From Domains to Requirements On a Triptych of Software Development 1

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

31 December 2009: Compiled: April 22, 2010: 16:08 ECT

Abstract 2

We shall present core aspects of the Triptych approach to software engineering. The benefits from deploying this approach are that we both achieve *the right software* and *software that is right* [27, Boehm 1981]. *The right software* is software that meets all of the customers' expectations and only those.

Software that is right is software that is correct with respect to specific requirements prescriptions. 3 Experience has shown that using also the formal techniques part of the Triptych approach has lead to projects that are on time and at initially estimated costs [83]. To achieve **the right software** we "prefix" the phase of requirements engineering with a phase of domain engineering – and these lecture notes will present core aspects of domain engineering. To achieve **software that is right** we do two things: (i) "derive" requirements prescriptions from domain descriptions and software design from requirements prescriptions – and this these lecture notes will present core aspects of a somewhat different approach to requirements engineering, and (ii) formulate descriptions and prescriptions both informally, in precise, say English narratives, and formally. The latter is not shown in these lecture notes. 4 The "somewhat" different approach to requirements engineering, however, and as we shall see, fits reasonably "smoothly" with current requirements engineering approaches, [62, Axel van Lamsweerde]. Precursors of the 'triptych' approach was used in DDC's 44 man-year Ada Compiler development project [25]. That project was on time and at cost, and time and cost were significantly below those of other commercial Ada compiler developments [32]. 5 The 'triptych' approach has been in partial use since the early 1990s, including at the United Nations University's International Institute for Software Technology (www.iist.unu.edu). Young software engineers, while being tutored by UNU-IIST's science & engineering staff, domain engineered, requirements engineered and software designed (incl. implemented)[2] trustworthy

[1] Margin numerals refer to slides. Thus margin numeral i refer to Slide $\#i$. A total of 440 slides support the lecture presentation of these 152 pages of lecture notes.

software systems that have met customer expectations – with what seems be sub-
stantially fewer man-power resources than usually experienced and within planned
6 time limits.

Domain engineering, in the sense of these lecture notes, is offered as a means to
help secure that software engineers deliver *the right software* – where formalisation
of relevant stages and steps of software development helps secure that *the software*
is right [27]. In these lecture notes we shall present the essence of a software devel-
7 opment *triptych*: from domains via requirements to software design. We emphasize
the two first phases: *domain engineering* and *requirements engineering*. We show
the pragmatic stages of the construction of *domain descriptions*: the facets of *in-*
trinsics, support technologies, rules & regulations, script (licenses and contracts),
8 *management & organisation,* and *human behaviour.* And we show how to con-
struct main facets of *requirements prescriptions*: *domain requirements* and *interface*
requirements. In this respect we focus in particular on the *domain requirements*
development stages of *projection, instantiation, determination* and *extension*. The
lecture notes represents a summary as well as some significant improvements over
the domain-to-requirements coverage of [12, Vol. 3, Parts IV–V].

Actual Lecture Notes versus Reference Material

The barebones lecture notes consists of Sects. 1–6 (i.e., Pages 17–48). They cover five double lectures, say one week ! In that week the reader will learn the basic facts of domain and requirements engineering.

Appendix Sect. A (i.e., Pages 57–113) – five more double lectures – cover the formal specification language used in these lecture notes. In such a week the reader will additionally learn the formal specification language `RSL` [49]. With that knowledge the reader can rather easily adapt to such other formal specification languages as `Alloy` [59], `Event B` [1], `VDM-SL`[46] and `Z` [84].

Appendix Sects. B–C (i.e., Pages 115–152) explore the (related) concepts of *entities* and *mereology*. They can be used for lectures or for self-study.

Appendix Sects. D–U are more meant more as reference material (i.e., for self-study) on more detailed aspects of domain and requirements engineering facets. Appendix G (Pages 163–178) is rather research topic oriented and is meant for PhD student self-study.

Appendix Sects. V–X (i.e., Pages 262–337) provide medium-scale examples of informal and formal domain descriptions of aspects of The Tokyo Stock Exchange, Logistics and Pipelines (fpr gas or oil).

