D. Yet other examples are $\mathsf{M} = \mathsf{A} \underset{m}{\rightarrow} \mathsf{B}$, $\mathsf{S} = \mathsf{A}$-**set** and $\mathsf{L} = \mathsf{A}^*$. M is said to have attributes A and B. S is said to have attribute A. L is said to have attribute A. In general we make the distinction between an entity consisting of subentities (being decomposable into proper parts, cf. *subentity*[721]), and the entities having attributes. A person, like me, has a height attribute, but my height cannot be "composed away from me"!)

70. **Attribute grammar:** A grammar, usually expressed as a *BNF Grammar*[92], where, to each *rule*[638], and to each nonterminal, of the left-hand side or of the right-hand side of the rule, there is associated one or more (attribute) *assignable variable*[59]s together with a set of single assignments to some of these variables — such that the assignment expression variables are those of the attribute variables of the rule.

71. **Automaton:** An automaton is a device with *state*[705]s, *input*[382]s, some states designated as final states, and with a next state *transition*[772] function which to every state and input designates a next state. (There may be a finite, or there may be an infinite number of states. The next state transition function may be *deterministic*[226] or *nondeterministic*[481].)

72. **Automorphism:** An *isomorphism*[404] that maps an algebra into itself is an automorphism. (See also *endomorphism*[268], *epimorphism*[276], *homomorphism*[343], *monomorphism*[467].)

73. **Autonomous:** A *phenomenon*[524] (a *concept*[152], an *entity*[272]) is said to be autonomous if it changes *value*[802] at its own discretion or without influence from an *environment*[275]. (Rephrasing the above we get: (i) A phenomenon is said to be of, or possess, the autonomous active dynamic attribute if it changes value only on its own volition — that is, it cannot also change value as a result of external stimuli; (ii) or when its actions cannot be controlled in any way: That is, they are a "law onto themselves and their surroundings". We speak of such *phenomena* as being *dynamic*[260]. Other dynamic *active*[14] phenomena may be *active*[14] or *reactive*[578].)

74. **Availability:** We say that a *resource*[620] is available for use by other resources, if within a reasonable time interval these other resources can make use of the former resource. (Availability is a *dependability requirement*[218]. Usually availability is considered a *machine*[436] property. As such availability is (to be) expressed in a *machine requirements*[438] document.)

75. **Axiom:** An established rule or principle or a self-evident truth.

76. **Axiomatic specification:** A *specification*[698] presented, i.e., given, in terms of a set of *axiom*[75]s. (Usually an axiomatic specification also includes definitions of *sort*[694]s and *function signature*[318]s.)

77. **Axiom system:** Same as *axiomatic specification*[76].

. . . . $\mathcal{B}$

78. **B:** B stands for Bourbaki, pseudonym for a group of mostly French mathematicians which began meeting in the 1930s, aiming to write a thorough unified set-theoretic

account of all mathematics. They had tremendous influence on the way mathematics has been done since. (The founding of the Bourbaki group is described in André Weil's autobiography, titled something like "memoir of an apprenticeship" (orig. Souvenirs D'apprentissage). There is a usable book on Bourbaki by J. Fang. Liliane Beaulieu has a book forthcoming, which you can sample in "A Parisian Cafe and Ten Proto-Bourbaki Meetings 1934–1935" in the Mathematical Intelligencer 15 no. 1 (1993) 27–35. From `http://www.faqs.org/faqs/sci-math-faq/bourbaki/` (2004). Founding members were: Henri Cartan, Claude Chevalley, Jean Coulomb, Jean Delsarte, Jean Dieudonné, Charles Ehresmann, René de Possel, Szolem Mandelbrojt, André Weil. From: `http://www.bourbaki.ens.fr/` (2004). B also stands for a model-oriented specification language [2].)

79. **Behaviour:** A sequence of *action*[12]s and *event*[281]s is a behaviour. A set of behaviours is a behaviour.

    By behaviour we shall understand the way in which something functions or operates.

    In the context of domain engineering behaviour is a concept associated with *phenomena*[524], in particular manifest *simple entities*[681]. And then behaviour is that which can be observed about the *value*[802] of that *simple entity*[681] and its *interaction*[392] with its *environment*[275].

80. **Behaviour, Communicating:** A concurrent behaviour where actions of one behaviour synchronise and communicate with actions of other behaviours.

81. **Behaviour, Concurrent:** A set of behaviours.

82. **Behaviour, Parallel:** A set of behaviours.

83. **Behaviour, Sequential:** A sequence of actions and events.

84. **Beta-reduction:** By Beta-reduction we understand the substitution whereby all *free*[305] occurrences of a designated *variable*[803] in a *Lambda-expression*[414] are replaced by *Lambda-expression*[414] (in which some *Alpha-renamings* may have to be made first).

85. **Biddable:** A *phenomenon*[524] is biddable if it can be advised (through a "contractual arrangement") on which *action*[12]s are expected of it in various *state*[705]s. (A biddable phenomenon does not have to take these actions, but then the "contractual arrangement" need no longer be honoured by other phenomena (other [sub]domains) with which it *interact*[391]s (i.e., shares phenomena).)

86. **Bijection:** See *bijective function*[87].

87. **Bijective function:** A total *surjective function*[727] which maps all *value*[802]s of its postulated *definition set*[211] into all distinct values of its postulated *range*[576]set is called bijective. (See also *injective function*[380] and *surjective function*[727].)

88. **Binding:** By binding we mean a pairing of, usually, an *identifier*[351], a *name*[474], with some *resource*[620]. (In the context of software engineering we find such bindings as: (i) of an *assignable variable*[59] to a *storage*[715] *location*[431], (ii) of a *procedure*[543] *name*[474] to a procedure *denotation*[213], etc.)

89. **Block:** By a block we shall here understand a textual entity, one that is suitably delineated. (In the context of software engineering a block is normally some partial *specification*[698] which locally introduces some (*applicative*[47], i.e., expression) constant definitions (i.e., **let .. in .. end**), or some (*imperative*[352], i.e., statement) local variable declarations (i.e., **begin dcl .. ; .. end**).)

90. **Block-structured programming language:** A *programming language*[551] is said to be block-structured if it permits such program constructs (incl. *procedures*) whose *semantics*[655] amount to the creation of a local identifier *scope*[649], and where such can be nested, zero, one or more within another.

91. **BNF:** Abbreviation for Backus–Naur Form (Grammar). (See *BNF Grammar*[92].)

92. **BNF Grammar:** By BNF Grammar we mean a concrete, linear textual representation of a *grammar*[325], i.e., a *syntax*[733], one that *designate*[222]s a set of strings. (A BNF Grammar usually is represented in the form of a set of *rule*[638]s. Each rule has a *nonterminal*[484] left-hand-side *symbol*[728] and a finite set of zero, one or more alternative right-hand-side strings of *terminal*[750] and nonterminal symbols.)

93. **Boolean:** By Boolean we mean a data type of logical values (**true** and **false**), and a set of connectives: $\sim$, $\wedge$, $\vee$, and $\Rightarrow$. (Boolean derives from the name of the mathematician George Boole.)

94. **Boolean connective:** By a *Boolean*[93] *connective*[167] we mean either of the Boolean operators: $\wedge$, $\vee$, $\Rightarrow$ (or $\supset$), $\sim$ (or $\neg$).

95. **Bound:** The concept of being bound is associated with (i) *identifier*[351]s (i.e., *name*[474]s) and *expression*[282]s, and (ii) with *name*[474]s (i.e., *identifier*[351]s) and *resource*[620]s. An identifier is said to be either *free*[305] or bound in an expression based on certain rules being satisfied or not. If an identifier is bound in an expression then bound occurrences of that identifier are bound to the same resource. If a name is bound to some resource then all bound occurrences of that name *denote*[216] that resource. (Cf. *free*[305].)

96. **BPR:** See *business process reengineering*[101]

97. **Branch:** Almost the same as an *edge*[262], except that branches are directed, i.e., are (like) *arrow*[54]s. (Used usually in connection with *tree*[777]s.)

98. **Brief:** By a brief is understood a *document*[237], or a part of a document which informs about a *phase*[523] , or a *stage*[702] , or a *step*[711] of *development*[228]. (A brief thus contains *information*[373].)

99. **Business process:** By a business process we shall understand a *behaviour*[79] of an enterprise, a business, an institution, a factory. (Thus a business process reflects the ways in which a business conducts its affairs, and is a *facet*[285] of the *domain*[239]. Other facets of an enterprise are those of its *intrinsics*[399], *management and organisation*[445] (a facet closely related, of course, to business processes), *support technology*[725], *rules and regulations*[640], and *human behaviour*[345].)

100. **Business process engineering:** By *business process engineering*[100] we shall understand the *design*[221], the determination, of *business process*[99]es. (In doing business process engineering one is basically designing, i.e., prescribing entirely new business processes.)

101. **Business process reengineering:** By *business process reengineering*[101] we shall understand the re*design*[221], the change, of *business process*[99]es. (In doing business process reengineering one is basically carrying out *change management*[109].)

. . . . $\mathcal{C}$

102. **Calculate:** Given an expression and an applicable *rule*[638] of a *calculus*[104], to change the former expression into a resulting expression. (Same as *compute*[148].)

103. **Calculation:** A sequence of steps which, from an initial expression, following rules of a *calculus*[104], *calculate*[102]s another, perhaps the same, expression. (Same as *computation*[144].)

104. **Calculus:** A method of *computation*[144] or *calculation*[103] in a special notation. (From mathematics we know the differential and the integral calculi, and also the Laplace calculus. From metamathematics we have learned of the λ-calculus. From logic we know of the Boolean (propositional) calculus.)

105. **Capture:** The term capture is used in connection with *domain knowledge*[254] (i.e., *domain capture*[242]) and with *requirements acquisition*[606]. It shall indicate the act of acquiring, of obtaining, of writing down, domain knowledge, respectively requirements.

106. **Carrier:** By a carrier is understood a, or the set of *entities* of an *algebra*[26] — the former in the case of a *heterogeneous algebra*[336].

107. **Cartesian:** By a Cartesian is understood an ordered product, a fixed grouping, a fixed composition, of *entities*. (Cartesian derives from the name of the French mathematician René Descartes.)

108. **C.C.I.T.T:** Abbreviation for Comité Consultative Internationale de Telegraphie et Telephonie. (CCITT is an alternative form of reference.)

109. **Change management:** Same as *business process reengineering*[101].

110. **Channel:** By a channel is understood a means of *interaction*[392], i.e., of *communication*[122] and possibly of *synchronisation*[729] between *behaviour*[79]s. (In the context of computing we can think of channels as being either input, or output, or both input and output channels.)

111. **Chaos:** By **chaos** we understand the totally undefined *behaviour*[79]: Anything may happen! (In the context of computing **chaos** may, for example, be the *designation*[223] for the never-ending, the never-terminating *process*[544].)

112. **CHI:** Abbreviation for Computer Human Interface. (Same as *HCI*[334].)

113. **CHILL:** Abbreviation for CCITT's High Level Language. (See [62, 118].)

114. **Class:** By a class we mean either of two things: a **class** *clause*[115], as in `RSL`, or a set of *entities* defined by some *specification*[698], typically a *predicate*[536].

115. **Clause:** By a clause is meant an *expression*[282], designating a *value*[802], or a *statement*[707], designating a *state*[705] change, or a sentential form, which designates both a value and a state change. (When we use the term clause we mean it mostly in the latter sense of both designating a value and a side effect.)

116. **Client:** By a client we mean any of three things: (i) The legal body (a person or a company) which orders the development of some software, or (ii) a *process*[544] or a *behaviour*[79] which *interact*[391]s with another process or behaviour (i.e., the *server*[663]), in order to have that server perform some *action*[12]s on behalf of the client, or (iii) a user of some software (i.e., computing system). (We shall normally use the term customer in the first or in the second sense (i, ii).)

117. **Closure:** By a closure is usually meant some transitive closure of a relation $\Re$: If $a\Re b$ and $b\Re c$ then $a\Re c$, and so forth. To this we shall add another meaning, used in connection with implementation of (for example) procedures: Denotationally a procedure, when invoked, in some calling environment, is to be interpreted in the defining environment. Hence a procedure closure is a pair: The procedure text and the defining environment.

118. **Code:** By code we mean a *program*[545] which is expressed in the machine language of a computer.

119. **Coding:** By coding we shall here, simply, mean the act of programming in a machine, i.e., in a computer-close language. (Thus we do not, except where explicitly so mentioned, mean the encoding of one string of characters into another, say for *communication*[122] over a possibly faulty communication *channel*[110] (usually with the decoding of the encoded string "back" into the original, or a similar string).)

120. **Cohesion:** Cohesion expresses a measure of "closeness", of "dependency", of "sticking together" among a set of entities. (In the context of software engineering cohesion

is, as it is here, a term used to express a dependency relation between *module*[464]s of a *specification*[698] or a *program*[545]. Two modules have a higher cohesion the larger the number of cross-references (to types and values, including, in particular functions) there are among them.)

121. **Collision:** Collision, as used here, means that two (or more) occurrences of the same identifier, of which at least one is free, and which at some stage occurred in different text parts, are brought together, say by function application (i.e., macro-expansion) and thereby become bound. (Collision is a concept introduced in the Lambda-calculus, see . Collision is an undesirable effect. See also *confusion*[162].)

122. **Communication:** A *process*[544] by which *information*[373] is exchanged between individuals (*behaviour*[79]s, *process*[544]es) through a common *system*[736] of *symbol*[728]s, *sign*[679]s, or *protocol*[561]s.

123. **Commutative:** Property of a binary operator o: If for all values a and b, $a\ o\ b = b\ o\ a$, then o is said to be a commutative operator. (Addition (+) and multiplication (*) of natural numbers are commutative operators.)

124. **Compilation:** By a compilation we shall mean the conversion, the *translation*[775], of one formal text to another, usually a high-level program text to a low-level machine code text.

125. **Compiler:** By a compiler we understand a device (usually a software package) which given *sentence*[660]s (i.e., *source program*[696]s) in one language, generates sentences (i.e., *target program*[743]s) in another language. (Usually the source and the target languages are related as follows: The source language is normally a so-called "higher-order" language, like Java, and the target language is normally a "lower (abstraction) level" language, like Java Byte Code (or a computer machine language) for which an interpreter is readily available.)

126. **Compiler dictionary:** By a compiler dictionary we shall understand a composite data structure (with a varying number of entries) and a fixed number of operations. The data structure values reflect properties of a program text being compiled. These properties could be: types of some program text variable, type structure of some program text type name, program point of definition of some (goto) label, etc. The possibly hierarchical, i.e., recursively nested, structure of the compiler dictionary further reflects a similarly hierarchical structure of the program text being compiled. The operations include those that insert, update, and search for entries in the compiler dictionary.

127. **Compile time:** By compile time we understand that time interval during which a *source program*[696] is being compiled and during which certain analyses, and hence decisions, can be made about, and actions taken with respect to the source program

(to be, i.e., being, compiled) — such as *type check*[783]ing, name *scope check*[650]ing, etc. (Contrast to *run time*[641].)

128. **Compiling algorithm:** By a compiling algorithm we shall understand a specification which, for every rule in a syntax (of a *source program*[696]ming language), prescribes which *target program*[743]ming language data structure to generate. (We refer to (Sects. 16.8–16.10) for "our story" on compiling algorithms.)

129. **Complete:** We say that a *proof system*[557] is complete if all true sentences are provable.

130. **Completeness:** Noun form of the *complete*[129] adjective.

131. **Component:** By a component we shall here understand a set of type definitions and component local variable declarations, i.e., a component local state, this together with a (usually complete) set of modules, such that these modules together implement a set of concepts and facilities, i.e., functions, that are judged to relate to one another.

132. **Component design:** By a component design we shall understand the *design*[221] of (one or more) *component*[131]s. (We shall refer to for "our story" on component design.)

133. **Composite:** We say that a *phenomenon*[524] or a *concept*[152], is composite when it is possible and meaningful to consider that phenomenon or concept as analysable into two or more subphenomena or subconcepts.

134. **Composite action:**

135. **Composite behaviour:**

136. **Composite entity:** Either a *composite action*[134], a *composite behaviour*[135], a *composite event*[137] or a *composite simple entity*[138].

137. **Composite event:**

138. **Composite simple entity:**

139. **Composite type:**

140. **Composition:** By composition we mean the way in which a *phenomenon*[524], a *concept*[152], is "put together" (i.e., composed) into a *composite*[133] *phenomenon*[524], resp. *concept*[152].

141. **Compositional:** We say that two or more *phenomena* or *concepts* are compositional if it is meaningful to *compose* these phenomena and/or concepts. (Typically a *denotational semantics*[215] is expressed compositionally: By composing the semantics of sentence parts into the semantics of the composition of the sentence parts.)

142. **Compositional documentation:** By compositional documentation we mean a development, or a presentation (of that development), of, as here, some *description*[220] (*prescription*[540] or *specification*[698]), in which some notion of "smallest", i.e., atomic phenomena and concepts are developed (resp. presented) first, then their compositions, etc., until some notion of full, complete development (etc.) has been achieved. (See also *composition*[140], *compositional*[141] and *hierarchical documentation*[340].)

143. **Comprehension:** By comprehension we shall here mean *set*[664], *list*[428] or *map*[449] comprehension, that is, the expression, of a set, a list, respectively a map, by a predicate over the elements of the set, list or pairings of the map, that belong to the set, list, respectively the map.

144. **Computation:** See *calculation*[103].

145. **Computational linguistics:** The study and knowledge of the *syntax*[733] and *semantics*[655] of *language*[417] based on notions of *computer science*[149] and *computing science*[150]. (Thus computational linguistics emphasises those aspects of language whose analysis (*recognition*), or synthesis (*generation*), can be mechanised.)

146. **Computational data+control requirements:** By a computational data + control requirements we mean a requirements which express how the dynamics of computations or data (may) warrant interaction between the machine and its environment, hence is an *interface requirements*[394] *facet*[285]. (See also *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448], and *machine-machine dialogue requirements*[437].)

147. **Computational semantics:** By a computational semantics we mean a specification of the semantics of a language which emphasises run-time computations, i.e., state-to-next-state transitions, as effected when following the prescriptions of programs. (Terms similar in meaning to computational semantics are *operational semantics*[496] and *structural operational semantics*[720].)

148. **Compute:** Given an expression and an applicable *rule*[638] of a *calculus*[104], to change the former expression into a resulting expression. (Same as *calculate*[102].)

149. **Computer Science:** The study and knowledge of the phenomena that can exist inside computers.

150. **Computing Science:** The study and knowledge of how to construct those phenomena that can exist inside computers.

151. **Computing system:** A combination of *hardware*[331] and *software*[685] that together make meaningful *computation*[144]s possible.

152. **Concept:** An abstract or generic idea generalised from phenomena or concepts. (A working definition of a concept has it comprising two components: The *extension*[283] and the *intension*[389]. A word of warning: Whenever we describe something claimed to be a "real instance", i.e., a physical *phenomenon*[524], then even the description becomes that of a concept, not of "that real thing"!)

153. **Concept formation:** The forming, the enunciation, the *analysis*[39], and definition of *concepts* (on the basis, as here, of *analysis*[39] of the *universe of discourse*[793] (be it a *domain*[239] or some *requirements*[605])). (Domain and requirements concept formation(s) is treated in Vol. 3, Chaps. 13 (Domain Analysis and Concept Formation) and 21 (Requirements Analysis and Concept Formation).)

154. **Concrete:** By concrete we understand a *phenomenon*[524] or, even, a *concept*[152], whose explication, as far as is possible, considers all that can be observed about the phenomenon, respectively the concept. (We shall, however, use the term concrete more loosely: To characterise that something, being specified, is "more concrete" (possessing more properties) than something else, which has been specified, and which is thus considered "more abstract" (possessing fewer properties [considered more relevant]).)

155. **Concrete algebra:** A *concrete*[154] *algebra*[26] is an algebra whose carrier is some known set of mathematical elements and whose functions are known, i.e., well-defined. That is, the *model*[460]s of both the carrier and all the functions are pre-established. (Concrete algebras are the level of the empirical (actual) world of mathematics and its applications, where one deals with specific sets of elements (integers, Booleans, reals, etc.), and where operations on these sets that are defined by rules or algorithms or combinations. In general one "knows" a concrete algebra when one knows what the elements of the carrier A are and how to *evaluate*[279] the functions $\phi_i : \Phi$ over A [158].)

156. **Concrete syntax:** A *concrete*[154] *syntax*[733] is a syntax which prescribes actual, computer representable *data structure*[199]s. (Typically a *BNF Grammar*[92] is a concrete syntax.)

157. **Concrete type:** A *concrete*[154] *type*[782] is a type which prescribes actual, computer representable *data*[193] *structure*[719]s. (Typically the type definitions of programming languages designate concrete types.)

158. **Concurrency:** By concurrency we mean the simultaneous existence of two or more *behaviour*[79]s, i.e., two or more *process*[544]es. (That is, a *phenomenon*[524] is said to exhibit concurrency when one can analyse the phenomenon into two or more *concurrent*[159] phenomena.)

159. **Concurrent:** Two (or more) *event*[281]s can be said to occur concurrently, i.e., be concurrent, when one cannot meaningfully describe any one of these events to ("always")

"occur" before any other of these events. (Thus concurrent systems are systems of two or more processes (behaviours) where the simultaneous happening of "things" (i.e., events) is deemed beneficial, or useful, or, at least, to take place!)

160. **Configuration:** By a configuration we shall here understand the *composition*[140] of two or more *semantic value*[802]s. (Usually we shall decompose a configuration into parts such that each part enjoys a *temporal*[747] relationship with respect to the other parts: being "more *dynamic*[260]", being "more *static*[708]", etc. More specifically, we shall typically model the semantics of *imperative*[352] programming languages in terms of *semantic function*[656]s over configurations composed from *environment*[275]s and *storage*[715]s.)

161. **Conformance:** Conformance is a relation between two *document*[237]s (A and B). B is said to conform to A, if everything A specifies is satisfied by B. (Conformance is thus, here, taken to be the same as *correct*[185]ness, i.e., *congruence*[163]. Usually conformance is used in standardisation documents: *Any system claiming to follow this standard must show conformance to it.*)

162. **Confusion:** Confusion, as used here, means that two (or more) occurrences of the same identifier, bound to possibly different values, may be confused in that it is difficult from a smaller context of the text in which they occur to discern, to decide, which meanings, which values, the various occurrences are bound to. (Confusion is a concept introduced in the Lambda-calculus, see . Confusion is an OK, albeit annoying, effect! See also *collision*[121].)

163. **Congruence:** An *algebra*[26], A, is said to be congruent with another algebra, B, if, for every operation, o_B, and suitable set of arguments, b_1, b_2, ..., b_n, to that operation, in B, there corresponds an operation, o_A, and a suitable set of arguments, a_1, a_2, ..., a_n, in A such that $o_A(a_1, a_2, \ldots, a_n) = o_B(b_1, b_2, \ldots, b_n)$. (Compare this definition to that of *conformance*[161]. The difference is one between a precise, mathematical meaning of congruence, as contrasted to an informal meaning of conformance.)

164. **Conjunction:** Being combined, being conjoined, composed. (We shall mostly think of conjunction as the (meaning of the) logical connective "and": $\wedge$.)

165. **Connection:** Connection is a topological notion, and, as such, is also an ontological concept related to "parts and wholes", where parts may be, or may not be connected, i.e., "so close" to one another, that there can be no other parts "inserted in between".

166. **Connector:** We shall here, by a connector, mean a hardware, or some software device that "connects" two like devices, hardware+hardware, or software+software. (Typically, in software engineering, when "connecting" two independently developed *component*[131]s, one deploys a connector in order to connect them.)

167. **Connective:** By a connective is here meant one of the Boolean "operators": "and" $\wedge$, "or" $\vee$, "imply" $\Rightarrow$, and "negation" $\sim$.

168. **Consistent:** A set of *axiom*[75]s is said to be consistent if, by means of these, and some *deduction rule*[206]s, one cannot *prove* a property and its negation.

169. **Consistency:** Being *consistent*[168] (throughout).

170. **Constraint:** By a constraint we shall here, in a somewhat narrow sense, understand a property that must be satisfied by certain values of a given type. (That is: The type may define more values than are to be satisfied by the constraint. We also use the terms *data invariant*[196], or *well-formedness*[812]. The term constraint has taken on a larger meaning than propagated in this book. We refer to *constraint programming*, *constraint satisfaction problems*, etc. For a seminal text book we refer to [8]. In constraint programming a constraint, as expressed in a problem model, and hence in a constraint program, is a relation on a sequence of values of (a sequence of) variables of that program.
As you see, the difference, in the two meanings of 'constraint', really, is minor.)