One set of Lecture Notes: Two possible Courses

These lecture notes can be used either for a five day course or for a 15 day course. A five day course would simply cover Chaps. 1–6 (Pages 17–48). For example in two lectures per day, approximately 50 minutes each, and with no course project. A fiften day course would cover Chaps. 1–6 and Appendices A–C (Pages 17–48 + 57–152). For example in two lectures per day, approximately 50 minutes each, and with a course project as described in Appendix Y (Pages 340–346).

Lecture Notes for T U Wien, April 2010 9

The present version of this document is intended as the “written” support for my April 2010
lectures at the Technical University of Vienna. Austria. The **www.imm.dtu.dk/˜db/wien**
web page gives details. From there you can see that Sects. 1–5 covers 5 lectures and that
Appendix A covers 8 lectures. To examples of sections 2–4 we have “added” formalisations. 10
These formalisations are in the RAISE specification languages RSL. And I have additionally
added an extensive appendix, An RSL Primer[2], so that readers can also learn RSL, the
specification language for a **r**igorous **a**pproach to **i**ndustrial **s**oftware **e**ngineering, RAISE.
The primer contains many examples which expands on the examples of sections 2–4.

“Formalisation–Parametrised” Examples and Primer 11

The formalisations of the examples of sections 2–4 could as well be expressed in one of the other prominent formal specifications languages current at this time (April 22, 2010), for example: Alloy [59], Event B [1], VDM-SL [46] or Z [84]. It could be interesting if this little book could entice my Alloy, Event B, VDM-SL and Z colleagues to “rewrite/reformulate” the formal parts of all examples into their main tool of formal expression (besides mathematics). I would be very willing to engage in such a project having the aim of making my and their notes Internet-based and thus publically available.

On Studying the Examples 12

In order to learn to **write** poems one must **read** poetry. In order to learn th **write** formal specifications one must **read** formal specifications We have ourselves found that even if students attend pedagogically and didactically exciting and sound lectures they must still, in the quiet of their study room, without listening to Ipod (or the like), carefully study the examples we are presenting. And we are presenting many examples, 49 in all !To begin with little explanation is given of the formulas. Instead we rely on the reader’s ability to relate the numbered formulas to the numbered annotation textst. As from Appendix A we present a schematic syntax and informal semantics of the spexification language, RSL, used in these notes. Readers are well adviced in studying all examples.

[2] a small introductory book on a subject

13 On Course Lectures Based on these Notes

On a Course Project Based on these Lecture Notes

We refer to Appendix Y (Pages 340–346).

Contents

Part I
5 Lectures Days

1 Introduction 14

1.1 Some Observations

Current software development, when it is pursued in a state-of-the-art, but still a conventional manner, starts with requirements engineering and proceeds to software design. Current software development practices appears to be focused on processes (viz.: "best practices': tools and techniques'). In a delightful paper: [58, CACM, April 2009] Daniel Jackson introduces the concept of *'a direct path to* [the development of] *dependable software'* – in contrast to the concept of processes. The current paper contributes to such *direct paths.* 15

An aeronautics engineer to be hired by Airbus 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 well versed in applied mathematics and in the physics of electromagnetic wave propagation in matter. And so forth. 16

Software engineers hired for the development of software for hospitals, or for railways, know little, if anything, about health care, respectively rail transportation (scheduling, rostering, signalling, 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. 17

It is therefore quite reasonable to expect the domain-specific software engineer to understand proper, including formal descriptions of their domains: for railways cf. www.railway-domain.org, and for pipelines pipelines.pdf, logistics logistics.pdf and for container lines container-paper.pdf – all at www.imm.dtu.dk/~db/. 18

The process knowledge and "best" practices of the triptych software engineering is well-founded and takes place in the context of established domain model and an established, carefully phrased (and formalised) requirements model. The 24 hour 7 days a week trustworthy operation of many software systems is so crucial that utmost care must be taken to ensure that they fulfill all (and only) the customers expectations and are correct. Barry 19
Boehm [27, 1981] has coined the statement: *it is the right software and the software is right*. Extra care must be taken to ensure those two "rights". And here it is not enough to only follow current "best process, technique and tool practices". Software engineers must follow – what is also clearly stated in [58, Daniel Jackson] – some form of direct path. This paper will illustrate some facets of such a direct approach. The phase, stage and step-wise, possibly iterative composition of what the Triptych approach offers has been so arranged as to provide *direct evidence* of the evolving software's dependability.