171. **Constructor:** By a constructor we mean either of two, albeit related, things, a type constructor, or a value constructor. By a type constructor we mean an operator on types which when applied to types, say A, constructs another type, say B. By a value constructor we mean a sometimes distributed fix operator which when applied to one or more values constructs a value of a different type. (Examples of type constructors are **-set**, $\times$, *, $^\omega$, $\overrightarrow{m}$, $\rightarrow$, $\overset{\sim}{\rightarrow}$ (sets, Cartesians, finite lists, finite and infinite lists, maps, total functions, partial functions), and mk_B. Examples of value constructors are: $\{\bullet,\bullet,\ldots,\bullet\}$, $(\bullet,\bullet,\ldots,\bullet)$, $\langle\bullet,\bullet,\ldots,\bullet\rangle$, $[\bullet\mapsto\bullet,\bullet\mapsto\bullet,\ldots,\bullet\mapsto\bullet]$ and mk_B$(\bullet,\bullet,\ldots,\bullet)$, etc., (sets, Cartesians, lists, maps, and variant records).)

172. **Context:** There are two related meanings: (i) the parts of a discourse that surround some text and (ii) the interrelated conditions in which something is understood. (The former meaning emphasises *syntactical* properties, i.e., speaks of a syntactic context; the latter, we claim, *semantical* properties (i.e., semantic context). We shall often, by a syntactic context speak of the *scope*[649] of an *identifier*[351]: the text (parts) over which the identifier is defined, i.e., is *bound*[95]. And by a semantic context we then speak of the *environment*[275] in which an *identifier*[351] is *bound*[95] to its semantic meaning. As such semantic contexts go, hand-in-hand, in *configuration*[160]s, with *state*[705]s.)

173. **Context-Free:** By context-free we mean that something is defined free of any considerations of the *context*[172] in which that "something" (otherwise) occurs. (We shall use the context-free concept extensively: *context-free grammar*[175] and *context-free syntax*[176], etc. The *type definition*[785] *rule*[638]s of RSL have a context-free interpretation.)

174. **Context-Free language:** By a context-free language we mean a *language*[417] which can be generated by a *context-free syntax*[176]. (See *generator*[322].)

175. **Context-Free Grammar:** See *context-free syntax*.

176. **Context-Free Syntax:** By a context-free syntax we shall understand a type system consisting of type definitions in which right-hand-side occurrences of defined *type name*[787]s can be freely substituted for any of a variety of their definitions. (Typically a *BNF Grammar*[92] specifies a context-free syntax.)

177. **Context-Sensitive Grammar:** See *context-sensitive syntax*.

178. **Context-Sensitive Syntax:** By a context-sensitive syntax we may understand a type system consisting of ordinary type definitions in which right-hand-side occurrences of defined *type name*[787]s cannot be freely substituted for any of a variety of their definitions, but may only be substituted provided these right-hand-side type names (i.e., *nonterminal*[484]s) occur in specified contexts (of other type names or *literal*[429]s). (Usually a context-sensitive syntax can be specified by a set of rules where both left-hand and right-hand sides are composite type expressions. The left-hand-side composite expression then specifies the contexts in which the right-hand side may be substituted.)

179. **Continuation:** By a continuation we shall, rather technically, understand a state-to-state transformation function, specifically one that is the denotation of a *program point*[548], that is, of any computation as from that program point (i.e., *label*[410]) onwards — until program *termination*[751].

180. **Continuous:** Of a mathematical curve, i.e., function: 'Having the property that the absolute value of the numerical difference between the value at a given point and the value at any point in a neighborhood of the given point can be made as close to zero as desired by choosing the neighborhood small enough' [213].

181. **Contract:** A contract is a *script*[651] specifically expressing a legally binding agreement between two or more parties — hence a document describing the conditions of the contract; a contract is business arrangement for the supply of goods or services at fixed prices, times and locations. In software development a contract specifies what is to be developed (a *domain description*[243], a *requirements prescription*[615], or a *software design*[688]), how it might, or must be developed, criteria for acceptance of what has been developed, delivery dates for the developed items, who the "parties" to the contract are: the *client*[116] and the *developer*[227], etc.

    A legally binding agreement between two or more parties — hence a document describing the conditions of the contract.

    In domains a contract is a set of *rule*[638]s and *regulation*[595]s.

182. **Control:** To control has two meanings: to check, test or verify by evidence or experiments, and to exercise restraining or directing influence over, to regulate. (We shall mostly mean the second form. And we shall often use the term 'control' in conjunction with the term '*monitor*[466]ing'.)

183. **Controller:** By a controller we here mean a *computing system*[151], which interfaces with some physical environment, a *reactive*[578] system, i.e., a plant, and which, by temporally sensing (i.e., sampling) characteristic values of that plant, and by similarly regularly activating *actuator*[17]s in the plant, can make the plant behave according to desired prescriptions. (We stress the reactive system nature of the plant to be controlled. See also *sensor*[659].)

184. **Conversion:** By conversion we shall here, in a rather limiting sense, with a base in the *Lambda-calculus*[412], understand either an *Alpha-renaming*[35] or a *Beta-reduction*[84] of some *Lambda-expression*[414]. (We refer to Chap. 7.)

185. **Correct:** See next entry: *correctness*[186].

186. **Correctness:** Correctness is a relation between two specifications A and B: B is correct with respect to A if every property of what is specified in A is a property of B. (Compare to *conformance*[161] and *congruence*[163].)

187. **Corrective maintenance:** By corrective maintenance we understand a change, predicated by a specification A, to a specification, B', resulting in a specification, B'', such that B'' satisfies more properties of A than does B'. (That is: Specification B' is in error in that it is not *correct*[185] with respect to A. But B'' is an improvement over B'. Hopefully B'' is then correct wrt. A. We also refer to *adaptive maintenance*[21], *perfective maintenance*[519], and *preventive maintenance*[541].)

188. **CSP:** Abbreviation for Communicating Sequential Processes. (See [130, 202] and Chap. 21. Also, but not in this book, a term that covers constraint satisfaction problem (or programming).)

189. **Curry:** Name of American mathematician: Haskell B. Curry. Also a verb: to Curry — see *Currying*[191].

190. **Curried:** A *function invocation*[317], commonly written $f(a_1, a_2, ..., a_n)$, is said to be Curried when instead written: $f(a_1)(a_2)...(a_n)$. (The act of rewriting a function invocation into Curried form is called *Currying*[191].)

191. **Currying:** A *function signature*[318], normally written, f: A×B×...×C→D can be Curried into being written f: A→B→...→C→D. The act of doing so is called Currying.

192. **Customer:** By a customer we mean either of three things: (i) the *client*[116], a person, or a company, which orders the development of some software, or (ii) a *client*[116] *process*[544] or a *behaviour*[79] which *interact*[391]s with another process or behaviour (i.e., the *server*[663]), in order to have that server perform some *action*[12]s on behalf of the client, or (iii) a user of some software (i.e., computing system). (We shall normally use the term customer in the third sense (iii).)

. . . . $\mathcal{D}$

193. **Data:** Data is formalised representation of information. (In our context information is what we may know, informally, and even express, in words, or informal text or diagrams, etc. Data is correspondingly the internal computer, including database representation of such information.)

194. **Data abstraction:** Data abstraction takes place when we abstract from the particular formal representation of data.

195. **Database:** By a database we shall generally understand a large collection of data. More specifically we shall, by a database, imply that the data are organised according to certain data structuring and data *query*[571] and *update*[794] principles. (Classically, three forms of (data structured) databases can be identified: The *hierarchical*[339], the *network*[478], and the *relation*[599]al database forms. We refer to [75, 76] for seminal coverage, and to [29, 28, 54, 55] for formalisation, of these database forms.)

196. **Data invariant:** By a *data*[193] invariant is understood some property that is expected to hold for all instances of the data. (We use the term 'data' colloquially, and really should say type invariance, or variable content invariance. Then 'instances' can be equated with values. See also *constraint*[170].)

197. **Data refinement:** Data refinement is a relation. It holds between a pair of data if one can be said to be a "more concrete" implementation of the other. (The whole point of *data abstraction*[194], in earlier *phase*[523]s, *stage*[702]s and *step*[711]s of *development*[228], is that we can later concretise, i.e., data refine.)

198. **Data reification:** Same as *data refinement*[197]. (To reify is to render something abstract as a material or concrete thing.)

199. **Data structure:** By a data structure we shall normally understand a composition of *data*[193] *value*[802]s, for example, in the "believed" form of a linked *list*[428], a *tree*[777], a *graph*[327] or the like. (As in contrast to an *information structure*[374], a data structure (by our using the term *data*[193]) is bound to some computer representation.)

200. **Data type:** By a *data*[193] *type*[782] is understood a set of *value*[802]s and a set of *function*[310]s over these values — whether *abstract*[1] or *concrete*[154].

201. **Declaration:** A declaration prescribes the allocation of a resource of the kind declared: (i) A variable, i.e., a location in some storage; (ii) a channel between active processes; (iii) an object, i.e., a process possessing a local state; etc.

202. **Decidable:** A formal logic system is decidable if there is an *algorithm*[31] which prescribes *computation*[144]s that can determine whether any given sentence in the system is a theorem.

203. **Decomposition:** By a decomposition is meant the presentation of the parts of a *composite*[133] "thing".

204. **Deduce:** To perform a *deduction*[205], see next. (Cf. *infer*[368].)

205. **Deduction:** A form of reasoning where a conclusion about particulars follows from general premises. (Thus deduction goes from the general (case) to the specific (case). See contrast to *induction*[364]: inferring from specific cases to general cases.)

206. **Deduction rule:** A *rule*[638] for performing *deduction*[205]s.

207. **Definiendum:** The left-hand side of a *definition*[210], that which is to be defined.

208. **Definiens:** The right-hand side of a *definition*[210], that which is defining "something".

209. **Definite:** Something which has specified limits. (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

210. **Definition:** A definition defines something, makes it conceptually "manifest". A definition consists of two parts: a *definiendum*[207], normally considered the left-hand part of a definition, and a *definiens*[208], normally considered the right-hand part (the body) of a definition.

211. **Definition set:** By a definition set we mean, given a *function*[310], the set of *value*[802]s for which the function is defined, i.e., for which, when it is *applied* to a member of the definition set yields a proper value. (Cf., *range set*[577].)

212. **Delimiter:** A delimiter delimits something: marks the start, and/or end of that thing. (A delimiter thus is a syntactic notion.)

213. **Denotation:** A direct specific meaning as distinct from an implied or associated idea [213]. (By a denotation we shall, in our context, associate the idea of mathematical functions: That is, of the *denotational semantics*[215] standing for functions.)

214. **Denotational:** Being a *denotation*[213].

215. **Denotational semantics:** By a denotational semantics we mean a *semantics*[655] which to *atomic*[63] syntactical notions associate simple mathematical structures (usually *function*[310]s, or *set*[664]s of *trace*[766]s, or *algebra*[26]s), and which to *composite*[133] syntactical notions prescribe a semantics which is the *functional*[312] *composition*[140] of the denotational semantics of the *composition*[140] parts.

216. **Denote:** Designates a mathematical meaning according to the principles of *denotational semantics*[215]. (Sometimes we use the looser term designate.)

217. **Dependability:** Dependability is defined as the property of a *machine*[436] such that reliance can justifiably be placed on the service it delivers [196]. (See definition of the related terms: *error*[278], *failure*[286], *fault*[287] and *machine service*[439].)

218. **Dependability requirements:** By *requirements*[605] concerning dependability we mean any such requirements which deal with either *accessibility*[8] requirements, or *availability*[74] requirements, or *integrity*[388] requirements, or *reliability*[601] requirements, or *robustness*[631] requirements, or *safety*[642] requirements, or *security* requirements.

219. **Describe:** To describe something is to create, in the mind of the reader, a *model*[460] of that something. The thing, to be describable, must be either a physically manifest *phenomenon*[524], or a concept derived from such phenomena. Furthermore, to be describable it must be possible to create, to formulate a mathematical, i.e., a formal description of that something. (This delineation of description is narrow. It is too narrow for, for example, philosophical or literary, or historical, or psychological discourse. But it is probably too wide for a *software engineering*[693], or a *computing science*[150] discourse. See also *description*[220].)

220. **Description:** By a description is, in our context, meant some text which designates something, i.e., for which, eventually, a mathematical *model*[460] can be established. (We readily accept that our characterisation of the term 'description' is narrow. That is: We take as a guiding principle, as a dogma, that an informal text, a *rough sketch*[634], a *narrative*[476], is not a description unless one can eventually demonstrate a mathematical model that somehow relates to, i.e., "models" that informal text. To further paraphrase our concern about "describability", we now state that a description is a description of the *entities*, *function*[310]s, *event*[281]s and *behaviour*[79]s of a further designated universe of discourse: That is, a description of a *domain*[239], a *prescription*[540] of *requirements*[605], or a *specification*[698] of a *software design*[688].)

221. **Design:** By a design we mean the *specification*[698] of a *concrete*[154] *artefact*[55], something that can either be physically manifested, like a chair, or conceptually demonstrated, like a software program.

222. **Designate:** To designate is to present a reference to, to point out, something. (See also *denote*[216] and *designation*[223].)

223. **Designation:** The relation between a *syntactic* marker and the semantic thing signified. (See also *denote*[216] and *designate*[222].)

224. **Destructor:** By a destructor we shall here understand a *function*[310] which applies to a *composite*[133] *value*[802] and yields a further specified part (i.e., a subpart) of that value. (Examples of destructors in RSL are the list indexing function, and the selector functions of a variant record. They do not destroy anything, however.)

225. **Determinate:** ()

226. **Deterministic:** In a narrow sense we shall say that a behaviour, a process, a set of actions, is deterministic if the outcome of the behaviour, etc., can be predicted: Is always the same given the same "starting conditions", i.e., the same initial *configuration*[160] (from which the behaviour, etc., proceeds). (See also *nondeterministic*[481].)

227. **Developer:** The person, or the company, which constructs an *artefact*[55], as here, a *domain description*[243], or a *requirements prescription*[615], or a *software design*[688].

228. **Development:** The set of actions that are carried out in order to construct an *artefact*[55].

229. **Diagram:** A usually two-dimensional drawing, a figure. (Sometimes a diagram is annotated with informal and *formal*[296] text.)

230. **Dialogue:** A "conversation" between two *agent*[24]s (men or machines). (We thus speak of man-machine dialogues as carried out over *CHI*[112]s (*HCI*[334]s).)

231. **Didactics:** Systematic instruction based on a clear conceptualisation of the bases, of the foundations, upon which what is being instructed rests. (One may speak of the didactics of a field of knowledge, such as, for example, software engineering. We believe that the present three volume book represents such a clearly conceptualised didactics, i.e., a foundationally consistent and complete basis.)

232. **Directed graph:** A directed graph is a *graph*[327] all of whose *edge*[262]s are directed, i.e., are *arrow*[54]s.

233. **Directory:** A collection of directions. (We shall here take the more limited view of a directory as being a list of names of, i.e., references to *resource*[620]s.)

234. **Discharge:** We use the term discharge in a very narrow sense, namely that of discharging a proof obligation, i.e., by carrying out a proof.

235. **Discrete:** As opposed to *continuous*[180]: consisting of distinct or unconnected elements [213].

236. **Disjunction:** Being separated, being disjoined, decomposed. (We shall mostly think of disjunction as the (meaning of the) logical connective "or": $\vee$.)

237. **Document:** By a document is meant any text, whether informal or *formal*[296], whether *informative*, *descriptive* (or *prescriptive*) or *analytic*[40]. (Descriptive documents may be *rough sketch*[634]es, *terminologies*, *narrative*[476]s, or *formal*[296]. Informative documents are not *descriptive*. Analytic documents "describe" relations between documents, *verification*[807] and *validation*[800], or describe properties of a document.)

238. **Documentation requirements:** By documentation requirements we mean requirements which state which kinds of documents shall make up the deliverable, what these documents shall contain and how they express what they contain.

239. **Domain:** Same as *application domain*[46]; hence see that term for a characterisation. (The term domain is the preferred term.)

240. **Domain acquisition:** The act of acquiring, of gathering, *domain knowledge*[254], and of analysing and recording this knowledge.

241. **Domain analysis:** The act of analysing recorded *domain knowledge*[254] in search of (common) properties of phenomena, or relating what may be considered separate phenomena.

242. **Domain capture:** The act of gathering *domain knowledge*[254], of collecting it — usually from domain *stakeholder*[703]s.

243. **Domain description:** A textual, informal or formal document which describes a domain **as it is**. (Usually a domain description is a set of documents with many parts recording many facets of the domain: The *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445], *rules and regulations*[640], and the *human behaviour*[345]s.)

244. **Domain description unit:** By a domain description unit we understand a short, "one- or two-liner", possibly *rough-sketch*[633] *description*[220] of some property of a *domain*[239] *phenomenon*[524], i.e., some property of an *entity*[272], some property of a *function*[310], of an *event*[281], or some property of a *behaviour*[79]. (Usually domain description units are the smallest textual, sentential fragments elicited from domain *stakeholder*[703]s.)

245. **Domain determination:** Domain determination is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] cum *requirements prescription*[615]. Any *nondeterminism*[482] expressed by either of these specifications which is not desirable for some required software design must be made deterministic (by this *requirements engineer*[612] performed operation). Other domain requirements facets are: *domain projection*[255], *domain instantiation*[253], *domain extension*[249] and *domain fitting*[251].

246. **Domain development:** By domain development we shall understand the *development*[228] of a *domain description*[243]. (All aspects are included in development: *domain acquisition*[240], domain *analysis*[39], domain *model*[460]ling, domain *validation*[800] and domain *verification*[807].)

247. **Domain engineer:** A domain engineer is a *software engineer*[692] who performs *domain engineering*[248]. (Other forms of *software engineer*[692]s are: *requirements engineer*[612]s and *software design*[688]ers (cum *programmer*[547]s).)

248. **Domain engineering:** The engineering of the development of a *domain description*[243], from identification of *domain*[239] *stakeholder*[703]s, via *domain acquisition*[240], *domain analysis*[241] and *domain description*[243] to *domain validation*[256] and *domain verification*[257].

249. **Domain extension:** Domain extension is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] or a *requirements prescription*[615]. It effectively extends a *domain description*[243] by entities, functions, events and/or behaviours conceptually possible, but not necessarily humanly or technologically feasible in the domain (as it was). Other domain requirements facets are: *domain projection*[255], *domain determination*[245], *domain instantiation*[253] and *domain fitting*[251].

250. **Domain facet:** By a domain facet we understand one amongst a finite set of generic ways of analysing a domain: A view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain. (We consider here the following domain facets: *business process*[99], *intrinsics*[399], *support technology*[725], *management and organisation*[445], *rules and regulations*[640], and *human behaviour*[345].)

251. **Domain fitting:** By *domain requirements fitting* we understand an operation which takes n domain requirements prescriptions, d_{r_i}, that are claimed to share m independent sets of tightly related sets of simple entities, actions, events and/or behaviours and map these into $n+m$ domain requirements prescriptions, δ_{r_j}, where m of these, $\delta_{r_{n+k}}$ capture the shared phenomena and concepts and the other n prescriptions, δ_{r_ℓ}, are like the n "input" domain requirements prescriptions, d_{r_i}, except that they now, instead of the "more-or-less" shared prescriptions, that are now consolidated in $\delta_{r_{n+k}}$, prescribe interfaces between δ_{r_i} and $\delta_{r_{n+k}}$ for $i : \{1..n\}$. Other domain requirements facets are: *domain projection*[255], *domain determination*[245], *domain instantiation*[253] and *domain extension*[249].

252. **Domain initialisation:** Domain initialisation is an *interface requirements facet*[395]. It is an operation performed on a *requirements prescription*[615]. For an explanation see *shared data initialisation*[670] (its 'equivalent'). Other *interface requirements facet*[395]s are: *shared data refreshment*[672], *computational data+control*[146], *man-machine dialogue*[446], *man-machine physiological*[448] and *machine-machine dialogue*[437] *requirements*[605].

253. **Domain instantiation:** Domain instantiation is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] (cum *requirements prescription*[615]). Where, in a domain description certain *entities* and *function*[310]s are left undefined, domain instantiation means that these entities or functions are now instantiated into constant *value*[802]s. Other requirements facets are: *domain projection*[255], *domain determination*[245], *domain extension*[249] and *domain fitting*[251].

254. **Domain knowledge:** By domain knowledge we mean that which a particular group of people, all basically engaged in the "same kind of activities", know about that domain of activity, and what they believe that other people know and believe about the same domain. (We shall, in our context, strictly limit ourselves to "knowledge", staying short of "beliefs", and we shall similarly strictly limit ourselves to assume just one "actual" world, not any number of "possible" worlds. More specifically, we

shall strictly limit our treatment of domain knowledge to stay clear of the (albeit very exciting) area of reasoning about knowledge and belief between people (and agents) [127, 106].)

255. **Domain projection:** Domain projection is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] cum *requirements prescription*[615]. The operation basically "removes" from a description definitions of those *entities* (including their *type definition*[785]s), *functions*, *events* and *behaviours* that are not to be considered in the *requirements*[605]. The removed phenomena and concepts are thus projected "away". Other domain requirements facets are: *domain determination*[245], *domain instantiation*[253], *domain extension*[249] and *domain fitting*[251].

256. **Domain validation:** By domain validation we rather mean: '*validation*[800] of a domain description', and by that we mean the informal assurance that a description purported to cover the *entities*, *function*[310]s, *event*[281]s and *behaviour*[79]s of a further designated domain indeed does cover that domain in a reasonably representative manner. (Domain validation is, necessarily, an informal activity: It basically involves a guided reading of a domain description (being validated) by *stakeholder*[703]s of the domain, and ends in an evaluation report written by these domain *stakeholder*[703] readers.)

257. **Domain verification:** By domain verification we mean *verification*[807] of claimed properties of a domain description, and by that we mean the formal assurance that a description indeed does possess those claimed properties. (The usual principles, techniques and tools of verification apply here.)

258. **Domain requirements:** By domain *requirements*[605] we understand such requirements — save those of *business process reengineering*[101] — which can be expressed sôlely by using professional terms of the *domain*[239]. (Domain requirements constitute one requirements *facet*[285]. Others requirements facets are: *business process reengineering*[101], *interface requirements*[394] and *machine requirements*[438].)

259. **Domain requirements facet:** By *domain requirements*[258] facets we understand such domain requirements that basically arise from either of the following operations on *domain description*[243]s (cum *requirements prescription*[615]s): *domain projection*[255], *domain determination*[245], *domain extension*[249], *domain instantiation*[253] and *domain fitting*[251].

260. **Dynamic:** An *entity*[272] is said to be dynamic if its value changes over time, i.e., it is subjected, somehow, to actions. (We distinguish three kinds of dynamic entities: *inert*[367], *active*[14] and *reactive*[578]. This is in contrast to *static*[708].)

261. **Dynamic typing:** Enforcement of *type checking* at *run time*[641]. (A language is said to be dynamically typed if it is not *statically typed*.)

. . . . $\mathcal{E}$

262. **Edge:** A line, a connection, between two *node*[479]s of a *graph*[327] or a *tree*[777]. (Other terms for the same idea are: *arc*[50] and *branch*[97].)

263. **Elaborate:** See next: *elaboration*[264].

264. **Elaboration:** The three terms *elaboration*, *evaluation*[280] and *interpretation*[397] essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*[160], or as a function from configurations to *value*[802]s. Given that configuration typically consists of *static*[708] *environment*[275]s and *dynamic*[260] *state*[705]s (or *storage*[715]s), we use the term elaboration in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to pairs of states and values.

265. **Elicitation:** To elicit, to extract. (See also: *acquisition*[11]. We consider elicitation to be part of acquisition. Acquisition is more than elicitation. Elicitation, to us, is primarily the act of extracting information, i.e., knowledge. Acquisition is that plus more: Namely the preparation of what and how to elicit and the postprocessing of that which has been elicited — in preparation of proper analysis. Elicitation applies both to domain and to requirements elicitation.)

266. **Embedded:** Being an integral part of something else. (When something is embedded in something else, then that something else is said to surround the embedded thing.)

267. **Embedded system:** A *system*[736] which is an integral part of a larger system. (We shall use the term embedded system primarily in the context of the larger, 'surrounding' system being *reactive*[578] and/or *hard real time*[330].)

268. **Endomorphism:** A *homomorphism*[343] that maps an algebra into itself is an endomorphism. (See also *automorphism*[72], *epimorphism*[276], *isomorphism*[404], *monomorphism*[467].)

269. **Engineer:** An engineer is a person who "walks the bridge" between science and technology: (i) Constructing, i.e., designing, *technology*[746] based on scientific insight, and (ii) analysing technology for its possible scientific content.

270. **Engineering:** Engineering is the design of *technology*[746] based on scientific insight, and the analysis of technology for its possible scientific content. (In the context of this glossary we single out three forms of engineering: *domain engineering*[248], *requirements engineering*[613] and *software design*[688]; together we call them *software engineering*[693]. The technology constructed by the *domain engineer*[247] is a *domain description*[243]. The technology constructed by the *requirements engineer*[612] is a *requirements prescription*[615]. The technology constructed by the *software design*[688]er is *software*[685].)

271. **Enrichment:** The addition of a property to something already existing. (We shall use the term enrich in connection with a collection (i.e., a RSL **scheme** or a RSL

class) — of definitions, declaration and axioms — being '**extend**ed **with**' further such definitions, declaration and axioms.)

272. **Entity:** By an entity we shall understand either a *simple entity*[681], an *action*[12], an *event*[281] or a *behaviour*[79].

273. **Enumerable:** By enumerable we mean that a set of elements satisfies a *proposition*[560], i.e., can be logically characterised.

274. **Enumeration:** To list, one after another. (We shall use the term enumeration in connection with the syntactic expression of a "small", i.e., definite, number of elements of a(n enumerated) *set*[664], *list*[428] or *map*[449].)

275. **Environment:** A context, that is, in our case (i.e., usage), the ("more static") part of a *configuration*[160] in which some syntactic entity is *elaborated*, *evaluated*, or *interpreted*. (In our "metacontext", i.e., that of software engineering, environments, when deployed in the elaboration (etc.) of, typically, specifications or programs, record, i.e., list, associate, identifiers of the specification or program text with their meaning.)

276. **Epimorphism:** If a *homomorphism*[343] ϕ is a *surjective function*[727] then ϕ is an epimorphism. (See also *automorphism*[72], *endomorphism*[268], *isomorphism*[404], *monomorphism*[467].)

277. **Epistemology:** The study of knowledge. (Contrast, please, to *ontology*[492].)

278. **Error:** An error is an action that produces an incorrect result. An error is that part of a *machine*[436] *state*[705] which is "liable to lead to subsequent failure". An error affecting the *machine service*[439] is an indication that a *failure*[286] occurs or has occurred [196]. (An error is caused by a *fault*[287].)

279. **Evaluate:** See next: *evaluation*[280].

280. **Evaluation:** The three terms *elaboration*, *evaluation*[280] and *interpretation*[397] essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*[160], or as a function from configurations to *value*[802]s. Given that configuration typically consists of *static*[708] *environment*[275]s and *dynamic*[260] *state*[705]s (or *storage*[715]s), we use the term evaluation in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to values.

281. **Event:** Something that occurs instantaneously. (We shall, in our context, take events as being manifested by certain *state*[705] changes, and by certain *interaction*[392]s between *behaviour*[79]s or *process*[544]es. The occurrence of events may "trigger" actions. How the triggering, i.e., the *invocation*[402] of *functions* are brought about is usually left implied, or unspecified. We consider *event*[281]s to be one of the four kinds of

entities[272] that the `Triptych` "repeatedly" considers. The other three are: *simple entities*[681], *action*[12]s and *behaviour*[79]s. Consideration of these are included in the specification of all *domain facet*[250]s and all *requirements facet*[614]s.)

282. **Expression:** An expression, in our context (i.e., that of software engineering), is a syntactical entity which, through *evaluation*[280], designates a *value*[802].

283. **Extension:** We shall here take extension to be the same as *enrichment*[271]. In *domain requirements*[258], when we 'perform' extension, we introduce *entities*[272] (*simple entities*[681], *action*[12]s, *event*[281]s and *behaviour*[79]s) that were not [originally] in the domain [but will now become entities of the domain resulting from implementing the requirements].

284. **Extensional:** Concerned with objective reality [213]. (Please observe a shift here: We do not understand the term extensional as 'relating to, or marked by extension in the above sense, but in contrast to *intensional*[390].)

. . . . $\mathcal{F}$

285. **Facet:** By a facet we understand one amongst a finite set of generic ways of analysing and presenting a *domain*[239], a *requirements*[605] or a *software design*[688]: a view of the universe of discourse, such that the different facets cover conceptually different views, and such that these views together cover that universe of discourse. (Examples of domain facets are *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445], *rules and regulations*[640] and *human behaviour*[345]. Examples of requirements facets are *business process reengineering*[101], *domain requirements*[258], *interface requirements*[394] and *machine requirements*[438]. Examples of software design facets are *software architecture*[687], *component design*[132], *module design*[465], etc.)

286. **Failure:** A *fault*[287] may result in a failure. A *machine*[436] failure occurs when the delivered *machine service*[439] deviates from fulfilling the machine function, the latter being what the machine is aimed at [196]. (A failure is thus something relative to a *specification*[698], and is due to a *fault*[287]. Failures are concerned with such things as *accessibility*[8], *availability*[74], *reliability*[601], *safety*[642] and *security*.)

287. **Fault:** The adjudged (i.e., the 'so judged') or hypothesised cause of an *error*[278] [196]. (An *error*[278] is caused by a fault, i.e., faults cause errors. A software fault is the consequence of a human *error*[278] in the development of that software.)

288. **Finite:** Of a fixed number less than infinity, or of a fixed structure that does not "flow" into perpetuity as would any *information structure*[374] that just goes on and on. (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

289. **Finite state automaton:** By a finite state automaton we understand an *automaton*[71] whose state set is finite. (We shall usually consider only what is known as Moore automata: that is, automata which have some final states.)

290. **Finite state machine:** By a finite state machine we understand an extended *finite state automaton*[289]. The extension amounts simply to the following: Every transition (caused by an input, in a state, to another state) also yields an output. (We shall thus consider only what is known as Mealy machines. The output is intended to designate some action, or some signal, to be considered by an environment of the machine.)

291. **Finite state transducer:** By a finite state transducer we simply mean the same as a finite state machine. (The machine in question is said to transduce, to "translate" any sequence of inputs to some corresponding sequence of outputs.)

292. **First-order:** We say that a *predicate logic*[537] is first order when quantified variables are not allowed to range over functions. (If they range over functions we call the logic a *higher-order*[341] logic [190, 179]. Similar remarks can be made for general first-order functions, respectively higher-order functions.)

293. **Fix point:** The fix point of a function, F, is any value, f, for which $Ff = f$. A function may have any number of fixed points from none (e.g., $Fx = x{+}1$) to infinitely many (e.g., $Fx = x$). The fixed point combinator, written as either "**fix**" or "**Y**" will return the fixed point of a function. (The fix point identity is $\mathbf{Y}F = F(\mathbf{Y}F)$.)

294. **Fitting:** Fitting in the context of requirements engineering is an operation that applies to n (where n is 2 or more) domain requirements descriptions $(d_1, d_2, \ldots, d_n)$ and yields $n+1$ domain requirements descriptions $(d'_1, d'_2, \ldots, d'_n$ and $d_{\text{"shared"}})$ where n of these each, d'_i, cover major parts of respective d_i and where $d_{\text{"shared"}}$ covers what is "somehow" common to $d_1, d_2, \ldots, d_n$.

295. **Flowchart:** A diagram (a chart), for example of circles (input, output), annotated (square) boxes, annotated diamonds and infixed arrows, that shows step by step flow through an algorithm.

296. **Formal:** By formal we shall, in our context (i.e., that of software engineering), mean a language, a system, an argument (a way of reasoning), a program or a specification whose syntax and semantics is based on (rules of) mathematics (including mathematical logic).

297. **Formal definition:** Same as *formal description*[299], *formal prescription*[303] or *formal specification*[304].

298. **Formal development:** Same as the standard meaning of the composition of *formal*[296] and *development*[228]. (We usually speak of a spectrum of development modes: *systematic development*[737], *rigorous development*[629], and formal development. Formal software development, to us, is at the "formalistic" extreme of the three modes of development: Complete *formal specification*[304]s are always constructed, for all (phases and) stages of development; all *proof obligation*[555]s are expressed; and all are discharged (i.e., proved to hold).)

299. **Formal description:** A *formal*[296] *description*[220] of something. (Usually we use the term formal description only in connection with *formalisation*[300] of *domain*[239]s.)

300. **Formalisation:** The act of making a formal specification of something elsewhere informally specified; or the document which results therefrom.

301. **Formal method:** By a formal method we mean a *method*[456] whose techniques and tools[14] are *formal*[296]ly based. (It is common to hear that some notation is claimed to be that of a formal method — where it then turns out that few, if any, of the building blocks of that notation have any formal foundation. This is especially true of many diagrammatic notations. `UML` is a case in point — much is presently being done to formalise subsets of `UML` [180].)

302. **Formal parameter:** By a formal parameter we mean an identification (say a naming and a typing), in a *function definition*[316]'s *function signature*[318], of an argument of the function, a place-holder for *actual argument*[16]s.

303. **Formal prescription:** Same as *formal definition*[297] or *formal specification*[304]. (Usually we use the term formal prescription only in connection with *formalisation*[300] of *requirements*[605].)

304. **Formal specification:** A *formalisation*[300] of something. (Same as *formal definition*[297], *formal description*[299] or *formal prescription*[303]. Usually we use the term formal specification only in connection with *formalisation*[300] of *software design*[688]s.)

305. **Free:** The concept of being free is associated with (i) *identifier*[351]s (i.e., *name*[474]s) and *expression*[282]s, and (ii) with *name*[474]s (i.e., *identifier*[351]s) and *resource*[620]s. An identifier is said to be either *bound*[95] or free in an expression based on certain rules being satisfied or not. If an identifier is free in an expression then nothing is said about what free occurrences of that identifier are bound to. (Cf. *bound*[95].)

306. **Freeing:** The removal of *storage*[715] *location*[431]s, or of *stack activation*[701]s.

307. **Frontier:** The concept of frontier is here associated with *tree*[777]s. Visualise that tree as represented as a flat diagram with no crosses (i.e., intersecting) *branch*[97]es. A frontier of a tree is a reading of the leaves (cf. *leaf*[419]) of the tree in one of the two possible directions, say left to right or right to left. (See *tree traversal*[778].)

308. **FUNARG:** A specification or a programming language is said to enjoy, i.e. possess, the FUNARG property if *value*[802]s of *function invocation*[317]s may be *function*[310]s defined locally to the invoked function. (`LISP` has the FUNARG property. So does SAL, a simple applicative language defined in .)

[14]Tools include specification and programming languages as such, as well as all the software tools relating to these languages (editors, syntax checkers, theorem provers, proof assistants, model checkers, specification and program (flow) analysers, interpreters, compilers, etc.).

309. **Full algebra:** A full *algebra* is a *total algebra*[765].

310. **Function:** By a function we understand something which when *applied* to a *value*[802], called an *argument,* yields a value called a *result.* (Functions can be modelled as sets of (argument, result) pair — in which case applying a function to an argument amounts to "searching" for an appropriate pair. If several such pairs have the same argument (value), the function is said to be *nondeterministic*[481]. If a function is applied to an argument for which there is no appropriate pair, then the function is said to be partial; otherwise it is a total function.)

311. **Function activation:** When, in an operational, i.e., computational ("mechanical") sense, a function is being applied, then some resources have to be set aside in order to carry out, to handle, the application. This is what we shall call a function activation. (Typically a function activation, for conventional *block-structured*[90] languages (like `C#, Java, Standard ML` [125, 207, 119]), is implemented by means (also) of a stack-like data structure: Function invocation then implies the stacking (pushing) of a stack activation on that stack, i.e., the *activation stack*[13] (a circular reference!). Elaboration of the function definition body means that intermediate values are pushed and popped from the topmost activation element, etc., and that completion of the function application means that the top stack activation is popped.)

312. **Functional:** A function whose arguments are allowed themselves to be functions is called a functional. (The *fix point*[293] (finding) function is a functional.)

313. **Functional programming:** By functional programming we mean the same as *applicative programming*[48]: In its barest rendition functional programming involves just three things: definition of functions, functions as ordinary *value*[802]s, and *function application*[315] (i.e., *function invocation*[317]). (Most current functional programming languages (`Haskell, Miranda, Standard ML`) go well beyond just providing the three basic building blocks of functional programming [219, 220, 175].)

314. **Functional programming language:** By a functional programming language we mean a *programming language*[551] whose principal values are functions and whose principal operations on these values are their creation (i.e., definition), their application (i.e., invocation) and their composition. (Functional programming languages of interest today, 2005, are (alphabetically listed): `CAML` [69, 63, 64, 226, 156], `Haskell` [219], `Miranda` [220], `Scheme` [1, 115, 97] and `SML` (Standard ML) [175, 119]. `LISP 1.5` was a first functional programming language [169].)

315. **Function application:** The act of applying a function to an argument is called a function application. (See 'comment' field of *function activation*[311] just above.)

316. **Function definition:** A *function*[310] *definition*[210], as does any definition, consists of a *definiens*[208] and a *definiendum*[207]. The definiens is a *function signature*[318] and the definiendum is a clause, typically an expression. (Cf. *Lambda-function*[415]s.)

317. **Function invocation:** Same as *function application*[315]. (See parenthesized remark of entry 311 (*function activation*[311]).)

318. **Function signature:** By a function signature we mean a text which presents the name of the function, the types of its argument values and the type(s) of its result value(s).

. . . . $\mathcal{G}$

319. **Garbage:** By garbage we shall here understand those (computing) *resource*[620]s which can no longer be referenced. (Usually we restrict our 'garbage' concern to that of *storage*[715] *location*[431]s that can no longer be accessed because there are no references to them.)

320. **Garbage collector:** To speak of garbage collection we must first introduce the notions of allocatable *storage*[715], i.e., storage — what shall be known as free, i.e., unallocated — *location*[431]s (including those that can be considered *garbage*[319]). By a garbage collector we shall here understand a device, a software program or a hardware mechanism which "returns" to a set of free locations that can subsequently be made available for *allocation*[33].

321. **Generate:** By generate we shall understand that which can be associated both with a *grammar*[325] and with an *automaton*[71]: namely a *language*[417], i.e., a set of strings. Either accepted as *input*[382] to a *finite state automaton*[289], or *denote*[216]d by a *grammar*[325]. (Acceptance by an automaton means that the automaton is started in an initial state and upon completion of reading the input is in a final state. Generation by a grammar means the recursive (i.e., repeated) *substitution*[722] of *nonterminal*[484]s of a grammar *rule*[638] left-hand side with the left-hand sides of the rules whose right-hand side is the substituted nonterminal.)

322. **Generator:** A generator is a concept: It can be thought of as a device, i.e., a software program or a machine mechanism, which outputs typically sequences of structures — typically symbols. (A *BNF Grammar*[92] can thus be said to generate the (usually infinite) set of strings, i.e., of sentence of the designated language. A *finite state machine*[290] can likewise be said to be a generator: Upon being presented with any input string it generates an output string (a transduction).)

323. **Generator function:** To speak of a generator function we need first introduce the concept of a *sort*[694] "of interest". A generator function is a function which when applied to arguments of some kind, i.e., types, yields a value of the type of the sort "of interest". (Typically the sort "of interest" can be thought of as the state (a stack, a queue, etc.).)

324. **Glossary:** According to [159] a *gloss* is "a word inserted between the lines or in the margin as an explanatory rendering of a word in the text; hence a similar rendering in a glossary or dictionary. Also, a comment, explanation, interpretation."

Furthermore according to [159] a *glossary* is therefore "a collection of glosses, a list with explanations of abstruse, antiquated, dialectical, or technical terms; a partial dictionary."

325. **Grammar:** See *syntax*, in general, or *regular syntax*, *context-free syntax*, *context-sensitive syntax* and *BNF* in specific.

326. **Grand state:** "Grand state" is a colloquial term. It is meant to have the same meaning as *configuration*[160]. (The colloquialism is used in the context of, for example, praising a software engineer as "being one who really knows how to design the grand state for some universe of discourse" being specified.)

327. **Graph:** By a graph we shall here mean the term as usually used in the discrete mathematics discipline of graph theory: as a (usually, but not necessarily finite) set of *node*[479]s (*vertexes*), some of which may be connected by (one or more) *arc*[50]s (*edge*[262]s, lines). (A graph edge defines a *path*[517] of length one. If there is a path from one node to another, and from that other node to yet a third node, then the graph, by transitivity, defines a path from the first to the third node, etc. A graph can be either an *acyclic graph*[19] (no path "cycles back") or a *cyclic graph*, a *directed graph*[232] (edges are one-directional arrows) or an *undirected graph* [20, 21, 182, 122].)

328. **Ground term:** A ground term is either an *identifier*[351] or a *value*[802] *literal*[429]. (The identifier is then assumed to be bound to a value. The value literal typically is an alphanumeric string designating, for example, an integer, a real, a truth value, a character, etc.)

329. **Grouping:** By grouping we mean the ordered, finite collection, into a *Cartesian*[107], of mathematical structures (i.e., *value*[802]s).

. . . . $\mathcal{H}$

330. **Hard real time:** By hard real time we mean a *real time*[580] property where the exact, i.e., absolute timing, or time interval, is of essence. (Thus, if a system is said to enjoy, or must possess, a certain real time property, for example, (i) the system must emit a certain signal on the 11th of December 2015 at 17:20:30 hours[15], or (ii) that a response signal must be issued after an interval of exactly 1234 days, 5 hours, 6 minutes, and 7 seconds plus/minus 8 microseconds (from when an initiating signal was received), then it is hard real time. Cf. *soft real time*[684].)

331. **Hardware:** By hardware is meant the physical embodiment of a computer: its electronics, its boards, the racks, cables, button, lamps, etc.

332. **Hazard:** A hazard is a source of danger.

[15]That time is when the current author hopes to celebrate the exact hour of his anniversary of 50 years of marriage to Kari Skallerud!

333. **Hazard analysis:** Hazard analysis is a process used to determine how a device can cause hazards to occur and then reducing the risks to an acceptable level. (The process consists of: (1) the developer of the system determining what could go wrong with the device, (2) determining how the effects of the failure can be mitigated, and (3) implementing and testing mitigations.)

334. **HCI:** Abbreviation for human computer interface. (Same as *CHI*[112], and same as *man-machine*[446] interface.)

335. **Heap:** By a heap is here meant an unordered, finite collection, i.e., a set, of *storage*[715] *location*[431]s, such that each of these locations can be said to be allocated (for some purpose), and such that a freeing, i.e., deallocation, of these locations usually does not follow the inverse order of their allocation. (Thus a heap works in contrast to an *activation stack*[13] — complementary, so to speak! Typically a *garbage collector*[320] is involved in helping to secure locations on the heap available for allocation.)

336. **Heterogeneous algebra:** A heterogeneous *algebra* is an algebra whose carrier A is an indexed set of carriers: A_1, A_2,..., A_m, and whose functions, $\phi_{i_n} : \Phi$, or *arity* n, are of *type*[782]: $A_{i_1} \times A_{i_2} \times \cdots \times A_{i_n} \rightarrow A_j$ where i_k, for all $k \in \{1, \ldots, n\}$, are in the set $\{1, 2, \ldots, m\}$.

337. **Hiding:** Hiding is a concept related to *module*[464]s. In fact, it is a main purpose of syntactically providing the module mechanism. You have, somewhat mechanistically, to imagine a group of (developers of) modules. One module mentions (i.e., uses), say, functions defined in other modules. But those other modules, besides, in order to define those "exported" functions, define auxiliary functions (types, etc.) that "reveal" details of implementation which it is not necessary to divulge. (One may wish, later, in "the life of that module", to change those implementation decisions.) Hence, by syntactic means, such as, for example, *export*, *import* and *hide* clauses, the developer requests the module compiling system to statically (or otherwise) secure that other modules cannot "inspect" those auxiliary functions, types, etc. (We refer to [186, 185, 189, 188, 187]. Parnas must be credited, among others, for having skillfully propagated the hiding concept.)

338. **Hierarchy:** By a hierarchy we understand a conceptual decomposition of resources into what can be "pictured" as a *tree*[777]-like structure (and where the emphasis is on the root of the structure).

339. **Hierarchical:** By something being hierarchical we mean that that something forms a *hierarchy*[338]. (See also *compositional*[141].)

340. **Hierarchical documentation:** By hierarchical documentation we mean a development, or a presentation (of that development), of, as here, some *description*[220] (*prescription*[540] or *specification*[698]), in which a notion of "largest", overall, phenomena and concepts are developed (resp. presented) first, then their decompositions into

component phenomena and concepts, etc., until some notion of atomic, i.e., "smallest" development (etc.) has been achieved. (See also *hierarchy*[338] (just above) and *compositional documentation*[142].)

341. **Higher-order:** A *functional*[312] or a *value*[802] whose *definition set*[211] or *range set*[577] values are *function*[310]s. (See, in contrast, *first-order*[292].)

342. **Homeomorphism:** A function that is a one-to-one mapping between sets such that both the function and its inverse are continuous. (Not to be confused with *homomorphism*[343].)

343. **Homomorphism:** A *function*[310], $\phi : A \to A'$, from values of the carrier A of one *algebra*[26] (A, Ω) to values of the carrier A' of another algebra (A', Ω') is said to be a homomorphism (same as a morphism) from (A, Ω) to (A', Ω'), if for any $\omega : \Omega$ and for any $a_i : A$, there is a corresponding $\omega' : \Omega'$ such that: $\phi(\omega(a_1, a_2, ..., a_n)) = \omega'(\phi(a_1), \phi(a_2), ..., \phi(a_n))$. (See also *automorphism*[72], *endomorphism*[268], *epimorphism*[276], *isomorphism*[404] and *monomorphism*[467].)

344. **Homomorphic principle:** The homomorphic principle advises the software engineer to formulate *function definition*[316]s such that they express a *homomorphism*[343]. (It is a basic tenet of a *denotational semantics*[215] *definition*[210] that it is expressed as a *homomorphism*[343].)

345. **Human behaviour:** By human behaviour we shall here understand the way a human follows the enterprise *rules and regulations*[640] as well as interacts with a *machine*[436]: dutifully honouring specified (machine *dialogue*[230]) *protocol*[561]s, or negligently so, or sloppily not quite so, or even criminally not so! (Human behaviour is a *facet*[285] of the *domain*[239] (of the enterprise). We shall thus model human behaviour also in terms of it failing to react properly, i.e., humans as *nondeterministic*[481] *agent*[24]s! Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445], and *rules and regulations*[640].)

346. **Hybrid:** Something heterogeneous, something (as a computing device) that has two different types of components (*software*[685], respectively hardware, the latter including, besides the digital computer, also *controller*[183]s (*sensor*[659]s, *actuator*[17]s)) performing essentially the same function by cooperating on computing *"that same"* function. (Typically we speak of, i.e., deploy hybridicity when *monitor*[466]ing and *control*[182]ling *reactive system*[579]s — but then hybridicity additionally, to us, means a combination in which the *controller*[183] handles analog matters of continuity, and the *software*[685] plus computer handles discrete matters. Finally, for a conventional analogue *controller*[183] there is usually but one "decision mode". With the software-directed computing system there is now the possibility of multiple discrete + continuous *controller*[183] "regimes".)

347. **Hypothesis:** An assumption made for the sake of argument.

. . . . $\mathcal{I}$

348. **Icon:** A pictorial representation, an image, a sign whose form (shape, etc.) suggests its meaning. (A graphic symbol on a computer display screen which suggests the purpose of an available *function*[310] or *value*[802] which *designate*[222]s that *entity*[272].)

349. **Iconic:** Adjective form of *icon*[348].

350. **Identification:** The pointing out of a relation, an association, between an *identifier*[351] and that "thing", that *phenomenon*[524], it *designate*[222]s, i.e., it stands for or identifies.

351. **Identifier:** A name. (Usually represented by a string of alphanumeric characters, sometimes with properly infixed "-"s or "_"s.)

352. **Imperative:** Expressive of a command [213]. (We take imperative to more specifically be a reflection of *do this, then do that.* That is, of the use of a *state*[705]-based programming approach, i.e., of the use of an *imperative programming language*[354]. See also *indicative*[362], *optative*[499], and *putative*[566].)

353. **Imperative programming:** Programming, *imperative*[352]ly, "with" references to *storage*[715] *location*[431]s and the updates of those, i.e., of *state*[705]s. (Imperative programming seems to be the classical, first way of programming digital computers.)

354. **Imperative programming language:** A programming language which, significantly, offers language constructs for the creation and manipulation of variables, i.e., *storage*[715]s and their *location*[431]s. (Typical imperative programming languages were, in "ye olde days", `Fortran, Cobol, Algol 60, PL/I, Pascal, C,` etc. [166, 164, 12, 165, 12, 142]. Today programming languages like `C++, Java, C#,` etc. [215, 207, 125] additionally offer *module*[464] cum *object*[487] "features".)

355. **Implementation:** By an implementation we understand a computer program that is made suitable for *compilation*[124] or *interpretation*[397] by a *machine*[436]. (See next entry: *implementation relation*[356].)

356. **Implementation relation:** By an *implementation*[355] relation we understand a logical relation of *correctness*[186] between a *software design specification*[689] and an *implementation*[355] (i.e., a computer program made suitable for *compilation*[124] or *interpretation*[397] by a *machine*[436]).

357. **Incarnation:** A particular instance of a value, usually a state. (We shall here use the term incarnation to designate any one activation on an *activation stack*[13] — where such an incarnation, i.e., activation, represents a program *block*[89] or *function*[310] (or procedure, or *subroutine*[723]) *invocation*[402].)

358. **Incomplete:** We say that a *proof system*[557] is incomplete if not all true sentences are provable.

359. **Incompleteness:** Noun form of the *incomplete*[358] adjective.

360. **Inconsistent:** A set of *axiom*[75]s is said to be inconsistent if, by means of these, and some *deduction rule*[206]s, one can *prove* a property and its negation.

361. **Indefinite:** Not definite, i.e., of a fixed number or a specific property, but it is not known, at the point of uttering the term 'indefinite', what that number or property is. (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

362. **Indicative:** Stating an objective fact. (See also *imperative*[352], *optative*[499] and *putative*[566].)

363. **Induce:** The use of *induction*[364]. (To conclude a general property from special cases.)

364. **Induction:** Inference of a general property from particular instances. (On the basis of several, "similar" cases one may infer a general, say, principle or property. In contrast to *deduction*[205]: from general (e.g., from laws) to specific instances.)

365. **Inductive:** The use of *induction*[364].

366. **In extension:** A concept of logic. In extension is a correlative word that indicates the reference of a term or concept. (When we speak of functions in extension, we shall therefore mean it in the sense of presenting "all details", the "inner workings" of that function. Contrast to *in intension*[378].)

367. **Inert:** A *dynamic*[260] *phenomenon*[524] is said to be inert if it cannot change *value*[802] of its own volition, i.e., by itself, but only through the *interaction*[392] between that *phenomenon*[524] and a change-instigating *environment*[275]. An inert phenomenon only changes value as the result of external stimuli. These stimuli prescribe exactly which new value they are to change to. (Contrast to *active*[14] and *reactive*[578].)

368. **Infer:** Common term for *deduce*[204] or *induce*[363].

369. **Inference rule:** Same as *deduction rule*[206].

370. **Infinite:** As you would think of it: not finite! (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

371. **Informal:** Not formal! (We normally, by an informal specification mean one which may be precise (i.e., unambiguous, and even concise), but which, for example is expressed in natural, yet (domain specific) professional language — i.e., a language which does not have a precise semantics let alone a formal *proof system*[557]. The `UML` notation is an example of an informal language [180].)

372. **Informatics:** The confluence of (i) *application*s, (ii) *computer science*, (iii) *computing science* [i.e., the art [143, 144, 145] (1968–1973), craft [200] (1981), discipline [82] (1976), logic [123] (1984), practice [124] (1993–2004), and science [114] (1981) of programming], (iv) *software engineering* and (v) *mathematics*.

373. **Information:** The communication or reception of knowledge. (By information we thus mean something which, in contrast to *data*[193], informs us. No computer representation is, let alone any efficiency criteria are, assumed. Data as such does, i.e., bit patterns do, not 'inform' us.)

374. **Information structure:** By an information structure we shall normally understand a composition of more "formally" represented (i.e., structured) *information*[373], for example, in the "believed" form of *table*[739], a *tree*[777], a *graph*[327], etc. (In contrast to *data structure*[199], an information structure does not necessarily have a computer representation, let alone an "efficient" such.)

375. **Informative documentation:** By informative documentation we understand texts which *inform*, but which do not (essentially) describe that which a *development*[228] is to develop. (Informative documentation is balanced by *descriptive* and *analytic*[40] *documentation* to make up the full documentation of a *development*[228].)

376. **Infrastructure:** According to the World Bank: *'Infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spillovers from users to nonusers).* We shall use the term as follows: Infrastructures are concerned with supporting other systems or activities. Computing systems for infrastructures are thus likely to be distributed and concerned in particular with supporting communication of information, control, people and materials. Issues of (for example) openness, timeliness, security, lack of corruption, and resilience are often important. (Winston Churchill is quoted to have said, during a debate in the House of Commons, in 1946: *...The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms as 'infra-structures'.*)

377. **Inheritance:** The act of inheriting' a 'property. (The term inheritance, in software engineering, is deployed in connection with a relationship between two pieces (i.e., *module*[464]s) of specification and/or program texts A and B. B may be said to *inherit* some *type*[782], or *variable*[803], or *value*[802] definitions from A.)

378. **In intension:** A concept of logic: In intension is a correlative word that indicates the internal content of a term or concept that constitutes its formal definition. (When we speak of functions in intension, we shall therefore mean it in the sense of presenting only the "input/output" relation of the function. Contrast to *in extension*[366].)

379. **Injection:** A mathematical function, f, that is a one-to-one mapping from *definition set*[211] A to *range set*[577] B. (That is, if for some a in A, $f(a)$ yields a b, then for all $a : A$ all $b : B$ are yielded and there is a unique a for each b, or, which is the same, there is an *inverse function*[401], f^{-1}, such that $f^{-1}(f(a)) = a$ for all $a : A$. See also *bijection*[86] and *surjection*[726].)

380. **Injective function:** A *function*[310] which maps *value*[802]s of its postulated *definition set*[211] into some, but not all, of its postulated *range set*[577] is called injective. (See also *bijective function*[87] and *surjective function*[727] .)

381. **In-order:** A special order of *tree traversal*[778] in which visits are made to nodes of trees and subtrees as follows: First the tree root is visited and "marked" as having been in-order visited. Then for each subtree a subtree in-order traversal is made, in the order left to right (or right to left). When a tree, whose number of subtrees is zero, is in-order traversed, then just that tree's root is visited (and that tree has then been in-order traversed) and (the leaf) is "marked" as having been visited. After each subtree visit the root (of the tree of which the subtree is a subtree) is revisited, i.e., again "marked" as having been in-order visited. (Cf. Fig. 13: a left to right in-order traversal of that tree yields the following sequence of "markings": AQCQALXLFLAKUKJKZMZKA. Cf. also Fig. 10).

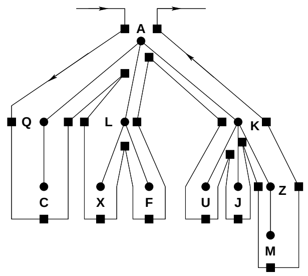

Figure 10: A left-to-right in-order tree traversal

382. **Input:** By input we mean the *communication*[122] of *information*[373] (*data*[193]) from an outside, an *environment*[275], to a *phenomenon*[524] "within" our universe of discourse. (More colloquially, and more generally: Input can be thought of as *value*[802](s) transferred over *channel*[110](s) to, or between *process*[544]es. Cf. *output*[502]. In a narrow sense we talk of input to an *automaton*[71] (i.e., a *finite state automaton*[289] or a *pushdown automaton*[564]) and a *machine*[436] (here in the sense of, for example, a *finite state machine*[290] (or a *pushdown machine*[565])).)

383. **Input alphabet:** The set of *symbol*[728]s *input*[382] to an *automaton*[71] or a *machine*[436] in the sense of, for example, a *finite state machine*[290] or a *pushdown machine*[565].

384. **Instance:** An individual, a thing, an *entity*[272]. (We shall usually think of an 'instance' as a *value*[802].)

385. **Instantiation:** 'To represent (an abstraction) by a concrete *instance*[384]' [213]. (We shall sometimes be using the term 'instantiation' in lieu of a *function invocation*[317] on an *activation stack*[13].)

386. **Installation manual:** A *document*[237] which describes how a *computing system*[151] is to be installed. (A special case of 'installation' is the downloading of *software*[685] onto a *computing system*[151]. See also *training manual*[767] and *user manual*[798].)

387. **Intangible:** Not *tangible*[742].

388. **Integrity:** By a *machine*[436] having integrity we mean that that machine remains unimpaired, i.e., has no faults, errors and failures, and remains so even in the situations where the environment of the machine has faults, errors and failures. (Integrity is a *dependability requirement*[218].)

389. **Intension:** Intension indicates the internal content of a term. (See also *in intension*[378]. The intension of a *concept*[152] is the collection of the properties possessed jointly by all conceivable individuals falling under the concept [178]. The intension determines the *extension*[283] [178].)

390. **Intensional:** Adjective form of *intension*[389].

391. **Interact:** The term interact here addresses the phenomenon of one *behaviour*[79] acting in unison, simultaneously, *concurrent*[159]ly, with another behaviour, including one behaviour influencing another behaviour. (See also *interaction*[392].)

392. **Interaction:** Two-way reciprocal action.

393. **Interface:** Boundary between two disjoint sets of communicating phenomena or concepts. (We shall think of the systems as *behaviour*[79]s or *process*[544]es, the boundary as being *channel*[110]s, and the communications as *input*[382]s and *output*[502]s.)

394. **Interface requirements:** Interface requirements are those *requirements*[605] which can on be expressed using professional, i.e., technical terms from *both* the *domain*[239] and the *machine*[436]. Thus, by interface requirements we understand the expression of expectations as to which software-software, or software-hardware *interface*[393] places (i.e., *channel*[110]s), *input*[382]s and *output*[502]s (including the *semiotics*[658] of these input/outputs) there shall be in some contemplated *computing system*[151]. (Interface requirements can often, usefully, be classified in terms of *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *computational data+control*

requirements[146], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448] and *machine-machine dialogue requirements*[437]. Interface requirements constitute one requirements *facet*[285]. Other requirements facets are: *business process reengineering*[101], *domain requirements*[258] and *machine requirements*[438].)

395. **Interface requirements facet:** See *interface requirements*[394] for a list of facets: *shared data initialisation*[670], *shared data refreshment*[672], *computational data+control*[146], *man-machine dialogue*[446], *man-machine physiological*[448] and *machine-machine dialogue*[437] *requirements*[605].

396. **Interpret:** See next: *interpretation*[397].

397. **Interpretation:** The three terms *elaboration*, *evaluation*[280] and *interpretation*[397] essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*[160], or as a function from configurations to *value*[802]s. Given that configuration typically consists of *static*[708] *environment*[275]s and *dynamic*[260] *state*[705]s (or *storage*[715]s), we use the term interpretation in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to states.

398. **Interpreter:** An interpreter is an *agent*[24], a *machine*[436], which performs *interpretation*[397]s.

399. **Intrinsics:** By the intrinsics of a *domain*[239] we shall understand those phenomena and concepts of a domain which are basic to any of the other facets, with such a domain intrinsics initially covering at least one specific, hence named, *stakeholder*[703] view. (Intrinsics is thus one of several *domain facet*[250]s. Others include: *business process*[99]es, *support technology*[725], *rules and regulations*[640], *scripts*[651], *management and organisation*[445], and *human behaviour*[345].)

400. **Invariant:** By an invariant we mean a property that holds of a *phenomenon*[524] or a *concept*[152], both before and after any *action*[12] involving that phenomenon or a concept. (A case in point is usually an *information*[373] or a *data structure*[199]: Assume an action, say a repeated one (e.g., a while loop). We say that the action (i.e., the while loop) preserves an invariant, i.e., usually a *proposition*[560], if the proposition holds true of the *state*[705] before and the state after any *interpretation*[397] of the while loop. Invariance is here seen separate from the *well-formedness*[812] of an *information*[373] or a *data structure*[199]. We refer to the explication of *well-formedness*[812]!)

401. **Inverse function:** See *injection*[379].

402. **Invocation:** See *function invocation*[317].

403. **Isomorphic:** One to one. (See *isomorphism*[404].)

404. **Isomorphism:** If a *homomorphism*[343] ϕ is a *bijective function*[87] then ϕ is an isomorphism. (See also *automorphism*[72], *endomorphism*[268], *epimorphism*[276] and *monomorphism*[467].)

. . . . $\mathcal{J}$

405. **J:** The **J** operator (**J** for **J**ump) was introduced (before 1965) by Peter Landin as a *functional*[312] used to explain the creation and use of program *closure*[117]s, and these again are used to model the *denotation*[213] of *label*[410]s. (We refer to [150, 152, 151, 149, 74]. Cf. `www.dcs.qmw.ac.uk/~peterl/danvy/`.)

. . . . $\mathcal{K}$

406. **Keyword:** A significant word from a title or document.

407. **Knowledge:** What is, or what can be known. The body of truth, information, and principles acquired by mankind [213]. (See *epistemology*[277] and *ontology*[492]. *A priori knowledge:* Knowledge that is independent of all particular experiences. *A posteriori knowledge:* Knowledge, which derives from experience alone.)

408. **Knowledge engineering:** The representation and modelling of knowledge. (The construction of ontological and epistemological knowledge and its manipulation. Involves such subdisciplines as *modal logic*[459]s (promise and commitment, knowledge and belief), *speech act* theories, *agent*[24] theories, etc. Knowledge engineering usually is concerned with the knowledge that one agent may have about another agent.)

409. **KWIC:** Abbreviation for keyword-in-context (A classical software application.)

. . . . $\mathcal{L}$

410. **Label:** Same as named *program point*[548].

411. **Lambda-application:** Within the confines of the *Lambda-calculus*[412], *Lambda-application*[411] is the same as *function application*[315]. (Subject, however, to simple *term-rewriting*[753] using (say just) *Alpha-renaming*[35] and *Beta-reduction*[84].)

412. **Lambda-calculus:** A *calculus*[104] for expressing and "manipulating" functions. The Lambda-calculus (λ-calculus) is a de facto "standard" for "what is computable". See *Lambda-expression*[414]s. As a *calculus*[104] it prescribes a language, the language of *Lambda-expression*[414]s, a set of *conversion*[184] rules — these apply to *Lambda-expression*[414]s and result in *Lambda-expression*[414]s. They "mimic" *function definition*[316] and *function application*[315]. The seminal texts on the Lambda-calculi are [66, 13, 14, 15].

413. **Lambda-combination:** See *Lambda-application*[411].

414. **Lambda-expression:** The language of the "pure" (i.e., simple, but fully powerful) *Lambda-calculus*[412] has three kinds of Lambda-expressions: *Lambda-variable*[416]s, *Lambda-function*[415]s and *Lambda-application*[411]s.

415. **Lambda-function:** By a Lambda-function we understand a *Lambda-expression*[414] of the form $\lambda x \bullet e$, where x is a binding variable and e is a Lambda-expression. (It is usually the case that e contains *free*[305] occurrences of x — these being bound by the binding variable in $\lambda x \bullet e$.)

416. **Lambda-variable:** The x in the *Lambda-function*[415] expression $\lambda x \bullet e$: both the formal parameter, the first x you see in $\lambda x \bullet e$, and all the *free*[305] occurrences of x in the *block*[89] (i.e., body) expression e.

417. **Language:** By a language we shall understand a possibly infinite set of *sentence*[660]s which follow some *syntax*[733], express some *semantics*[655] and are uttered, or written down, due to some *pragmatics*[534].

418. **Law:** A law is a rule of conduct prescribed as binding or enforced by a controlling authority. (We shall take the term law in the specific sense of law of Nature (cf., Ampére's Law, Boyle's Law, the conservation laws (of mass-energy, electric charge, linear and angular momentum), Newton's Laws, Ohm's Law, etc.), and laws of Mathematics (cf. "law of the excluded middle" (as in logic: a proposition must either be true, or false, not both, and not none)).)

419. **Leaf:** A leaf is a *node*[479] in a *tree*[777] for which there are no sub*tree*[777]s of that node. (Thus a leaf is a concept of *tree*[777]s. Cf. Fig. 13 on page 228.)

420. **Lemma:** An auxiliary *proposition*[560] used in the demonstration of another proposition. (Instead of proposition we could use the term *theorem*[756].)

421. **Lexical analysis:** The analysis of a *sentence*[660] into its constituent *word*[814]s. (Sentences also are usually "decorated" with such signs as for example punctuation marks (, . : ;), delimiters (( ) [ ], etc.), and other symbols (? !, etc.). Lexical analysis therefore is a process which serves to recognise which character sequences are words and which are not (i.e., which are delimiters, etc.).)

422. **Lexicographic:** The principles and practices of establishing, maintaining and using a dictionary. (We shall, in software engineering, mostly be using the term 'lexicographic' in connection with compilers and, more rarely, database schemas — although, as the definition implies, it is of relevance in any context where a computing system builds, maintains and uses a dictionary.)

423. **Lexicographical order:** The order, i.e., sequence, in which entries of a dictionary appear. (More specifically, the lexicographical ordering of entries in a *compiler dictionary*[126] is, for a *block-structured programming language*[90], determined by the nesting structure of *block*[89]s. The dictionary itself, generally "mimics" the nesting structure of the language.)

424. **License:** A license is a *script*[651] specifically expressing a permission to act; is freedom of action; is a permission granted by competent authority to engage in a business or occupation or in an activity otherwise unlawful; a document, plate, or tag evidencing a license granted; a grant by the holder of a copyright or patent to another of any of the rights embodied in the copyright or patent short of an assignment of all rights. Licenses appear more to have morally than legally binding poser.

425. **Link:** A link is the same as a *pointer*[528], an *address*[22] or a *reference*[587]: something which refers to, i.e., designates something (typically something else).

426. **Lifted function:** A lifted function, say of type $A \to B \to C$, has been created from a function of type $B \to C$ by 'lifting' it, i.e., by abstracting it in a variable, say a of type A. (Assume $\lambda b : B \cdot \mathcal{E}(b)$ to be a function of type $B \to C$. Now $\lambda a : A \cdot \lambda b : B \cdot \mathcal{E}(b)$ is a lifted version of $\lambda b : B \cdot \mathcal{E}(b)$. An example is **and**: $\lambda b_1, b_2 : \mathbf{Bool} \cdot b_1 \wedge b_2$, Boolean conjunction. We lift **and** to be a function, $\wedge_T$, over time: $\lambda t : T \cdot b_1(t) \wedge b_2(t)$, where the variables b_1, b_2 typically could be (e.g., assignable) variables whose values change over time.)

427. **Linguistics:** The study and knowledge of the *syntax*[733], *semantics*[655] and *pragmatics*[534] of *language*[417](s).

428. **List:** A list is an ordered sequence of zero, one or more not necessarily distinct entities.

429. **Literal:** A term whose use in software engineering, i.e., programming, shall mean: an identifier which denotes a constant, or is a keyword. (Usually that identifier is emphasised. Examples of RSL literals are: **Bool, true, false, chaos, if, then, else, end, let, in,** and the numerals 0, 1, 2., ..., 1234.5678, etc.)

430. **Live Sequence Chart:** The Live Sequence Chart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [121].)

431. **Location:** By a location is meant an area of *storage*[715].

432. **Logic:** The principles and criteria of validity of inference and deduction, that is, the mathematics of the formal principles of reasoning. (We refer to Vol. 1, Chap. 9 for our survey treatment of mathematical logic.)

433. **Logic programming:** Logic programming is programming based on an interpreter which either performs deductions or inductions, or both. (In logic programming the chief values are those of the Booleans, and the chief forms of expressions are those of propositions and predicates.)

434. **Logic programming language:** By a *logic programming*[433] language is meant a language which allows one to express, to prescribe, *logic programming*[433]. (The classical logic programming language is `Prolog` [160, 132].)

435. **Loose specification:** By a loose specification is understood a specification which either *underspecifies* a problem, or specifies this problem *nondeterministically*.

. . . . $\mathcal{M}$

436. **Machine:** By the machine we understand the *hardware*[331] plus *software*[685] that implements some *requirements*[605], i.e., a *computing system*[151]. (This definition follows that of M.A. Jackson [138].)

437. **Machine-Machine dialogue requirements:**By machine-machine dialogue requirements we understand the *syntax*[733] (incl. sequential structure), and *semantics*[655] (i.e., meaning) of the communications (i.e., messages) transferred in either direction over the automated interface between *machine*[436]s (including supporting technologies). (See also *computational data+control requirements*[146], *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447], and *man-machine physiological requirements*[448].)

438. **Machine requirements:** Machine requirements are those *requirements*[605] which, in principle, can be expressed without using professional (i.e., technical) terms from the *domain*[239] (for which these requirements are established). Thus, by *machine*[436] *requirements*[605] we understand *requirements*[605] put specifically to, i.e., expected specifically from, the *machine*[436]. (We normally analyse machine requirements into *performance requirements*[521], *dependability requirements*[218], *maintenance requirements*[443], *platform requirements*[527] and *documentation requirements*[238].)

439. **Machine service:** The service delivered by a machine is its *behaviour*[79] as it is perceptible by its user(s), where a user is a human, another machine, or a(nother) system which *interact*[391]s with it [196].

440. **Macro:** Macros have the same syntax as procedures, that is, a pair of a *signature*[680] (i.e., a macro name followed by a formal argument list of distinct identifiers (i.e., the *formal parameter*[302]s)) and a macro body, a text. Syntactically we can distinguish between macro definitions and macro *invocation*[402]s. Semantically, invocations, in some text, of the macro name and an *actual argument*[16] list are then to be thought of as an expansion of that part of the text with the macro (definition) body and such that formal parameters are replaced (*macro substitution*[441]) by actual arguments. Semantically a macro is different from a *procedure*[543] in that a macro expansion takes place in a *context*[172], i.e., an *environment*[275], where *free*[305] identifiers of the macro body are replaced by their value as defined at the place of the occurrence of the macro invocation. Whereas, for a procedure, the free identifiers of a procedure body are bound to their value at the point where the procedure was defined. (Thus the

difference between a macro and a procedure is the difference between *evaluation*[280] in a calling, versus in a defining environment.)

441. **Macro substitution:** See under *macro*[440]s.

442. **Maintenance:** By maintenance we shall here, for software, mean change to *software*[685], i.e., its various *document*[237]s, due to needs for (i) adapting that software to new *platform*[526]s, (ii) correcting that software due to observed software errors, (iii) improving certain performance properties of the *machine*[436] of which the software is part, or (iv) avoiding potential problems with that machine. (We refer to subcategories of maintenance: *adaptive maintenance*[21], *corrective maintenance*[187], *perfective maintenance*[519] and *preventive maintenance*[541].)

443. **Maintenance requirements:** By *maintenance*[442] *requirements*[605] we understand requirements which express expectations on how the *machine*[436] being desired (i.e., required) is expected to be maintained. (We also refer to *adaptive maintenance*[21], *corrective maintenance*[187], *perfective maintenance*[519] and *preventive maintenance*[541].)

444. **Management:** Management is about resources: their acquisition, scheduling (over time), allocation (over locations), deployment (in performing actions) and disposal ("retirement"). (We distinguish between board-directed, strategic, tactical and operational actions: *board-directed* actions target mainly financial resources: obtaining new funds through conversion of goodwill into financial resources, acquiring and selling "competing" or "supplementary" business units; *strategic* actions convert financial resources into production, service supplies and resources and vice-versa — and in this these actions schedule availability of such resources; *tactical* actions mainly allocate resources; and *operational* actions order, monitor and control the deployment of resources in the performance of actions.)

445. **Management and organisation:** The composite term management and organisation applies in connection with *management*[444] as outlined just above and with *organisation*[500]. The term then emphasises the relations between the organisation and management of an enterprise. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *rules and regulations*[640] and *human behaviour*[345].)

446. **Man-machine dialogue:** By man-machinedialogues we understand actual instantiations of *user*[796] interactions with *machine*[436]s, and machine interactions with users: what input the users provide, what output the machine initiates, the interdependencies of these inputs/outputs, their temporal and spatial constraints, including response times, input/output media (locations), etc. (

447. **Man-machine dialogue requirements:** By man-machine dialogue requirements we understand those *interface requirements*[394] which express expectations on, i.e.,

mandates the *protocol*[561] according to which *user*[796]s are to interact with the *machine*[436], and the machine with the users. (See *man-machine dialogue*[446]. For other *interface requirements*[394] see *computational data+control requirements*[146], *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine physiological requirements*[448] and *machine-machine dialogue requirements*[437].))

448. **Man-machine physiological requirements:** By man-machine physiological requirements we understand those *interface requirements*[394] which express expectations on, i.e., mandates, the form and appearance of ways in which the *man-machine dialogue*[446] utilises such physiological devices as visual display screens, keyboards, "mouses" (and other tactile instruments), audio microphones and loudspeakers, television cameras, etc. (See also *computational data+control requirements*[146], *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447] and *machine-machine dialogue requirements*[437].)

449. **Map:** A map is like a *function*[310], but is here thought of as an *enumerable*[273] set of pairs of argument/result values. (Thus the *definition set*[211] of a map is usually decidable, i.e., whether an entity is a member of a definition set of a map or not can usually be decided.)

450. **Mechanical semantics:** By a mechanical semantics we understand the same as an *operational semantics*[496] (which is again basically the same as a *computational semantics*[147]), i.e., a semantics of a language specified using concrete constructs (like stacks, program pointers, etc.), and otherwise as defined in *operational semantics*[496] and *computational semantics*[147].

451. **Mereology:** The theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole. (Mereology is often considered a branch of *ontology*[492]. Leading investigators of mereology were Franz Brentano, Edmund Husserl, Stanislaw Lesniewski [208, 163, 174, 211, 212, 216] and Leonard and Goodman [155].)

452. **Meta-IV:** `Meta-IV` stands for the fourth metalanguage (for programing language definition conceived at the IBM Vienna Laboratory in the 1960s and 1970s). (`Meta-IV` is pronounced meta-four.)

453. **Metalanguage:** By a metalanguage is understood a *language*[417] which is used to explain another language, either its *syntax*[733], or its *semantics*[655], or its *pragmatics*[534], or two or all of these! (One cannot explain any language using itself. That would lead to any interpretation of what is explained being a valid solution, in other words: Nonsense. `RSL` thus cannot be used to explain `RSL`. Typically formal specification languages are metalanguages: being used to explain, for example, the semantics of ordinary programming languages.)

454. **Metalinguistic:** We say that a language is used in a metalinguistic manner when it is being deployed to explain some other language. (And we also say that when we examine a language, like we could, for example, examine `RSL`, and when we use a subset of `RSL` to make that analysis, then that subset of `RSL` is used metalinguistically (wrt. all of `RSL`).)

455. **Metaphysics:** We quote from: http://mally.stanford.edu/: "Whereas physics is the attempt to discover the laws that govern fundamental concrete objects, metaphysics is the attempt to discover the laws that systematize the fundamental abstract objects presupposed by physical science, such as natural numbers, real numbers, functions, sets and properties, physically possible objects and events, to name just a few. The goal of metaphysics, therefore, is to develop a formal ontology, i.e., a formally precise systematization of these abstract objects. Such a theory will be compatible with the world view of natural science if the abstract objects postulated by the theory are conceived as patterns of the natural world." (Metaphysics may, to other scientists and philosophers, mean more or other, but for software engineering the characterisation just given suffices.)

456. **Method:** By a method we shall here understand a set of *principle*[542]s for selecting and using a number of *technique*[745]s and *tool*[763]s in order to construct some *artefact*[55]. (This is our leading definition — one that sets out our methodological quest: to identify, enumerate and explain the principles, the techniques and, in cases, the tools — notably where the latter are specification and programming languages. (Yes, languages are tools.))

457. **Methodology:** By methodology we understand the study and knowledge of *method*[456]s, one, but usually two or more. (In some dialects of English, methodology is confused with method.)

458. **Mixed computation:** By a mixed computation we understand the same as by a *partial evaluation*[516]. (The term mixed computation was used notably by Andrei Petrovich Ershov [99, 104, 105, 98, 100, 101, 102, 103], in my mind the "father" of Russian computing science.)

459. **Modal logic:** A modal is an expression (like "necessarily" or "possibly") that is used to qualify the truth of a judgment. Modal logic is, strictly speaking, the study of the deductive behavior of the expressions "it is necessary that" and "it is possible that". (The term "modal logic" may be used more broadly for a family of related systems. These include logics for belief, for tense and other temporal expressions, for the deontic (moral) expressions such as "it is obligatory that", "it is permitted that" and many others. An understanding of modal logic is particularly valuable in the formal analysis of philosophical argument, where expressions from the modal family are both common and confusing. Modal logic also has important applications in computer science [234].)

460. **Model:** A model is the mathematical meaning of a description (of a domain), or a prescription (of requirements), or a specification (of software), i.e., is the meaning of a specification of some universe of discourse. (The meaning can be understood either as a mathematical function, as for a *denotational semantics*[215] meaning, or an *algebra*[26] as for an *algebraic semantics*[27] or a *denotational semantics*[215] meaning, etc. The essence is that the model is some mathematical structure.)

461. **Model-oriented:** A specification (description, prescription) is said to be model-oriented if the specification (etc.) *denote*[216]s a *model*[460]. (Contrast to *property-oriented*[559].)

462. **Model-oriented type:** A type is said to be model-oriented if its specification *designate*[222]s a *model*[460]. (Contrast to *property-oriented*[559] *type*[782].)

463. **Modularisation:** The act of structuring a text using *module*[464]s.

464. **Module:** By a module we shall understand a clearly delineated text which denotes either a single complex quantity, as does, usually, an *object*[487], or a possibly empty, possibly infinite set of *model*[460]s of objects. (The RSL module concept is manifested in the use of one or more of the RSL *class*[114] (**class** ... **end**), *object*[487] (**object** identifier **class** ... **end**, etc.), and *scheme*[648] (**scheme** identifier **class** ... **end**), etc., constructs. We refer to [73, 72, 23] and to [186, 185] for original, early papers on modules.)

465. **Module design:** By module design we shall understand the *design*[221] of (one or more) *module*[464]s.

466. **Monitor:** Syntactically a monitor is "a programming language construct which encapsulates variables, access procedures and initialisation code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections." Semantically "a monitor may have a queue of processes which are waiting to access it" [108].

467. **Monomorphism:** If a *homomorphism*[343] ϕ is an *injective function*[380] then ϕ is an isomorphism. (See also *automorphism*[72], *endomorphism*[268], *epimorphism*[276], and *monomorphism*[467].)

468. **Monotonic:** A function, $f : A \to B$, is monotonic, if for all a, a' in the definition set A of f, and some ordering relations, $\sqsubseteq$, on a and B, we have that if $a \sqsubseteq a'$ then $f(a) \sqsubseteq f(a')$.

469. **Mood:** A conscious state of mind, as here, of a specification. (We can thus express an *indicative*[362] mood, an *optative*[499] mood, a *putative*[566] mood or an *imperative*[352] mood. Our use of these various forms of moods is due to Michael Jackson [138].)

470. **Morphism:** Same as *homomorphism*[343].

471. **Morphology:** (i) A study and description of word formation (as inflection, derivation, and compounding) in language; (ii) the system of word-forming elements and processes in a language; (iii) a study of structure or form [213].

472. **Multi-dimensional:** A composite (i.e., a non*atomic*[63]) *entity*[272] is a multi-dimensional *entity*[272] if some relations between properly contained (i.e., constituent) subentities (cf. *subentity*[721]) can only be described by both forward and backward references, and/or with recursive references. (This is in contrast to *one-dimensional*[491] entities.)

473. **Multimedia:** The use of various forms of input/output media in the man-machine interface: Text, two-dimensional graphics, voice (audio), video, and tactile instruments (like "mouse").

. . . . $\mathcal{N}$

474. **Name:** A name is syntactically (generally an expression, but usually it is) a simple alphanumeric identifier. Semantically a name denotes (i.e., designates) "something". Pragmatically a name is used to uniquely identify that "something". (Shakespeare: Romeo: "What's in a name?" Juliet to Romeo: "That which we call a rose by any other name would smell as sweet.")

475. **Naming:** The action of allocating a unique name to a value.

476. **Narrative:** By a narrative we shall understand a document text which, in precise, unambiguous language, introduces and describes (prescribes, specifies) all relevant properties of entities, functions, events and behaviours, of a set of phenomena and concepts, in such a way that two or more readers will basically obtain the same idea as to what is being described (prescribed, specified). (More commonly: Something that is narrated, a story.)

477. **Natural language:** By a natural language we shall understand a language like Arabic, Chinese, English, French, Russian, Spanish, etc. — one that is spoken today, 2005, by people, has a body of literature, etc. (In contrast to natural languages we have (i) professional languages, like the languages of medical doctors, or lawyers, or skilled craftsmen like carpenters, etc.; and we have (ii) formal languages like software specification languages, programming languages, and the languages of first-order predicate logics, etc.)

478. **Network:** By a network we shall understand the same as a directed, but not necessarily *acyclic graph*[19]. (Our only use of it here is in connection with network *databases*.)

479. **Node:** A point in some *graph*[327] or *tree*[777].

480. **Nondeterminate:** Same as *nondeterministic*[481].

481. **Nondeterministic:** A property of a specification: May, on purpose, i.e., deliberately have more than one meaning. (A specification which is ambiguous also has more than one meaning, but its ambiguity is of overriding concern: It is not 'nondeterministic' (and certainly not 'deterministic'!).)

482. **Nondeterminism:** A *nondeterministic*[481] specification models nondeterminism.

483. **Nonstrict:** Nonstrictness is a property associated with functions. A function is nonstrict, in certain or all arguments, if, for undefined values of these it may still yield a defined value. (See also *strict function*[717]s.)

484. **Nonterminal:** The concept of a nonterminal (together with the concept of a *terminal*[750]) is a concept associated with the *rule of grammar*[639]s. (See that term: *rule of grammar*[639] for a full explanation.)

485. **Notation:** By a notation we shall usually understand a reasonably precisely delineated language. (Some notations are textual, as are programming notations or specification languages; some are diagrammatic, as are, for example, *Petri net*[522]s, *Statechart*[706]s, *Live Sequence Chart*[430]s, etc.)

486. **Noun:** Something, a name, that refers to an *entity*[272], a quality, a *state*[705], an *action*[12], or a *concept*[152]. Something that may serve as the subject of a *verb*[806]. (But beware: In English many nouns can be "verbed", and many verbs can be "nouned"!)

. . . . $\mathcal{O}$

487. **Object:** An instance of the *data structure*[199] and *behaviour*[79] defined by the object's *class*[114]. Each object has its own *value*[802]s for the instance *variable*[803]s of its class and can respond to the *function*[310]s defined by its class. (Various *specification language*[699]s, `object Z` [61, 95, 96], `RSL`, etc., each have their own, further refined, meaning for the term 'object', and so do *object-oriented*[488] *programming language*[551] (viz., `C++` [215], `Java` [10, 113, 157, 224, 6, 207], `C#` [191, 173, 172, 125] and so on).)

488. **Object-oriented:** We say that a program is *object-oriented*[488] if its main structure is determined by a *modularisation*[463] into a *class*[114], that is, a cluster of *type*[782]s, *variable*[803]s and *procedure*[543]s, each such set acting as a separate *abstract data type*[4]. Similarly we say that a *programming language*[551] is object-oriented if it specifically offers language constructs to express the appropriate *modularisation*[463]. (Object-orientedness became a mantra of the 1990s: Everything had to be object-oriented. And many programming problems are indeed well served by being structured around some object-oriented notion. The first *object-oriented*[488] *programming language*[551] was `Simula 67` [23].)

489. **Observer:** By an observer we mean basically the same as an *observer function*[490].

490. **Observer function:** An observer function is a *function*[310] which when "applied" to an *entity*[272] (a *phenomenon*[524] or a *concept*[152]) yields subentities or attributes of that entity (without "destroying" that entity). (Thus we do not make a distinction between functions that observe subentities (cf. *subentity*[721]) and functions that observe *attribute*[69]s. You may wish to make distinctions between the two kinds of observer function. You can do so by some simple *naming*[475] convention: assign names the prefix `obs_` when you mean to observe subentities, and `attr_` when you mean to observe attributes. Vol. 3 Chap. 5 introduces these concepts.)

491. **One-dimensional:** A composite *entity*[272] is a one-dimensional *entity*[272] if all relations between properly contained (i.e., constituent) subentities can be described by either no references to other subentities, or only by backward or only by forward references. (This is in contrast to *multi-dimensional*[472] entities. Thus arrays of arbitrary order (vectors, matrices, tensors) are usually one-dimensional.)

492. **Ontology:** In philosophy: A systematic account of Existence. To us: An explicit formal specification of how to represent the phenomena, concepts and other entities that are assumed to exist in some area of interest (some universe of discourse) and the relationships that hold among them. (Further clarification: An ontology is a catalogue of *concept*[152]s and their relationships — including properties as relationships to other concepts.)

493. **Operation:** By an operation we shall mean a *function*[310], or an *action*[12] (i.e., the effect of function *invocation*[402]). (The context determines which of these two strongly related meanings are being referred to.)

494. **Operational:** We say that a *specification*[698] (a *description*[220], a *prescription*[540]), say of a *function*[310], is operational if what it explains is explained in terms of how that thing, **how** that phenomenon, or concept, operates (rather than by **what** it achieves). (Usually operational definitions are *model oriented*[461] (in contrast to *property oriented*[559]).)

495. **Operational abstraction:** Although a definition (a *specification*[698], a *description*[220], or a *prescription*[540]) may be said, or claimed, to be *operational*[494], it may still provide *abstraction*[3] in that the *model-oriented*[461] concepts of the definition are not themselves directly representable or performable by humans or computers. (This is in contrast to *denotational*[214] *abstraction*[3]s or *algebra*[26]ic (or *axiom*[75]atic) *abstraction*[3]s.)

496. **Operational semantics:** A *definition*[210] of a *language*[417] *semantics*[655] that is *operational*[494]. (See also *structural operational semantics*[720].)

497. **Operation reification:** To speak of *operation*[493] *reification*[597] one must first be able to refer to an abstract, usually *property-oriented*[559], specification of the operation.

Then, by operation *reification*[597] we mean a *specification*[698] which indicates how the operation might be (possibly efficiently) implemented. (Cf. *data reification*[198] and *operation transformation*[498].)

498. **Operation transformation:** To speak of *operation*[493] *reification*[597] one must first be able to refer to an abstract, usually *property-oriented*[559], specification of the operation. Then, by operation *transformation*[771] we mean a *specification*[698] which is, somehow, *calculate*[102]d from the abstract specification. (Three nice books on such calculi are: [176, 22, 11].)

499. **Optative:** Expressive of wish or desire. (See also *imperative*[352], *indicative*[362], and *putative*[566].)

500. **Organisation:** Organisation is about the "grand scale", executive and strategic national, continental or global (world wide) (i) allocation of major resource (e.g., business) units, whether in a hierarchical, in a matrix, or in some other organigram-specified structure, (ii) as well as the clearly defined relations (which information, decisions and actions are transferred) between these units, and (iii) organisational dynamics.

501. **Organisation and management:** The composite term organisation and management applies in connection with *organisation*[500] as outlined just above and with *management*[444]s (cf. Item 444 on page 190). The term then emphasises the relations between the organisation and management of an enterprise. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *rules and regulations*[640] and *human behaviour*[345].)

502. **Output:** By output we mean the *communication*[122] of *information*[373] (*data*[193]) to an outside, an *environment*[275], from a *phenomenon*[524] "within" our universe of discourse. (More colloquially, and more generally: output can be thought of as *value*[802](s) transferred over *channel*[110](s) from, or between, *process*[544]es. Cf. *input*[382]. In a narrow sense we talk of output from a *machine*[436] (e.g., a *finite state machine*[290] or a *pushdown machine*[565]).)

503. **Output alphabet:** The set of *symbol*[728]s *output*[502] from a *machine*[436] in the sense of, for example, a *finite state machine*[290] or a *pushdown machine*[565].

504. **Overloaded:** The concept of 'overloaded' is a concept related to *function*[310] *symbol*[728]s, i.e., *function*[310] *name*[474]s. A function name is said to be overloaded if there exists two or more distinct *signature*[680]s for that function name. (Typically overloaded function symbols are '+', which applies, possibly, in some notation, to addition of integers, addition of reals, etc., and '=', which applies, possibly, in some notation, to comparison of any pair of *value*[802]s of the same *type*[782].)

. . . . $\mathcal{P}$

505. **Paradigm:** A philosophical and theoretical framework of a scientific school or discipline within which theories, laws and generalizations and the experiments performed in support of them are formulated; a philosophical or theoretical framework of any kind. (Software engineering is full of paradigms: Object-orientedness is one.)

506. **Paradox:** A statement that is seemingly contradictory or opposed to common sense and yet is perhaps true. An apparently sound argument leading to a contradiction. (Some famous examples are Russell's Paradox[16] and the Liar Paradox.[17] Most paradoxes stem from some kind of self-reference.)

507. **Parallel programming language:** A *programming language*[551] whose major kinds of concepts are *process*[544]es, process *composition*[140] [putting processes in parallel and *nondeterministic*[481] {internal or external} choice of process *elaboration*[264]], and synchronisation and communication between processes. (A main example of a practical parallel programming language is `occam` [134], and of a specificational 'programming' language is `CSP` [130, 202, 206]. Most recent *imperative programming language*[354]s (Java, C#, etc.) provide for programming constructs (e.g., threads) that somehow mimic parallel programming.)

508. **Parameter:** Same as *formal parameter*[302].

509. **Parametric polymorphism:** See the parenthesised part of the *polymorphic*[529] entry.

510. **Parameterised:** We say that a *definition*[210], of a *class*[114] (or of a *function*[310]) is parameterised if an *instantiation*[385] of an *object*[487] of the class (respectively an *invocation*[402] of the function) allows an *actual argument*[16] to be substituted (cf. *substitution*[722]) into the class definition (function body) for every occurrence of the [formal] *parameter*[508].

511. **Parser:** A parser is an *algorithm*[31], say embodied as a *software*[685] *program*[545], which accepts text strings, and, if the text string is generated by a suitable *grammar*[325], then it will yield a *parse tree*[512] of that string. (See *generator*[322].)

512. **Parse tree:** To speak of a parse tree we assume the presence of a string of *terminal*[750]s and *nonterminal*[484]s, and of a *grammar*[325]. A parse tree is a *tree*[777] such that each subtree (of a *root*[632] and its immediate descendants, whether *terminal*[750]s or *nonterminal*[484]s) corresponds to a *rule*[638] of the grammar, and hence such that the *frontier*[307] of the tree is the given string.

513. **Parsing:** The act of attempting to construct a *parse tree*[512] from a *grammar*[325] and a text string.

[16] If R is the set of all sets which do not contain themselves, does R contain itself? If it does then it doesn't and vice versa.
[17] "This sentence is false" or "I am lying".

514. **Part:** To speak of parts we must be able to speak of "parts and wholes". That is: We assume some *mereology*[451], i.e., a theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole.

515. **Partial algebra:** A partial *algebra* is an algebra whose functions are not defined for all combinations of arguments over the carrier.

516. **Partial evaluation:** To speak of partial evaluation we must first speak of *evaluation*[280]. Normally evaluation is a *process*[544], as well as the result of that process, whereby an *expression*[282] in some language is evaluated in some *context*[172] which binds every *free identifier*[305] of the expression to some *value*[802]. A partial evaluation is an evaluation in whose context not all free identifiers are bound to (hence, defined) values. The result of a partial evaluation is therefore a symbolic evaluation, one in which the resulting value is expressed in terms of actual values and the undefined free identifiers. (We refer to [51, 141].)

517. **Path:** The concept of paths is usually associated with *graph*[327]s and *tree*[777]s (i.e., networks). A path is then a sequence of one or more graph edges or tree branches such that two consecutive edges (branches) share a node of the graph (or [root] of a tree). (We shall also use the term *route*[635] synonymously with paths.)

518. **Pattern:** We shall take a pattern, p, (as in RSL) to mean an expression with identifiers, a, and constants, k, as follows. Basis clauses: Any identifier a is a pattern, and any constant, k, is a pattern. Inductive clause: If $p_1, p_2, \ldots, p_m$ are patterns, then so are $(p_1, p_2, \ldots, p_m)$, $< p_1,\ p_2,\ \ldots,\ p_m >$, $\{p_1,\ p_2, \ldots,\ p_m\}$, $[p_{d_1} \mapsto p_{r_1},\ p_{d_2} \mapsto p_{r_2},\ \ldots,\ p_{d_m} \mapsto p_{r_m}]$, and so are: $< p > \hat{}\, a$, $a \hat{}\, < p >$, $\{p\} \cup a$, and $[p_{d_1} \mapsto p_{r_1}] \cup a$. (The idea is that a pattern, p, is "held up against" a value, v, "of the same kind" and then we attempt to "match" the pattern, p, with the value, v, and if a matching can be made, then the free identifiers of p are bound to respective component values of v.)

519. **Perfective maintenance:** By perfective maintenance we mean an update, as here, of software, to achieve a more desirable use of resources: time, storage space, equipment. (We also refer to *adaptive maintenance*[21], *corrective maintenance*[187] and *preventive maintenance*[541].)

520. **Performance:** By performance we, here, in the context of computing, mean quantitative figures for the use of computing resources: time, storage space, equipment.

521. **Performance requirements:** By performance requirements we mean *requirements*[605] which express *performance*[520] properties (desiderata).

522. **Petri net:** The Petri net language is a special graphic notation for expressing concurrency of actions, and simultaneity of events, of processes. (See [199].)

523. **Phase:** By a phase we shall here, in the context of software development, understand either the *domain*[239] *development*[228] phase, the *requirements*[605] *development*[228] phase, or the *software design*[688] phase.

524. **Phenomenon:** By a phenomenon we shall mean a physically manifest "thing". (Something that can be sensed by humans (seen, heard, touched, smelled or tasted), or can be measured by physical apparatus: Electricity (voltage, current, etc.), mechanics (length, time and hence velocity, acceleration, etc.), chemistry, etc.)

525. **Phenomenology:** Phenomenology is the study of structures of consciousness as experienced from the first-person point of view [234].

526. **Platform:** By a platform, we shall, in the context of computing, understand a *machine*[436]: Some computer (i.e., hardware) equipment and some software systems. (Typical examples of platforms are: `Microsoft Windows` running on an `IBM ThinkPad Series T` model, or `Trusted Solaris` operating system with an `Oracle Database` 10g running on a `Sun Fire E25K Server`.)

527. **Platform requirements:** By platform requirements we mean *requirements*[605] which express *platform*[526] properties (desiderata). (There can be several platform requirements: One set for the platform on which software shall be developed. Another set for the platform(s) on which software shall be utilised. A third set for the platform on which software shall be demonstrated. And a fourth set for the platform on which software shall be maintained. These platforms need not always be the same.)

528. **Pointer:** A pointer is the same as an *address*[22], a *link*[425], or a *reference*[587]: something which refers to, i.e., designates something (typically something else).

529. **Polymorphic:** Polymorphy is a concept associated with functions and the type of the values to which the function applies. If, as for the length of a list function, **len**, that function applies to lists of elements of any type, then we say the **len**gth function is polymorphic. So, in general, the ability to appear in many forms; the quality or state of being able to assume different forms. From Wikipedia, the Free Enclycopedia [227]:

> In computer science, polymorphism is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations. The concept of polymorphism applies to functions as well as types: A function that can evaluate to and be applied to values of different types is known as a polymorphic function. A data type that contains elements of an unspecified type is known as a polymorphic data type. There are two fundamentally different kinds of polymorphism: If the range of actual types that can be used is finite and the combinations must be specified individually prior to use, it is called *ad hoc polymorphism*[23]. If all code is written without mention of any specific type and thus can

be used transparently with any number of new types, it is called *parametric polymorphism*. Programming using the latter kind is called *generic programming*, particularly in the object-oriented community. However, in many statically typed functional programming languages the notion of parametric polymorphism is so deeply ingrained that most *programmers* simply take it for granted.

530. **Portability:** Portability is a concept associated with *software*[685], more specifically with the *program*[545]s (or *data*[193]). Software is (or files, including *database*[195] records, are) said to be portable if it (they), with ease, can be "ported" to, i.e., made to "run" on, a new *platform*[526] and/or compile with a different compiler, respectively different database management system.

531. **Post-condition:** The concept of post-condition is associated with function application. The post-condition of a function f is a predicate p_{o_f} which expresses the relation between argument a and result r values that the function f defines. If a represent argument values, r corresponding result values and f the function, then $f(a) = r$ can be expressed by the post-condition predicate p_{o_f}, namely, for all applicable a and r the predicate p_{o_f} expresses the truth of $p_{o_f}(a, r)$. (See also *pre-condition*[535].)

532. **Postfix:** The concept of postfix is basically a syntactic one, and is associated with operator/operand expressions. It is one about the displayed position of a unary (i.e., a monadic) operator with respect to its operand (expression). An expression is said to be in postfix form if a monadic operator is shown, is displayed, after the expression to which it applies. (Typically the factorial operator, say !, is shown after its operand expression, viz. 7!.)

533. **Post-order:** A special order of *tree traversal*[778] in which visits are made to nodes of trees and subtrees as follows: First, for each subtree, a subtree post-order traversal is made, in the order left to right (or right to left). When a tree, whose number of subtrees is zero, is post-order traversed, then just that tree's root is visited (and that tree has then been post-order traversed) and (the leaf) is "marked" as having been post-order visited. After each subtree visit the root of the tree of which the subtree is a subtree is revisited and now it is "marked" as having been visited. (Cf. Fig. 13 on page 228: A left to right post-order traversal of that tree yields the following sequence of "markings": CQXFLUJMZKA; cf. also Fig. 11).

534. **Pragmatics:** Pragmatics is the (i) study and (ii) practice of the factors that govern our choice of language in social interaction and the effects of our choice on others. (We use the term pragmatics in connection with the use of language, as complemented by the *semantics*[655] and *syntax*[733] of language.)

535. **Pre-condition:** The concept of pre-condition is associated with function application where the function being applied is a partial function. That is: for some arguments of its definition set the function yields **chaos**, that is, does not terminate. The

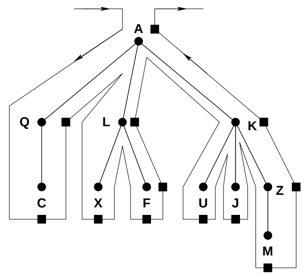

Figure 11: A left to right post-order tree traversal

pre-consition of the function is then a predicate which expresses those values of the arguments for which the function application terminates, that is, yields a result value. (See *weakest pre-condition*[811].)

536. **Predicate:** A predicate is a truth-valued expression involving terms over arbitrary values, well-formed formula relating terms and with *Boolean*[93] *connective*[167]s and *quantifier*[569]s.

537. **Predicate logic:** A predicate logic is a language of *predicate*[536]s (given by some *formal*[296] *syntax*[733]) and a *proof system*[557].

538. **Pre-order:** A special order of *tree traversal*[778] in which visits are made to nodes of trees and subtrees as follows: First to the root of the tree with that root now being "marked" as having been pre-order visited. Then for each subtree a subtree pre-order traversal is made, in the order left to right (or right to left). When a tree, whose number of subtrees is zero, is pre-order traversed, then just that tree's root is visited (and that tree has then been pre-order traversed) and the leaf is then "marked" as having been pre-order visited. (Cf. Fig. 13 on page 228: A right-to-left pre-order traversal of that tree yields the following sequence of "markings": AKZMJULFXQC. Cf. also Fig. 12 on the facing page).

539. **Presentation:** By presentation we mean the syntactic *document*[237]ation of the results of some *development*[228].

540. **Prescription:** A prescription is a specification which prescribes something designatable, i.e., which states what shall be achieved. (Usually the term 'prescription' is used only in connection with *requirements*[605] prescriptions.)

541. **Preventive maintenance:** By preventive maintenance — of a *machine*[436] — we mean that a set of special tests are performed on that *machine*[436] in order to ascertain whether the *machine*[436] needs *adaptive maintenance*[21], and/or *corrective main-*

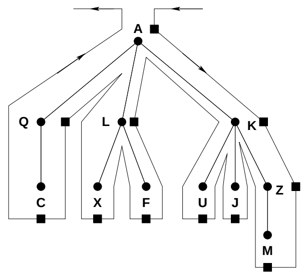

Figure 12: A right-to-left pre-order tree traversal

tenance[187], and/or *perfective maintenance*[519]. (If so, then an update, as here, of software, has to be made in order to achieve suitable *integrity*[388] or *robustness*[631] of the *machine*[436].)

542. **Principle:** An accepted or professed rule of action or conduct, ..., a fundamental doctrine, right rules of conduct, ... [214]. (The concept of principle, as we bring it forth, relates strongly to that of *method*[456]. The concept of principle is "fluid". Usually, by a method, some people understand an orderliness. Our definition puts the orderliness as part of overall principles. Also, one usually expects analysis and construction to be efficient and to result in efficient artifacts. Also this we relegate to be implied by some principles, techniques and tools.)

543. **Procedure:** By a procedure we mean the same as a *function*[310]. (Same as *routine*[636] or *subroutine*[723].)

544. **Process:** By a process we understand a sequence of actions and events. The events designate interaction with some environment of the process.

545. **Program:** A program, in some *programming language*[551], is a formal text which can be subject to *interpretation*[397] by a computer. (Sometimes we use the term *code*[118] instead of program, namely when the program is expressed in the machine language of a computer.)

546. **Programmable:** An *active*[14] *dynamic*[260] *phenomenon*[524] has the programmable (active dynamic) attribute if its *action*[12]s (hence *state*[705] changes) over a future time interval can be accurately prescribed. (Cf. *autonomous*[73] and *biddable*[85].)

547. **Programmer:** A person who does *software design*[688].

548. **Program point:** By a program point we shall here understand any point in a program text (whether of an *applicative programming language*[49] (i.e., *functional programming language*[314]), an *imperative programming language*[354], or a *logic programming language*[434]) between any two textually neighbouring *token*[762]s. (The idea of a program point is the following: Assume an *interpreter*[398] of programs of the designated kind. Such an interpreter, at any step of its *interpretation*[397] *process*[544], can be thought of as interpreting a special token, or a sequence of neighbouring tokens, in both cases: "between two program points".)

549. **Program organisation:** By program organisation we loosely mean how a *program*[545] (i.e., its text) is structured into, for example, *module*[464]s (eg., *class*[114]es), *procedure*[543]s, etc.

550. **Programming:** The act of constructing *program*[545]s. From [108]:

> *1: The art of debugging a blank sheet of paper (or, in these days of on-line editing, the art of debugging an empty file). 2: A pastime similar to banging one's head against a wall, but with fewer opportunities for reward. 3: The most fun you can have with your clothes on (although clothes are not mandatory).*

551. **Programming language:** A language for expressing *program*[545]s, i.e., a language with a precise *syntax*[733], a *semantics*[655] and some textbooks which provides remnants of the *pragmatics*[534] that was originally intended for that programming language. (See next entry: *programming language type*[552].)

552. **Programming language type:** With a *programming language*[551] one can associate a *type*[782]. Typically the name of that type intends to reveal the type of a main paradigm, or a main data type of the language. (Examples are: *functional programming language*[314] (major data type is functions, major operations are definition of functions, application of functions and composition of functions), *logic programming language*[434] (major kinds of expressions are ground terms in a Boolean algebra, propositions and predicates), *imperative programming language*[354] (major kinds of language constructs are declaration of assignable variables, and assignment to variables, and a more or less indispensable kind of data type is references [locations, addresses, pointers]), and *parallel programming language*[507].)

553. **Projection:** By projection we shall here, in a somewhat narrow sense, mean a technique that applies to *domain description*[243]s and yields *requirements prescription*[615]s. Basically projection "reduces" a domain description by "removing" (or, but rarely, *hiding*[337]) *entities*[272], *function*[310]s, *event*[281]s and *behaviour*[79]s from the domain description. (If the domain description is an informal one, say in English, it may have expressed that certain entities, functions, events and behaviours *might* be in (some instantiations of) the domain. If not "projected away" the similar, i.e., informal requirements prescription will express that these entities, functions, events and behaviours *shall* be in the domain and hence *will* be in the environment of the *machine*[436] being requirements prescribed.)

554. **Proof:** A *proof* of a theorem, ϕ, from a set, Γ, of sentences of some *formal*[296] *proposition*[560]al or *predicate*[536] language, $\mathcal{L}$, is a finite sequence of sentences, ϕ_1, ϕ_2, $\ldots$, ϕ_n, where $\phi = \phi_1$, where $\phi_n =$ **true**, and in which each ϕ_i is either an *axiom*[75] of $\mathcal{L}$, or a member of Γ, or follows from earlier ϕ_j's by an *inference rule*[369] of $\mathcal{L}$.

555. **Proof obligation:** A clause of a program may only be (dynamically) well-defined if the values of clause parts lie in certain ranges (viz. no division by zero). We say that such clauses raise proof obligations, i.e., an obligation to prove a property. (Classically it may not be statically (i.e., compile time) checkable that certain expression values lie within certain *subtype*[724]s. Discharging a proof may help ensure such constraints.)

556. **Proof rule:** Same as *inference rule*[369] or *axiom*[75].

557. **Proof system:** A *consistent*[168] and (relative) *complete*[129] set of *proof rule*[556]s.

558. **Property:** A quality belonging and especially peculiar to an individual or thing; an *attribute*[69] common to all members of a class. (Hence: "Not a property owned by someone, but a property possessed by something".)

559. **Property-oriented:** A specification (description, prescription) is said to be property-oriented if the specification (etc.) expresses *attribute*[69]s. (Contrast to *model oriented*[461].)

560. **Proposition:** An expression in language which has a truth value.

561. **Protocol:** A set of formal rules describing how to exchange messages, between a human user and a *machine*[436], or, more classically, across a network. (Low-level protocols define the electrical and physical standards to be observed, bit and byte ordering, and the transmission and error detection and correction of the bit stream. High-level protocols deal with the data formatting, including the syntax of messages, the terminal-to-computer dialogue, character sets, sequencing of messages, etc.)

562. **Pure functional programming language:** A *functional programming language*[314] is said to be pure if none of its constructs designates *side-effects*.

563. **Pushdown stack:** A pushdown stack is a simple *stack*[700]. (Usually a simple stack has just the following operations: *push* an element onto the stack, *pop* the top element from the stack, and observe the *top* element of the stack.)

564. **Pushdown automaton:** A pushdown automaton is an *automaton*[71] with the addition of a *pushdown stack*[563] such that (i) the pushdown automaton *input*[382] is provided both from an environment external to the pushdown automaton and from the *top* of the pushdown stack, (ii) the pushdown automaton *output*[502] is provided to the pushdown stack by being *push*ed onto the top of that stack, and (iii) such

that the pushdown automaton may direct an element to be *pop*ped from the pushdown stack. (The pushdown automaton still has the notion of the final states of the *automaton*[71].)

565. **Pushdown machine:** A pushdown (stack) machine is like a *pushdown automaton*[564] with the addition that now the pushdown machine also provides *output*[502] to the environment of the pushdown machine.

566. **Putative:** Commonly accepted or supposed, that is, assumed to exist or to have existed. (See also *imperative*[352], *indicative*[362] and *optative*[499].)

. . . . $\mathcal{Q}$

567. **Quality:** Specific and essential character. (Quality is an *attribute*[69], a *property*[558], a characteristic (something has character).)

568. **Quantification:** The operation of quantifying. (See *quantifier*[569]. The x (the y) is quantifying expression $\forall x{:}X{\cdot}P(x)$ (respectively $\exists y{:}Y{\cdot}Q(y)$).)

569. **Quantifier:** A marker that quantifies. It is a prefixed operator that binds the variables in a logical formula by specifying their possible range of *value*[802]s. (Colloquially we speak of the **universal** and the **existential** quantifiers, $\forall$, respectively $\exists$. Typically a quantified expression is then of either of the forms $\forall x{:}X{\cdot}P(x)$ and $\exists y{:}Y{\cdot}Q(y)$. They 'read': For all quantities x of type X it is the case that the predicate $P(x)$ holds; respectively: There exists a quantity y of type Y such that the predicate $Q(y)$ holds.)

570. **Quantity:** An indefinite *value*[802]. (See the *quantifier*[569] entry: The quantities in $P(x)$ (respectively $Q(y)$) are of type X (respectively Y). y is indefinite in that it is one of the quantities of Y, but which one is not said.)

571. **Query:** A request for information, generally as a formal request to a *database*[195].

572. **Query language:** A *formal*[296] *language*[417] for expressing queries (cf. *query*[571]). (The most well-known query language, today, 2005, is `SQL` [77].)

573. **Queue:** A queue is an *abstract data type*[4] with a queue data structure and, typically, the following operations: enqueue (insert into one end of the queue), dequeue (remove from the other end of the queue). Axioms then determine specific queue properties. ()

. . . . $\mathcal{R}$

574. **Radix:** In a positional representation of numbers, that integer by which the significance of one digit place must be multiplied to give the significance of the next higher digit place. (Conventional decimal numbers are radix ten, binary numbers are radix two.)

575. **RAISE:** `RAISE` stands for `R`igorous `A`pproach to `I`ndustrial `S`oftware `E`ngineering. (`RAISE` refers to a method, `The RAISE Method` [112, 31, 33, 34], a specification language, `RSL` [110], and "comes" with a set of tools.)

576. **Range:** The concept of range is here used in connection with functions. Same as *range set*[577]. See next entry.

577. **Range set:** Given a *function*[310], its range set is that set of *value*[802]s which is yielded when the function is *applied* to each member of its *definition set*[211].

578. **Reactive:** A *phenomenon*[524] is said to be reactive if the phenomenon performs *action*[12]s in response to external stimuli. Thus three properties must be satisfied for a system to be of reactive dynamic attribute: (i) An interface must be definable in terms of (ii) provision of input stimuli and (iii) observation of (state) reaction. (Contrast to *inert*[367] and *active*[14].)

579. **Reactive system:** A *system*[736] whose main phenomena are chiefly *reactive*[578]. (See the *reactive*[578] entry just above.)

580. **Real time:** We say that a *phenomenon*[524] is real time if its behaviour somehow must guarantee a response to an external event within a given time. (Cf. *hard real time*[330] and *soft real time*[684].)

581. **Reasoning:** Reasoning is the ability to *infer*[368], i.e., to make *deduction*[205]s or *induction*[364]s. (Automated reasoning is concerned with the building and use of computing systems that automate this process. The overall goal is to mechanise different forms of reasoning.)

582. **Recogniser:** A recogniser is an *algorithm*[31] which can decide whether a string can be *generate*[321]d by a given *grammar*[325] of a *language*[417]. (Typically a recogniser can be abstractly formulated as a *finite state automaton*[289] for a *regular language*[594], and as a *pushdown automaton*[564] for a *context-free language*[174].)

583. **Recognition rule:** A recognition rule is a text which describes some *phenomenon*[524], that is, a possibly singleton *class*[114] of such (i.e., their embodied *concept*[152], i.e., *type*[782]), such that it is uniquely decidable, by a human, whether a phenomenon satisfies the rule or not, i.e., is a member of the class, or not. (The recognition rule concept used here is due to Michael A. Jackson [138].)

584. **Recursion:** Recursion is a concept associated both with the *function definition*[316]s and with *data*[193] *type definition*[785]s. A function definition [a data type] is said to possess recursion if it is defined in terms of itself. (Cf. with the slightly different concept of *recursive*[585].)

585. **Recursive:** Recursive is a concept associated with *function*[310]s. A function is said to be recursive if, in the course of the evaluation of an invocation of the function,

that function is repeatedly invoked. (Cf. with the slightly different concept of *recursion*[584].)

586. **Reengineering:** By reengineering we shall, in a narrow sense, only consider the reengineering of business processes. Thus, to us, reengineering is the same as *business process reengineering*[101]. (Reengineering is also used in the wider sense of a major change to some already existing engineering *artefact*[55].)

587. **Reference:** A reference is the same as an *address*[22], a *link*[425], or a *pointer*[528]: something which refers to, i.e., designates something (typically something else).

588. **Referential transparency:** A concept which is associated with certain kinds of *programming*[550] or *specification language*[699] constructs, namely those whose *interpretation*[397] does not entail *side effects*. (A *pure functional programming language*[562] is said to be referentially transparent.)

589. **Refinement:** Refinement is a *relation*[599] between two *specification*[698]s: One specification, D, is said to be a refinement of another specification, S, if all the properties that can be observed of S can be observed in D. Usually this is expressed as $D \sqsubseteq S$. (Set-theoretically it works the other way around: in $D \supseteq S$, D allows behaviours not accounted for in S.)

590. **Refutable assertion:** A refutable assertion is an assertion that might be refuted (i.e., convincingly shown to be false). (Einstein's theory of relativity, in a sense, refuted Newton's laws of mechanics. Both theories amount to assertions.)

591. **Refutation:** A refutation is a statement that (convincingly) refutes an assertion. (Lakatos [146] drew a distinction between refutation (evidence that counts against a theory) and rejection (deciding that the original theory has to be replaced by another theory). We can still use Newton's theory provided we stay within certain boundaries, within which that theory is much easier to handle than Einstein's theory.)

592. **Regular expression:** To introduce the notion of regular expression we assume an *alphabet*[34], A, say finite. Basis clause: For any a in the alphabet, a is a regular expression. Inductive clause: If r and r' are regular expressions, then so are rr', (r), $r \mid r'$, and $r^\star$. (The denotation, $\mathcal{L}(r)$, of a regular expression r is defined as follows: (i) If r is of the form a, for a in the alphabet A, then $\mathcal{L}(a) = \{a\}$; (ii) if r is of the form $r'r''$ then $\mathcal{L}(r'r'') = \{s'{}^\frown s'' \mid s' \in \mathcal{L}(r'), s'' \in \mathcal{L}(r'')\}$; (iii) or if r is of the form (r') then $\mathcal{L}((r')) = \{s \mid s \in \mathcal{L}(r')\}$; (iv) or if r is of the form $r' \mid r''$ then $\mathcal{L}(r' \mid r'') = \{s \mid s \in \mathcal{L}(r') \vee s \in \mathcal{L}(r'')\}$; (v) or if r is of the form $r'^\star$ then $\mathcal{L}(r'^\star) = \{s \mid s = <> \vee s \in \mathcal{L}(r') \vee s' \in \mathcal{L}(r'r') \vee s' \in \mathcal{L}(r'r'r') \vee \ldots\}$ where $<>$ is the empty string, idempotent under concatenation.)

593. **Regular grammar:** See *regular syntax*.

594. **Regular language:** By a regular language we understand a *language*[417] which is the denotation of a *regular expression*[592]. (Some simple forms of *grammar*[325]s, that is, *regular syntax*[596]es, also generate regular languages.)

595. **Regulation:** A regulation stipulates that an *action*[12] be taken in order to remedy a previous action which "broke" a *rule*[638]. That is, a regulation is some text which designates a possibly composite *action*[12] which, in turn, denotes a state-to-state change which ostensibly results in a result state in which the rule now holds. Usually a domain regulation is paired with domain rule.

596. **Regular syntax:** A regular syntax is a *syntax*[733] which denotes (i.e., which *generate*[321]s) a *regular language*[594].

597. **Reification:** The result of a *reify*[598] action. (See also *data reification*[198], *operation reification*[497] and *refinement*[589].)

598. **Reify:** To regard (something *abstract*[1]) as a material or *concrete*[154] thing. (Our use of the term is more *operational*[494]: To take an *abstract*[1] thing and turn it into a less abstract, more *concrete*[154] thing.)

599. **Relation:** By a relation we usually understand either a mathematical *entity*[272] or an *information structure*[374] consisting of a set of (relation) tuples (like rows in a *table*[739]). The mathematical entity, a relation, can be thought of, also, as a possibly infinite set of n-groupings (i.e., *Cartesian*[107]s of the same *arity*[53]), such that if $(a, b, \cdots, c, d, \cdots, e, f)$ is such an n-tuple, then we may say that $(a, b, \cdots, c)$ (a relation argument) relates to $(d, \cdots, e, f)$ (a relation result). Thus *function*[310]s are special kinds of relations, namely where every argument relates to exactly one result. (Relations, as information structures, are well-known in *relational database*[600]s.)

600. **Relational database:** A *database*[195] whose *data*[193] *types* are (i) *atomic*[63] *values*, (ii) *tuples* of these, and *relations* seen as sets of *tuples*. (The relational database model is due to E.F. Codd [68].)

601. **Reliability:** A system being *reliable* — in the context of a machine being dependable — means some measure of continuous correct service, that is: Measure of time to *failure*[286]. (Cf. *dependability*[217] [being dependable].) (Reliability is a *dependability requirement*[218]. Usually reliability is considered a *machine*[436] property. As such, reliability is (to be) expressed in a *machine requirements*[438] document.)

602. **Renaming:** By renaming we mean *Alpha-renaming*[35]. (Renaming, in this sense, is a concept of the *Lambda-calculus*[412].)

603. **Rendezvous:** Rendezvous is a concept related to parallel processes. It stands for a way of synchronising a number, usually two, of processes. (In `CSP` the pairing of output (!) / input (?) clauses designating the same channel provides a language construct for rendezvous.)

604. **Representation abstraction:** By *representation abstraction* of [typed] values we mean a specification which does not hint at a particular data (structure) model, that is, which is not implementation biased. (Usually a representation abstraction (of data) is either *property oriented*[559] or is *model oriented*[461]. In the latter case it is then expressed, typically, in terms of mathematical entities such as sets, Cartesians, lists, maps and functions.)

605. **Requirements:** A condition or capability needed by a user to solve a problem or achieve an objective [133].

606. **Requirements acquisition:** The gathering and enunciation of *requirements*[605]. (Requirements acquisition comprises the activities of preparation, requirements *elicitation*[265] (i.e. *requirements capture*[608]) and preliminary requirements evaluation (i.e., requirements vetting).)

607. **Requirements analysis:** By *requirements analysis* we understand a reading of requirements acquisition *rough-sketch*[633] prescription units, (i) with the aim of forming concepts from these requirements prescription units, (ii) as well as with the aim of discovering inconsistencies, conflicts and incompleteness within these requirements prescription units, and (iii) with the aim of evaluating whether a requirements can be objectively shown to hold, and if so what kinds of tests (etc.) ought be devised.

608. **Requirements capture:** By requirements capture we mean the act of eliciting, of obtaining, of extracting, requirements from *stakeholder*[703]s. (For practical purposes requirements capture is synonymous with *requirements elicitation*[611].)

609. **Requirements definition:** Proper *definition*[210]al part of a *requirements prescription*[615].

610. **Requirements development:** By requirements development we shall understand the *development*[228] of a *requirements prescription*[615]. (All aspects are included in development: *requirements acquisition*[606], requirements *analysis*[39], requirements *model*[460]ling, requirements *validation*[800] and requirements *verification*[807].)

611. **Requirements elicitation:** By requirements elicitation we mean the actual extraction of *requirements*[605] from *stakeholder*[703]s.

612. **Requirements engineer:** A requirements engineer is a *software engineer*[692] who performs *requirements engineering*[613]. (Other forms of *software engineer*[692]s are *domain engineer*[247]s and *software design*[688]ers (cum *programmer*[547]).)

613. **Requirements engineering:** The engineering of the development of a *requirements prescription*[615], from identification of *requirements*[605] *stakeholder*[703]s, via *requirements acquisition*[606], *requirements analysis*[607], and *requirements prescription*[615] to requirements *validation*[800] and requirements *verification*[807].

614. **Requirements facet:** A requirements facet is a view of the requirements — "seen from a *domain description*[243]" — such as *domain projection*[255], *domain determination*[245], *domain instantiation*[253], *domain extension*[249], *domain fitting*[251] or *domain initialisation*[252].

615. **Requirements prescription:** By a *requirements*[605] *prescription*[540] we mean just that: the prescription of some requirements. (Sometimes, by requirements prescription, we mean a relatively complete and consistent specification of all requirements, and sometimes just a *requirements prescription unit*[616].)

616. **Requirements prescription unit:** By a *requirements*[605] *prescription*[540] unit we understand a short, "one or two liner", possibly *rough-sketch*[633], *prescription*[540] of some property of a *domain requirements*[258], an *interface requirements*[394], or a *machine requirements*[438]. (Usually requirements prescription units are the smallest textual, sentential fragments elicited from requirements *stakeholder*[703]s.)

617. **Requirements specification:** Same as *requirements prescription*[615] — the preferred term.

618. **Requirements unit:** By a requirements unit we mean a single sentence, i.e., a short expression of a "singular" *requirements*[605]. (A "full" (or complete) *requirements*[605] thus consists of (usually very many) *requirements unit*[618]s.)

619. **Requirements validation:** By requirements validation we rather mean the *validation*[800] of a *requirements prescription*[615].

620. **Resource:** From Old French *ressourse relief, resource, from resourdre to relieve, literally, to rise again, from Latin resurgere ... an ability to meet and handle a situation* [213] (being resourceful). (In computing we deal with computing resources such as *storage*[715], *time*[761] and further computing equipment. Many computing applications handle enterprise resources such as enterprise staff, production equipment, building or land space, production time, etc. In enterprise domains resources include monies, people, equipment, buildings, time and locations (geographical space).)

621. **Resource allocation:** The *allocation* of *resource*[620]s.

622. **Resource scheduling:** The *scheduling*[646] of *resource*[620]s.

623. **Retrieval:** Used here in two senses: The general (typically *database*[195]-oriented) sense of 'the retrieval [the fetching] of data (of obtaining information) from a repository of such'. And the special sense of 'the retrieval of an abstraction from a concretisation', i.e., abstracting a concept from a phenomenon (or another, more operational concept). (See the next entry for the latter meaning.)

624. **Retrieve function:** By a *retrieve function*[310] we shall understand a function that applies to *values* of some *type*[782], the "more concrete, operational" type, and yields *values* of some *type*[782] claimed to be more *abstract*[1]. (Same as *abstraction function*[5].)

625. **Rewrite:** The replacement of some text or structure by some other text, respectively structure. (See *rewrite rule*[626].)

626. **Rewrite rule:** A rewrite rule is a directed equation: $lhs = rhs$. The left- and right-hand sides are *patterns*. If some *text* can be decomposed into three parts, i.e., $text_0 = text_1{}^\frown text_2{}^\frown text_3$, where $text_1$ and/or $text_3$ may be empty texts, and where $text_2 = lhs$, then an application of the rewrite rule $lhs = rhs$ to $text_0$ yields $text_1{}^\frown rhs{}^\frown text_3$. (The equation $lhs = rhs$ is said to be directed in that this rule does not prescribe that a subtext equal to rhs is to be rewritten into lhs.)

627. **Rewrite system:** Rewrite systems are sets of *rewrite rule*[626]s used to compute, by repeatedly replacing subterms of a given formula with equal terms, until the simplest form possible is obtained [79]. (Rewrite systems form a both theoretically and practically interesting subject. They abound in instrumenting *theorem proving*[758], and the *interpretation*[397] of notably *algebraic semantics*[27] *specification language*[699]s, cf. `CafeOBJ` [81, 80] and `Maude` [67, 171, 59].)

628. **Rigorous:** Favoring rigor, i.e., being precise.

629. **Rigorous development:** Same as the composed meaning of the two terms *rigorous*[628] and *development*[228]. (We usually speak of a spectrum of development modes: *systematic development*[737], rigorous development and *formal development*[298]. Rigorous software development, to us, "falls" somewhere between the two other modes of development: (Always) complete *formal specification*[304]s are constructed, for all (phases and) stages of development; some, but usually not all *proof obligation*[555]s are expressed; and usually only a few are discharged (i.e., proved to hold).)

630. **Risk:** The Concise Oxford Dictionary [159] defines risk (noun) in terms of a hazard, chance, bad consequences, loss, etc., exposure to mischance. Other characterisations of the term risk are: someone or something that creates or suggests a hazard, and possibility of loss or injury.

631. **Robustness:** A *system*[736] is robust — in the context of a *machine*[436] being *dependable* — if it retains all its *dependability*[217] attributes (i.e., properties) after *failure*[286] and after *maintenance*[442]. (Robustness is (thus) a *dependability requirement*[218].)

632. **Root:** A root is a *node*[479] of a *tree*[777] which is not a sub*tree*[777] of a larger, *embedding* (*embedded*[266]) tree.

633. **Rough-sketch:** See next item.

634. **Rough sketch:** By a rough sketch — in the context of *descriptive software development*[690] *documentation* — we shall understand a *document*[237] text which describes something which is not yet consistent and complete, and/or which may still be too concrete, and/or overlapping, and/or repetitive in its descriptions, and/or with which the describer has yet to be fully satisfied.

635. **Route:** Same as *path*[517].

636. **Routine:** Same as *procedure*[543].

637. **RSL:** `RSL` stands for the `RAISE` [112] Specification Language [110, 31, 33, 34]. ()

638. **Rule:** A regulating principle. In the *rules and regulations*[640] facet context of modelling domain rules we shall understand a domain rule as some text whose meaning is a *predicate*[536] over a pair of suitably chosen domain *state*[705]s. We may assume that a domain *action*[12] or a domain *event*[281] takes place in the first of these states and results in the second of these states. If the predicate is true then we say that the rule has been obeyed, otherwise that it has not been obeyed. Usually a *regulation*[595] is attached to the rule. (We use the concept of rules in several different contexts: *rewrite rule*[626], *rule of grammar*[639] and *rules and regulations*[640].)

639. **Rule of grammar:** A grammar is made up of one or more rules. A rule has a (left-hand-side) *definiendum*[207] and a (right-hand-side) *definiens*[208]. The definiendum is usually a single *identifier*[351]. The definiens is usually a possibly empty string of *identifier*[351]s. These identifiers are either *terminal*[750]s or *nonterminal*[484]s. A definiendum identifier is a nonterminal. In a grammar all nonterminals have a defining rule. Those identifiers which do not appear as a definiendum of a rule are thence considered terminals.

640. **Rules and regulations:** By rules and regulations we mean guidelines that are intended to be adhered to by the enterprise staff and enterprise customers (i.e., users, clients) in conducting their "business", i.e., their actions within, and with, the enterprise. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445] and *human behaviour*[345].)

641. **Run time:** The time (or time interval) during which a software *program*[545] is subject to *interpretation*[397] by a computer. (The term run time is usually deployed in order to distinguish between that concept and the concept of *compile time*[127].)

. . . . $\mathcal{S}$

642. **Safety:** By safety — in the context of a *machine*[436] being *dependable* — we mean some measure of continuous delivery of service of either correct service, or incorrect service after benign *failure*[286], that is, measure of time to catastrophic failure. (Safety is a *dependability requirement*[218]. Usually safety is considered a *machine*[436] property. As such safety is (to be) expressed in a *machine requirements*[438] *document*[237].)

643. **Safety critical:** A *system*[736] whose *failure*[286] may cause injury or death to human beings, or serious loss of property, or serious disruption of services or production, is said to be safety critical.

644. **Satisfiable:** A *predicate*[536] is said to be *satisfiable* if it is true for at least one *interpretation*[397]. (In this context think of an interpretation as a *binding*[88] of all *free*[305] *variable*[803]s of the predicate expression to *value*[802]s. Cf. *valid*[799].)

645. **Schedule:** A schedule is a *syntactic composite*[133] *concept*[152]. A schedule is a *prescription*[540] for (usually where and) when some *resources* are to be present, i.e., *information*[373] about being spatially and temporally available. (As such a schedule usually also includes some *allocation*[33] *information*[373].)

646. **Scheduling:** The act of providing, of constructing, a *schedule*[645].

647. **Schema:** A structured framework or plan. (We shall also use the term 'schema' in connection with, i.e., as a *rewrite rule*[626] and some *axioms* that apply to, for example, applicative program texts and rewrite into imperative program texts.)

648. **Scheme:** See *schema*[647].

649. **Scope:** We shall use the term scope in two sufficiently different senses: (1) In *programming*[550] the scope of an *identifier*[351] is the region of a *program*[545] text within which it represents a certain thing. This usually extends from the place where it is declared to the end of the smallest enclosing *block*[89] (begin/end or procedure/function body). An inner block may contain a redeclaration of the same identifier, in which case the scope of the outer declaration does not include (is shadowed, occluded, blocked off or obstructed by) the scope of the inner. (2) We also use the term scope in the context of the degree to which a project *scope* and *span* extends: Scope being the "larger, wider" delineation of what a project "is all about", *span*[697] being the "narrower", more precise extent.

650. **Scope check:** Usually a function performed by a *compiler*[125] concerning the definition (declaration) and places of use of identifiers of *program*[545] texts. (Thus the use of *scope*[649] is that of the first (1) sense of item 649.)

651. **Script:** A plan of action. By a domain script we shall, more specifically, understand the structured, almost, if not outright, formally expressed, wording of *rules and regulations*[640] of behaviour. See also *license*[424] and *contract*[181].

652. **Secure:** To properly define the concept of secure, we first assume the concept of an authorised user. Now, a *system*[736] is said to be secure if an un-authorised user, when supposedly making use of that system, (i) is not able to find out what the system does, (ii) is not able to find out how it does 'whatever' it does do, and (iii), after some such "use", does not know whether he/she knows! (The above characterisation represents an unattainable proposition. As a characterisation it is acceptable. But it does not hint at ways and means of implementing secure systems. Once such a system is believed implemented the characterisation can, however be used as a guide in devising tests that may reveal to which extent the system indeed is secure. Secure

systems usually deploy some forms of authorisation and encryption mechanisms in guarding access to system functions.)

653. **Security:** When we say that a *system*[736] exhibits security we mean that it is *secure*[652]. (Security is a *dependability requirement*[218]. Usually security is considered a *machine*[436] property. As such security is (to be) expressed in a *machine requirements*[438] document.)

654. **Selector:** By a selector (a selector function) we understand a function which is applicable to *values* of a certain, defined, composed *type*[782], and which yields a proper component of that value. The function itself is defined by the *type definition*[785].

655. **Semantics:** Semantics is the study and knowledge [incl. specification] of meaning in language [70]. (We make the distinction between the *pragmatics*[534], the semantics and the *syntax*[733] of languages. Leading textbooks on semantics of programming languages are [78, 116, 201, 205, 218, 228].)

656. **Semantic function:** A semantics function is a function which when applied to *syntactic values* yields their *semantic values*.

657. **Semantic type:** By a semantic type we mean a *type*[782] that defines *semantic values*.

658. **Semiotics:** Semiotics, as used by us, is the study and knowledge of *pragmatics*[534], *semantics*[655] and *syntax*[733] of language(s).

659. **Sensor:** A sensor can be thought of as a piece of *technology*[746] (an electronic, a mechanical or an electromechanical device) that senses, i.e., measures, a physical *value*[802]. (A sensor is in contrast to an *actuator*[17].)

660. **Sentence:** (i) A word, clause, or phrase or a group of clauses or phrases forming a syntactic unit which expresses an assertion, a question, a command, a wish, an exclamation, or the performance of an action, that in writing usually begins with a capital letter and concludes with appropriate end punctuation, and that in speaking is distinguished by characteristic patterns of stress, pitch and pauses; (ii) a mathematical or logical statement (as an equation or a proposition) in words or symbols [213].

661. **Sequential:** Arranged in a sequence, following a linear order, one after another.

662. **Sequential process:** A process is sequential if all its observable actions can be, or are, ordered in sequence.

663. **Server:** By a server we mean a *process*[544] or a *behaviour*[79] which *interact*[391]s with another process or behaviour (i.e., a *client*[116]) in order for the server to perform some *action*[12]s on behalf of the client.

664. **Set:** We understand a set as a mathematical entity, something that is not mathematically defined, but is a concept that is taken for granted. (Thus by a set we understand the same as a collection, an aggregation, of distinct entities. Membership (of an entity) of a set is also a mathematical concept which is likewise taken for granted, i.e., undefined.)

665. **Set theoretic:** We say that something is set theoretically understood or explained if its understanding or explanation is based on *sets*.

666. **Shared action:** By a shared action we mean an action that can only be partly computed by the *machine*[436]. That is, the *machine*[436], in order to complete an action, may have to inquire with the *domain*[239] (in order, say, to extract some measurable, time-varying simple entity attribute value) in order to proceed in its computation.

667. **Shared behaviour:** By a shared behaviour we mean a behaviour many of whose actions and events occur both in the *domain*[239] and, in some encoded form, and in the same squence, in the *machine*[436].

668. **Shared concept:** See *shared phenomenon*[676].

669. **Shared data:** See *shared phenomenon*[676].

670. **Shared data initialisation:** By shared data initialisation we understand an *operation*[493] that (initially) creates a *data structure*[199] that reflects, i.e., models, some *shared phenomenon*[676] in the *machine*[436]. (See also *shared data refreshment*[672].)

671. **Shared data initialisation requirements:** *Requirements* for *shared data initialisation*[670]. (See also *computational data+control requirements*[146], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448], and *machine-machine dialogue requirements*[437].)

672. **Shared data refreshment:** By shared data refreshment we understand a *machine*[436] *operation*[493] which, at prescribed intervals, or in response to prescribed events updates an (originally initialised) *shared data*[669] structure. (See also *shared data initialisation*[670].)

673. **Shared data refreshment requirements:** *Requirements* for *shared data refreshment*[672]. (See also *computational data+control requirements*[146], *shared data initialisation requirements*[671], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448], and *machine-machine dialogue requirements*[437].)

674. **Shared event:** By a shared event we mean an event whose occurrence in the *domain*[239] need be communicated to the *machine*[436] – and, vice-versa, an event whose occurrence in the *machine*[436] need be communicated to the *domain*[239].

675. **Shared information:** See *shared phenomenon*[676].

676. **Shared phenomenon or concept:** A shared phenomenon (or concept) is a phenomenon (respectively a concept) which is present in some *domain*[239] (say in the form of facts, *knowledge*[407] or *information*[373]) and which is also represented in the *machine*[436] (say in the form of some *entity*[272], simple, action, evemt or behaviour). A phenomenon of a domain, when shared, becomes a concept of the machine.

677. **Shared simple entity:** By a shared simple entity we mean a simple entity which both occurs in the *domain*[239] (as a phenomenon or a concept) and in the *machine*[436]. Simple entities that are shared between the domain and the machine must initially be input to the machine. Dynamically arising simple entities must likewise be input and all such machine entities must have their attributes updated, when need arise. Requirements for shared simple entities thus entail requirements for their representation and for their human/machine and/or machine/machine transfer dialogue.

678. **Side effect:** A language construct that designates the modification of the state of a system is said to be a side-effect-producing construct. (Typical side effect constructs are assignment, input and output. A *programming language*[551] "without side effects" is said to be a *pure functional programming language*[562].)

679. **Sign:** Same as *symbol*[728].

680. **Signature:** See *function signature*[318].

681. **Simple entity:** By a simple entity we shall loosely understand an individual, *static*[708] or *inert*[367] *dynamic*[260] (We shall take the narrow view of a simple entity, being in contrast to an *action*[12], an *event*[281] and a *behaviour*[79]; that simple entities "roughly correspond" to what we shall think of as *value*[802]s. We shall further allow simple entities to be either *atomic*[63] or *composite*[133], i.e., in the latter case having decomposable *subentities*[721]. Simple entities have *attribute*[69]s. Composite entities have *attribute*[69]s, *subentities*[721] and a *mereology*[451], the latter explains how the subentities are formed into the simple entity. We consider *simple entities*[681] to be one of the four kinds of *entities*[272] that the `Triptych` "repeatedly" considers. The other three are: *action*[12]s, *event*[281]s and *behaviour*[79]s. Consideration of these are included in the specification of all *domain facet*[250]s and all *requirements facet*[614]s.)

682. **Simplification:** ()

683. **Simulation:** The imitation of the functioning of one system or process by means of the functioning of another. (Attempting to predict aspects of the behaviour of some system by creating an approximate (mathematical) model of it. This can be done by physical modelling, by writing a special-purpose computer program or using a more general simulation package, probably still aimed at a particular kind of simulation [108].)

684. **Soft real time:** By soft real time we mean a *real time*[580] property where the exact, i.e., absolute timing, or time interval, is only of loose, approximate essence. (Cf., *hard real time*[330].)

685. **Software:** By software we understand not only the code that when "submitted" to a computer enables desired computations to take place, but also all the documentation that went into its development (i.e., its *domain description*[243], *requirements specification*[617], its complete *software design*[688] (all stages and steps of *refinement*[589] and *transformation*[771]), the *installation manual*[386], *training manual*[767], and the *user manual*[798]).

686. **Software component:** Same as *component*[131].

687. **Software architecture:** By a software architecture we mean a first kind of specification of software — after requirements — one which indicates **how** the software is to handle the given requirements in terms of *software components* and their interconnection — though without detailing (i.e., designing) these software components.

688. **Software design:** By software design we shall understand the determination of which *components*, which *modules* and which *algorithms* shall implement the *requirements*[605] — together with all the *documents* that usually make up properly documented *software*[685]. (Software design entails *programming*[550], but programming is a "narrower" field of activity than software design in that programming usually excludes many documentation aspects.)

689. **Software design specification:** The *specification*[698] of a *software design*[688].

690. **Software development:** To us, software development includes all three phases of *software*[685] *development*[228]: *domain development*[246], *requirements development*[610] and *software design*[688].

691. **Software development project:** A *software*[685] development project is a planning, research and development project whose aim is to construct *software*[685].

692. **Software engineer:** A software engineer is an *engineer*[269] who performs one or more of the functions of *software engineering*[693]. (These functions include *domain engineering*[248], *requirements engineering*[613] and *software design*[688] (incl. *programming*[550]).)

693. **Software engineering:** The confluence of the science, logic, discipline, craft and art of *domain engineering*, *requirements engineering* and *software design*.

694. **Sort:** A sort is a collection, a structure, of, at present, further unspecified entities. (That is, same as an *algebraic type*. When we say "at present, further unspecified", we mean that the (values of the) sort may be subject to constraining axioms. When we say "a structure", we mean that "this set" is not necessarily a *set*[664] in the simple sense of mathematics, but may be a collection whose members satisfy certain

interrelations, for example, some *partially ordered set*, some *neighbourhood set* or other.)

695. **Sort definition:** The *definition*[210] of a *sort*[694]. (Usually a sort definition consists of the (introduction of) a type name, some (typically *observer function*[490] and *generator function*[323]) *signatures*, and some *axioms* relating sort *values* and *functions*.)

696. **Source program:** By a source program we mean a *program*[545] (text) in some *programming language*[551]. (The term source is used in contrast to target: the result of compiling a source text for some target *machine*[436].)

697. **Span:** Span is here used, in contrast to *scope*[649], more specifically in the context of the degree to which a project *scope* and *span* extend: Scope being the "larger, wider" delineation of what a project "is all about", *span*[697] being the "narrower", more precise extent.

698. **Specification:** We use the term 'specification' to cover the concepts of *domain description*[243]s, *requirements prescription*[615]s and *software design*[688]s. More specifically a specification is a *definition*[210], usually consisting of many definitions.

699. **Specification language:** By a specification language we understand a *formal*[296] *language*[417] capable of expressing *formal*[296] *specifications*. (We refer to such formal specification languages as: `Alloy` [137], `ASM` [198], `Event B` [2, 4, 60], `CafeOBJ` [80, 81], `RSL` [110, 111], `VDM-SL` [52, 107] and `Z` [209, 210, 233, 126].)

700. **Stack:** A stack is an *abstract data type*[4] with a stack data structure and, typically, the following operations: push (onto the top of the stack), pop (remove from the top of the stack). Axioms then determine specific stack properties. ()

701. **Stack activation:** Generally: The topmost element of a stack. Specifically, when a stack is used to record the local states of blocks of a block-structured programming language's blocks or procedure bodies (they are also blocks), then each stack element, i.e., each stack activation, records such a local state and — what is known as static and dynamic — pointers chain such activations together which correspond to the lexicographic scope of the program, respectively the calling invocation of the blocks. (We refer to Vol. 2, Chap. 16, Sect. 16.6.1 for a thorough treatment of stack activations.)

702. **Stage:** (i) By a development stage we shall understand a set of development activities which either starts from nothing and results in a complete phase documentation, or which starts from a complete phase documentation of stage kind, and results in a complete phase documentation of another stage kind. (ii) By a development stage we shall understand a set of development activities such that some (one or more) activities have created new, externally conceivable (i.e., observable) properties of what is being described, whereas some (zero, one or more) other activities have refined

previous properties. (Typical development stages are: *domain*[239] *intrinsics*[399], *domain*[239] *support technologies*, *domain*[239] *management and organisation*[445], *domain*[239] *rules and regulations*[640], etc., and *domain requirements*[258], *interface requirements*[394], and *machine requirements*[438], etc.)

703. **Stakeholder:** By a *domain*[239] (*requirements*[605], *software design*[688])[18] stakeholder we shall understand a person, or a group of persons, "united" somehow in their common interest in, or dependency on the domain (requirements, software design); or an institution, an enterprise, or a group of such, (again) characterised (and, again, loosely) by their common interest in, or dependency on the domain (requirements, software design). (The three stakeholder groups usually overlap.)

704. **Stakeholder perspective:** By a *stakeholder*[703] perspective we shall understand the, or an, understanding of the *universe of discourse*[793] shared by the specifically identified stakeholder group — a view that may differ from one stakeholder group to another stakeholder group of the same universe of discourse.

705. **State:** By a state we shall, in the context of computer *programs*, understand a summary of past *computations*, and, in the context of *domains*, a suitably selected set of *dynamic*[260] *entities*.

706. **Statechart:** The Statechart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [120].)

707. **Statement:** We shall take the rather narrow view that a statement is a *programming language*[551] construct which *denotes* a *state*[705]-to-state function. (Pure expressions are then programming language constructs which denote state-to-value functions (i.e., with no *side effect*[678]), whereas "impure" expressions, also called clauses, denote state-to-state-and-value functions.)

708. **Static:** An *entity*[272] is static if it is not subject to *actions* that change its *value*[802]. (In contrast to *dynamic*[260].)

709. **Static semantics:** The concept of static semantics is one that applies to *syntactic entities*, typically *programs* or *specifications* of *programming language*[551]s, respectively *specification language*[699]s. The static semantics of such a language is now a *predicate*[536] that applies to *programs* (respectively *specifications*) and yields true if the *program*[545] (*specification*[698]) is syntactically well formed according to the static semantics criteria, typically that certain relations are satisfied between dispersed parts of the *program*[545] (*specification*[698]) texts.

710. **Static typing:** Enforcement of *type checking* at *compile time*[127]. (A *programming language*[551] (or a *specification language*[699]) is said to be statically typed if its *programs* (resp. *specifications*) can be statically *type checked*.)

[18]These three areas of concern form three *universes of discourse*.

711. **Step:** By a development step we shall understand a refinement of a domain description (or a requirements prescription, or a software design specification) module, from a more abstract to a more concrete description (or a more concrete requirements prescription, or a more concrete software design specification).

712. **Stepwise development:** By a stepwise development we shall understand a *development*[228] that undergoes *phases*, *stages* or *steps* of development, i.e., can be characterised by pairs of two adjoining *phase*[523] *steps*, a last *phase*[523] *step*[711] and a (first) next *phase*[523] *step*[711], or two adjoining *stage*[702] *steps*.

713. **Stepwise refinement:** By a stepwise refinement we understand a pair of adjoining *development*[228] *steps* where the transition from one *step*[711] to the next *step*[711] is characterised by a *refinement*[589]. (Refinement is thus always stepwise refinement.)

714. **Store:** Same as *storage*[715]; see next.

715. **Storage:** By storage we shall understand a *function*[310] from *locations* to *values*. (Thus we emphasise the mathematical character of storage rather than any technological character (such as disk storage, etc.).)

716. **Strategy:** [213]: (1) The science and art of employing the political, economic, psychological, and military forces of a nation or group of nations to afford the maximum support to adopted policies in peace or war; (2) an adaptation or complex of adaptations (as of behaviour or structure) that serves or appears to serve an important function in achieving evolutionary success. (Applied to business enterprises the above "translates" into: the science and art of employing the economic and other resources of an enterprise to achieve maximum support for adopted enterprise policies: enterprise products & service profile, market share, growth, profitability, etc.)

717. **Strict function:** A strict function is a function which yields **chaos** (i.e., is undefined) if any of the function arguments are undefined (i.e., **chaos**). (In RSL the logical connectives are not strict. All other functions, built-in or defined, are strict.)

718. **Strongest post-condition:** See *weakest pre-condition*[811].

719. **Structure:** The term 'structure' is understood rather loosely. Normally we shall understand a structure as a mathematical structure, such as an *algebra*[26], or a *predicate logic*[537], or a *Lambda-calculus*[412], or some defined abstraction (a *scheme*[648] or a *class*[114]). (Set theory is a (mathematical) structure. So are RSL's Cartesian, list and map data types.)

720. **Structural operational semantics:** By a structural operational semantics we understand an *operational semantics*[496] which is expressed in terms of a number of *transition rule*[773]s. (See [195].)

721. **Subentity:** A subentity is a proper part of a (thus) non-*atomic*[63] *entity*[272]. (Do not confuse a subentity of an entity with an *attribute*[69] of that entity (or of that subentity).)

722. **Substitution:** By substitution we mean the replacement of a token (viz.: an identifier) by a structure, usually a text. (The most common form of substitution is that of *Beta-reduction*[84] (in the *Lambda-calculus*[412]). Substitution is a "simpler" form of *rewriting*.)

723. **Subroutine:** Same as *routine*[636].

724. **Subtype:** To speak of a subtype we must first be able to speak of a *type*[782], i.e., colloquially, a (suitably structured) set of *value*[802]s. A subtype of a type is then a (suitably structured) and proper subset of the values of the type. (Usually we shall, in RSL, think of a predicate, p, that applies to all members of the type, T, and singles out a proper subset whose elements satisfy the predicate: $\{a \mid a : T \cdot p(a)\}$.)

725. **Support technology:** By a support technology we understand a *facet*[285] of a *domain*[239], one which reflects its (current) dependency on mechanical, electro-mechanical, electronic and other technologies (i.e., tools) in order to carry out its *business process*[99]es. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *management and organisation*[445], *rules and regulations*[640] and *human behaviour*[345].)

726. **Surjection:** A *surjective function*[727] represents surjection. (See also *bijection*[86] and *injection*[379].)

727. **Surjective function:** A *function*[310] which maps *value*[802]s of its postulated *definition set*[211] into all of its postulated *range set*[577] is called surjective. (See also *bijective function*[87] and *injective function*[380].)

728. **Symbol:** Something that stands for or suggests something else, that is, an arbitrary or conventional sign used in writing.

729. **Synchronisation:** By synchronisation we understand the act of ensuring *synchronism*[730] between occurrence of designated *events* in two or more *processes*. (Usually synchronisation between occurrence of designated events in two or more processes entails the exchange of *information*[373], i.e., *data*[193], between these processes, i.e., *communication*[122].)

730. **Synchronism:** A chronological arrangement of *event*[281]s.

731. **Synchronous:** Happening, existing, or arising at precisely the same *time*[761] indicating *synchronism*[730].

732. **Synopsis:** By a synopsis we shall understand a composition of *informative documentation*[375] and *rough-sketch*[633] *description*[220] of some project.

733. **Syntax:** By syntax we mean (i) the ways in which words are arranged to show meaning (cf. *semantics*) within and between sentences, and (ii) rules for forming *syntactically correct* sentences. (See also *regular syntax*, *context-free syntax*, *context-sensitive syntax* and *BNF* for specifics.)

734. **Synthesis:** The construction of an *artefact*[55].

735. **Synthetic:** Result of *synthesis*[734]: not *analytic*[40].

736. **System:** A regularly interacting or interdependent group of phenomena or concepts forming a whole, that is, a group of devices or artificial objects or an organization forming a network especially for producing something or serving a common purpose. (This book will have its own characterisation of the concept of a system (commensurate, however, with the above encircling characterisation); cf. Vol. 2, Sect. 9.5's treatment of system.)

737. **Systematic development:** Systematic development of software is *formal development "lite"!* (We usually speak of a spectrum of development modes: systematic development, *rigorous development*[629], and *formal development*[298]. Systems software development, to us, is at the "informal" extreme of the three modes of development: *formal specification*[304]s are constructed, but maybe not for all stages of development; and usually no proof obligations are expressed, let alone proved. The three volumes of this series of textbooks in software engineering can thus be said to expound primarily the systematic approach.)

738. **Systems engineering:** By systems engineering we shall here understand computing systems engineering: The confluence of developing *hardware*[331] and *software*[685] solutions to *requirements*[605].

. . . . $\mathcal{T}$

739. **Table:** By a table we understand an *information structure*[374] which can be thought of as an ordered *list*[428] of rows, each row consisting of an ordered *list*[428] of entries, each consisting of some *information*[373]. (When thought of as a *data structure*[199], a table is normally thought of as either a matrix or a *relation*[599].)

740. **Tangibility:** Noun of *tangible*[742].

741. **Tactic:** [213]: (1) a device for accomplishing an end (2) a method of employing forces in combat.Applied to business enterprises the above "translates" into: a set of resource-dependent actions thought to accomplish a strategy.

742. **Tangible:** Physically manifest. That is, can be humanly sensed: heard, seen, smelled, tasted, or touched, or physically measured by a physical apparatus: length (meter, m), mass (kilogram, kg), time (second, s), electric current (Ampere, A), thermodynamic temperature (Kelvin, K), amount of substance (mole, mol), luminous intensity (candela, cd).

743. **Target program:** The concept of target program stems from the fact that *programs* of ordinary *programming languages* need to be translated into some intermediary language or final machine, i.e., computer hardware, language, before their designated computations (i.e., interpretations) can take place. By a target program we understand such an intermediary or final program. (Besides the final target languages made up from the repertoire of computer hardware instructions and computer (bit, byte, half-word, word, double-word and variable field) data formats, special intermediary languages have been devised: `P-code` [94] (into which `Pascal` programs can be translated) [229, 129, 230, 140, 231, 135, 7], `A-code` [93] (into which `Ada` programs can be translated) [56, 225], etc.)

744. **Taxonomy:** By *taxonomy* is meant [159]: "classification, especially in relation to its general laws or principles; that department of science, or of a particular science or subject, which consists in or relates to classification.".

745. **Technique:** A procedure, an approach, to accomplish something.

746. **Technology:** We shall in these volumes be using the term technology to stand for the results of applying scientific and engineering insight. This, we think, is more in line with current usage of the term IT, information technology.

747. **Temporal:** Of or relating to time, including sequence of time, or to time intervals (i.e., durations).

748. **Temporal logic:** A(ny) *logic*[432] over *temporal*[747] *phenomena*. (We refer to Vol. 2, Chap. 15 for our survey treatment of some temporal logics.)

749. **Term:** From [159]: A word or phrase used in a definite or precise sense in some particular subject, as a science or art; a technical expression. More widely: any word or group of words expressing a notion or conception, or denoting an object of thought. (Thus, in `RSL`, a term is a *clause*[115], an *expression*[282], a *statement*[707], which has a *value*[802] (statements have the **Unit** value).)

750. **Terminal:** By a terminal we shall mean a terminal *symbol*[728] which (in contrast to a *nonterminal*[484] symbol) designates something specific.

751. **Termination:** The concept of termination is associated with that of an *algorithm*[31]. We say that an algorithm, when subject to *interpretation*[397] (colloquially: 'execution'), may, or may not terminate. That is, may halt, or may "go on forever, forever looping". (Whether an algorithm terminates is *undecidable*[792].)

752. **Terminology:** By terminology is meant ([159]): The doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature.

753. **Term rewriting:** Same as *rewriting*.

754. **Test:** A test is a means to conduct *testing*[755]. (Typically such a test is a set of data values provided to a program (or a specification) as values for its *free*[305] *variables*. *Testing* then evaluates the program (resp., interprets (symbolically) the specification) to obtain a result (value) which is then compared with what is (believed to be) the, or a, correct result. See Vol. 3, Sects. 14.3.2, 22.3.2 and 29.5.3 for treatments of the concept of test.)

755. **Testing:** Testing is a systematic effort to refute a claim of correctness of one (e.g., a concrete) specification (for example a program) with respect to another (the abstract) specification. (See Vol. 3, Sects. 14.3.2, 22.3.2, and 29.5.3 for treatments of the concept of testing.)

756. **Theorem:** A theorem is a *sentence*[660] that is *provable* without assumptions, that is "purely" from *axioms* and *inference rules*.

757. **Theorem prover:** A mechanical, i.e., a computerised means for *theorem proving*[758]. (Well-known theorem provers are: `PVS` [183, 184] and `HOL/Isabelle` [179].)

758. **Theorem proving:** The act of *proving theorems*.

759. **Theory:** A formal theory is a *formal*[296] *language*[417], a set of *axioms* and *inference rules* for *sentences* in this language, and is a set of *theorems* proved about sentences of this language using the axioms and inference rules. A mathematical theory leaves out the strict formality (i.e., the *proof*[554]system) requirements and relies on mathematical proofs that have stood the social test of having been scrutinised by mathematicians.

760. **Three-valued logic:** Standard logics are two value: **true** and **false**. A three-valued logic is a logic for which the Boolean connectives accept a third value, usually referred to as the *undefined*, or *chaotic* (non-*termination*[751] of operand *expression*[282] *evaluation*[280]). (There can be, and are, many three-valued logics. `RSL` has one set of definitions of the outcome of Boolean ground term evaluation with **chaos** operands. `LPF` is a logic for partial functions sugggested as a logic for `VDM` [16, 65]. John McCarthy [167] first broached the topic of three-valued logics in computing.)

761. **Time:** Time is often a notion that is taken for granted. But one may do well, or better, in trying to understand time as some point set that satisfies certain axioms. Time and space are also often related (via [other] physically manifest "things"). Again their interrelationship needs to be made precise. (In comparative concurrency semantics one usually distinguishes between linear time and branching time semantic equivalences [222]. We refer to our treatment of time and space in Vol. 2 Chap. 5, to Johan van Benthem's book *The Logic of Time* [221], and to Wayne D. Blizard's paper *A Formal Theory of Objects, Space and Time* [57].)

762. **Token:** Something given or shown as an identity. (When, in `RSL`, we define a *sort*[694] with no "constraining" axioms, we basically mean to define a set of tokens.)

763. **Tool:** An instrument or apparatus used in performing an operation. (The tools most relevant to us, in software engineering, are the *specification*[698] and *programming language*[551]s as well as the *software*[685] packages that aid us in the development of (other) software.)

764. **Topology:** (i) A branch of mathematics concerned with those properties of geometric configurations (as point sets) which are unaltered by elastic deformations (as a stretching or a twisting) that are *homeomorphisms*; (ii) the set of all open subsets of a *topological space* (i.e., being or involving properties unaltered under a homeomorphism [continuity and connectedness are topological properties]) [213].

765. **Total algebra:** A total *algebra* is an algebra all of whose functions are total over the carrier.

766. **Trace:** The concept of trace is linked to the concept of a *behaviour*[79]. Trace is then defined as a sequence of *actions* and *events*. ()

767. **Training manual:** A *document*[237] which can serve as a basis for a (possibly self-study) course in how to use a *computing system*[151]. (See also *installation manual*[386] and *user manual*[798].)

768. **Transaction:** General: A communicative action or activity involving two *agent*[24]s that reciprocally influence each other. (Special: The term transaction has come to be used, in computing, notably in connection with the use of database management systems (DBMS, or similar multiuser systems): A transaction is then a unit of interaction with a DBMS (etc.). To further qualify as being a transaction, it must be handled, by the DBMS (etc.), in a coherent and reliable way independent of other transactions.)

769. **Transduce:** To convert (a physical signal, or a message) into another form.

770. **Transducer:** A device that is actuated by power from one system and supplies power usually in another form to a second system. (*Finite state machines* and *pushdown stack machines* are considered transducers.)

771. **Transformation:** The operation of changing one configuration or expression into another in accordance with a precise rule. (We consider the results of *substitution*[722], of *translation* and of *rewriting* to be transformations of what the *substitution*[722], the *translation* and the *rewriting* was applied to.)

772. **Transition:** Passage from one state, stage, subject or place to another; a movement, development, or evolution from one form, stage or style to another [213].

773. **Transition rule:** A *rule*[638], of such a form that it can specify how any of a well-defined class of *states* of a *machine*[436] may make *transitions* to another state, possibly *nondeterministically* to any one of a well-defined number of other states. (The seminal

1981 report *A Structural Approach to Operational Semantics*, by Gordon D. Plotkin [192], set a de facto standard for formulating transition rules (exploring their theoretical properties and uses).)

774. **Translate:** See *translation*[775].

775. **Translation:** An act, process or instance of translating, i.e., of rendering from one language into another.

776. **Translator:** Same as a *compiler*[125].

777. **Tree:** An *acyclic*[18] un-*directed graph*[232]. Thus a tree (i) has a *root*[632], which is a *node*[479], and (ii) zero, one or more, possibly (*branch*[97] or *edge*[262]) *label*[410]led subtrees. Trees or subtrees with no further subtrees have their roots being equated with leaves. Nodes may be labelled. (This characterisation allows for trees with no labels, with only labelled nodes, with only labelled branches, with labelled nodes and branches, or with only some nodes and some branches being labelled. The characterisation usually is interpreted as only allowing finite trees, but one could dispense of the "finite applicability" of the above (i–ii) clauses, to allow infinite trees. The branch concept, akin to the *edge*[262] concept, amounts, however, to a directed edge, i.e., an *arrow*[54]. We refer specifically to *parse tree*[512]s. See also a "redefinition" of trees as found just below, under *tree traversal*[778], including Fig. 13.)

778. **Tree traversal:** A way of visiting (all) the *node*[479]s of a *tree*[777]. Redefine the notion of a *tree*[777] as just given above: Now a tree is a root node and an ordered set (i.e., like a list) of zero, one or more subtrees; each subtree is a tree. Roots are labelled. Hence subtrees are labelled. A tree with an empty set of subtrees is called a leaf. Their roots are the leaves. A tree traversal is now a way of visiting, in some order, as indicated by the order of subtrees, (all) the nodes: the root, the branch nodes and leaves, of a tree. (See the tree of Fig. 13 on the next page. It will be referred to in entries *in-order*[381], *post-order*[533] and *pre-order*[538].)

779. **Triptych:** An ancient Roman writing tablet with three waxed leaves hinged together; a picture (as an altarpiece) or carving in three panels side by side [213]. (The trilogy of the *phases* of *software development*[690], *domain engineering*[248], *requirements engineering*[613] and *software design*[688] as promulgated by this trilogy of volumes!)

780. **Tuple:** A grouping of values. (Like 2-tuplets, quintuplets, etc. Used extensively, at least in the early days, in the field of relational databases — where a tuple was like a row in a relation (i.e., table).)

781. **Turing machine:** A hypothetical machine defined in 1935–1936 by Alan Turing and used for computability theory proofs. It can be understood as consisting of a *finite state machine*[290] and an infinitely long "tape" with symbols (chosen from some finite set) written at regular intervals. A pointer marks the current position and the

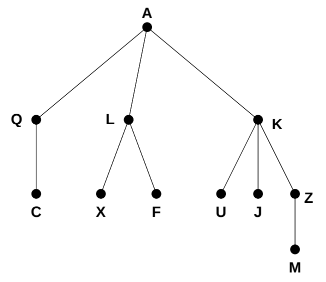

Figure 13: A labelled, ordered tree

machine is in one of states. At each step the machine reads the symbol at the current position on the tape. For each combination of current state and symbol read, the finite state machine specifies the new state and either a symbol to write to the tape or a direction to move the pointer (left or right) or to halt [108]. (Turing machines are equivalent, in computational power, to the *Lambda-calculus*[412].)

782. **Type:** Generally a certain kind of set of *value*s. (See *algebraic type*, *model-oriented type*, *programming language type* and *sort*.)

783. **Type check:** The concept of type check arises from the concepts of *function signatures* and function *arguments*. If arguments are not of the appropriate type then a type check yields an *error*[278] result. (By appropriate *static*[708] *typing*[788] of *declarations* of *variables* of a *programming language*[551] or a *specification language*[699] one can perform static type checking (i.e., at *compile time*[127]).)

784. **Type constructor:** A type constructor is an operation that applies to *types* and yields a *type*[782]. (The type constructors of `RSL` include the power set constructors: **-set** and **-infset**, the Cartesian constructor: $\times$, the list constructors: * and $^\omega$, the map constructor: $\xrightarrow[m]{}$, the total and partial function space constructors: $\to$ and $\xrightarrow{\sim}$, the union type constructor: $|$, and others.)

785. **Type definition:** A type definition semantically associates a *type name*[787] with a *type*[782]. Syntactically, as, for example, in `RSL`, a type definition is either a *sort*[694] definition or is a *definition*[210] whose right-hand side is a *type expression*[786].

786. **Type expression:** A type expression semantically denotes a *type*[782]. Syntactically, as, for example, in `RSL`, a type expression is an expression involving *type names* and *type constructors*, and, rarely, *terminals*.

787. **Type name:** A type name is usually just a simple *identifier*[351].

788. **Typing:** By typing we mean the association of *types* with *variables*. (Usually such an association is afforded by pairing a *variable*[803] *identifier*[351] with a *type name*[787] in the variable *declaration*[201]. See also *dynamic typing*[261] and *static typing*[710].)

. . . . $\mathcal{U}$

789. **UML:** Universal Modelling Language. A hodgepodge of notations for expressing requirements and designs of computing systems. (Vol. 2, Chaps. 10, and 12–14 outlines our attempt to "UML"-ize formal techniques.)

790. **Universal algebra:** A universal *algebra*[26] is an *abstract algebra*[2] where we leave the postulates (axioms, laws) unspecified. (The universal level of abstract, the viewpoint of universal algebras, represents for us [158], the high water mark of abstraction in the treatment of *algebraic systems*[28].)

791. **Underspecify:** By an underspecified expression, typically an identifier, we mean one which for repeated occurrences in a specification text always yields the same value, but what the specific value is, is not knowable. (Cf. *nondeterministic*[481] or *loose specification*[435].)

792. **Undecidable:** A formal logic system is undecidable if there is no *algorithm*[31] which prescribes *computation*[144]s that can determine whether any given sentence in the system is a theorem.

793. **Universe of discourse:** That which is being talked about; that which is being discussed; that which is the subject of our concern. (The four most prevalent universes of discourse of this book, this series of volumes on software engineering, are: *software development*[690] *methodology*[457], *domains*, *requirements*[605] and *software design*[688].)

794. **Update:** By an update we shall understand a change of value of a variable, including also the parts, or all, of a *database*[195].

795. **Update problem:** By the update problem we shall understand that data stored in a *database*[195] usually reflect some state of a domain, but that changes in the external state of that domain are not always properly, including timely, reflected in the database.

796. **User:** By a user we shall understand a person who uses a *computing system*[151], or a *machine*[436] (i.e., another computing system) which *interfaces* with the former. (Not to be confused with *client*[116] or *stakeholder*[703].)

797. **User-friendly:** A "lofty" term that is often used in the following context: *"A computing system, a machine, a software package, is required to be user-friendly"* — without the requestor further prescribing the meaning of that term. Our definition of the term user-friendly is as follows: A *machine*[436] (software + hardware) is said to be user-friendly (i) if the *shared phenomena* of the application *domain*[239] (and *machine*[436]) are each implemented in a transparent, one-to-one manner, and such that no IT jargon, but common application *domain*[239] *terminology*[752] is used in their (i.1) accessing, (i.2) *invocation*[402] (by a human *user*[796]), and (i.3) display (by the machine); i.e., (ii) if the *interface requirements*[394] have all been carefully expressed (commensurate, in further detailed ways: ..., with the user psyche) and correctly implemented; and (iii) if the machine otherwise satisfies a number of *performance* and *dependability requirements*[605] that are commensurate, in further detailed ways: ..., with the user psyche.

798. **User manual:** A *document*[237] which a regular user of a *computing system*[151] refers to when in doubt concerning the use of some features of that system. (See also *installation manual*[386] and *training manual*[767].)

. . . . $\mathcal{V}$

799. **Valid:** A *predicate*[536] is said to be *valid* if it is true for all *interpretation*[397]s. (In this context think of an interpretation as a *binding*[88] of all *free*[305] *variable*[803]s of the predicate expression to *value*[802]s; cf. *satisfiable*[644].)

800. **Validation:** (Let, in the following *universe of discourse*[793] stand consistently for either *domain*[239], *requirements*[605] or *software design*[688].) By universe of discourse validation we understand the assurance, with universe of discourse *stakeholders*, that the specifications produced as a result of universe of discourse acquisition, universe of discourse analysis and *concept formation*[153], and universe of discourse domain *modelling* are commensurate with how the stakeholder views the universe of discourse. (*Domain* and *requirements validation*[619] is treated in Vol. 3, Chaps. 14 and 22.)

801. **Valuation:** Same as *evaluation*[280].

802. **Value:** From (assumed) Vulgar Latin *valuta*, from feminine of *valutus*, past participle of Latin *valere* to be of worth, be strong [213]. (Commensurate with that definition, value, to us, in the context of programming (i.e., of software engineering), is whatever mathematically founded *abstraction*[3] can be captured by our *type*[782] and *axiom*[75] *systems*. (Hence numbers, truth values, *tokens*, sets, Cartesians, lists, maps, functions, etc., of, or over, these.))

803. **Variable:** (i) From Latin *variabilis*, from *variare* to vary; (ii) able or apt to vary; (iii) subject to variation or changes [213]. (Commensurate with that definition, a variable, to us, in the context of programming (i.e., of software engineering), is a *placeholder*, for example, a *storage*[715] *location*[431] whose *contents* may change. A variable, further, to us, has a name, the variable's identifier, by which it can be referred.)

804. **VDM:** `VDM` stands for the `Vienna Development Method` [52, 53]. (`VDM-SL` (SL for Specification Language) was the first formal specification language to have an international standard: `VDM-SL, ISO/IEC 13817-1: 1996`. The author of this book

coined the name VDM in 1974 while working with Hans Bekič, Cliff B. Jones, Wolfgang Henhapl and Peter Lucas, on what became the VDM description of PL/I. The IBM Vienna Laboratory, in Austria, had, in the 1960s, researched and developed semantics descriptions [17, 18, 19, 162] of PL/I, a programming language of that time. "JAN" (John A.N.) Lee [153] is believed to have coined the name VDL [154, 161] for the notation (the Vienna Definition Language) used in those semantics definitions. So the letter M follows, lexicographically, the letter L, hence VDM.)

805. **VDM–SL:** VDM-SL stands for the VDM Specification Language. (See entry VDM above. Between 1974 and the late 1980s VDM-SL was referred to by the acronym Meta-IV: the fourth metalanguage (for language definition) conceived at the IBM Vienna Laboratory during the 1960s and 1970s.)

806. **Verb:** A *word*[814] that characteristically is the grammatical centre of a sentence and expresses an act, occurrence or mode of being that in various languages is inflected for agreement with the subject, for tense, for voice, for mood, or for aspect, and that typically has rather full descriptive meaning and characterizing quality but is sometimes nearly devoid of these especially when used as an auxiliary or linking verb [213]. (We shall often find, in modelling, that we model verbs as *functions* (incl. *predicates*).)

807. **Verification:** By verification we mean the process of determining whether or not a specification (a description, a prescription) fulfills a stated property. (That stated property could (i) either be a property of the specification itself, or (ii) that the specification relates, somehow, i.e., is correct with respect to some other specification.)

808. **Verify:** Same, for all practical purposes, as *verification*[807].

809. **Vertex:** Same as an *node*[479].

. . . . $\mathcal{W}$

810. **Waterfall diagram:** By a waterfall diagram is understood a two-dimensional diagram with a number of boxes placed, say, on a diagonal, from a top left corner of the diagram to a lower right corner, such that the individual boxes are sufficiently spaced apart, i.e., do not overlap, and such that arrows (i.e., "the water") infix adjacent boxes along a perceived diagonal line. (The idea is then that a preceding box, from which an arrow emanates, designates a software development activity that must, somehow, be concluded before activity can start on the software development activity designated by the box upon which the infix arrow is incident.)

811. **Weakest pre-condition:** The condition that characterizes the set of all initial states, such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition, is called "the weakest pre-condition corresponding to that post-condition". (We call it "weakest", because the weaker a condition, the more states satisfy it and we aim here at characterising all possible starting states that are certain to lead to a desired final state.)

812. **Well-formedness:** By well-formedness we mean a concept related to the way in which *information*[373] or *data structure*[199] definitions may be given. Usually these are given in terms of *type definition*[785]s. And sometimes it is not possible, due to the *context-free*[173] nature of type definitions. (Well-formedness is here seen separate from the *invariant*[400] over an *information*[373] or a *data structure*[199]. We refer to the explication of *invariant*[400]!)

813. **Wildcard:** A special symbol that stands for one or more characters. (Many operating systems and applications support wildcards for identifying files and directories. This enables you to select multiple files with a single specification. Typical wildcard designators are * (asterisk) and _ (underscore).)

814. **Word:** A speech sound or series of speech sounds or a character or series of juxtaposed characters that symbolizes and communicates a meaning without being divisible into smaller units capable of independent use [213].