1.2 A Triptych of Software Engineering 20

> **Dogma:** *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.*

21 In consequence we suggest that software, "ideally"[3], 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 relative completeness. $\mathcal{P}$roperties, as
stated by domain stakeholders, are proved with respect to the domain description ($\mathcal{D}\models\mathcal{P}$).
This phase is the most important, we think, when it comes to secure the first of the two
22 "rights": that we are on our way to develop the right software.

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* points out which domain
description parts are to be kept (*projected*) out of the *domain requirements*, and for those
23 kept in, what *instantiations*, *determinations* and *extensions* are required. Similarly the
requirements stakeholders, guided by the domain and requirements engineers, informs as
to which domain *entities: simple*, *actions*, *events* and *behaviours* are *shared* between the
domain and the *machine*, that is, the *hardware* and the *software* being required. In these
24 lecture 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 lecture
25 notes – other than saying this: the design is "derived" from the requirements model.

To ensure that the software being developed is right, that is, correct, we can then rigorously argue, informally, or formally – test, model check and/or prove, that the $\mathcal{S}$oftware is correct with respect to the $\mathcal{R}$equirements in the context of the $\mathcal{D}$omain: $\mathcal{D},\mathcal{S}\models\mathcal{R}$. These, the $\mathcal{D}$omain descriptions, the $\mathcal{R}$equirements prescriptions, the $\mathcal{S}$oftware design specification, and the rigorous correctness arguments (whether informal or formal) are examples of [58]'s concept of *direct evidence*.

1.3 What are Domains ? 26

By a domain we shall here understand a universe of discourse, an area of nature subject
to laws of physics and study by physicists, or an area of human activity subject to its
interfaces with other domains and to nature. There are other domains – which we shall
27 ignore. We shall focus on the human-made domains. "Large scale" examples arethe *financial service industry: banking, insurance, securities trading, portfolio management, etc.*;
health care: hospitals, clinics, patients, medical staff, etc.; *transportation: road, rail/train,
sea/shipping*, and *air/aircraft transport (vehicles, transport nets, etc.)*; *oil and gas sys-
28 tems: pumps, pipes, valves, refineries, distribution, etc.* "Intermediate scale" examples
are*automobiles: manufacturing* or *monitoring and control, etc.*; *heating systems*; *heart
pumps*; etc. 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.

[3] Section 5.7 [Item 5] will discuss renditions of "idealism"!

1.4 What is a Domain Description ? 29

By a *domain description* we understand a description of the *simple 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. A description is *syntax*. The meaning 30
(*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 text alternates with clearly related formula text.

1.5 Description Languages 31

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, [49], in our formal texts. But any of the model-oriented approaches and languages offered by Alloy [59], Event B [1], VDM [46] and Z [84], should work as well. 32

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 [73], CSP: Communicating Sequential Processes [56], MSC: Message Sequence Charts [57], Statecharts [54], and some temporal logic, for example either DC: Duration Calculus [85] or TLA+ [61]. Research into how such diverse textual and diagrammatic languages can be meaningfully and proof-theoretically combined is ongoing [3].

1.6 Contributions of these Lecture Notes 33

We claim that the major contributions of the Triptych approach to software engineering as presented in this paper 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 *intrinsics, support technologies, management and organisation, rules and regulations, scripts (licenses and contracts)* and *human behaviour* whereby 'elaboration' we mean that we provide principles and techniques for the construction of these facet description parts (Sects. 3.2–3.7); (3) the *re-identification* and *'elaboration'* of the concept of *business pro-* 34
cess re-engineering (Sect. 4.1) on the basis of the notion of *business processes* as first introduced in Sect. 3.1; (4) the *identification* and *'elaboration'* of the technically determined *domain requirements facets* of *projection, instantiation, determination, extension* and *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 35
the technically determined *interface requirements facets* of *shared entity, shared action,*

shared event and *shared behaviour* requirements principles and techniques (Sects. 4.3.1–4.3.4). We claim that the facets of (2, 3, 4) and (5) are all *novel*. In Sect. 5 we shall discuss these contributions in relation to the works and contributions of other researchers and technologists.

1.7 Structure of Lecture Notes 36

Before going into some details on domain enginering (Sect. 3) and requirements engineering (Sect. 4) we shall in the next section (Sect. 2) cover the basic concepts of specifications, whether domain descriptions or requirements prescriptions. These are: entities, actions, events and behaviours. Section 5 then discuses the contributions of the Triptych approach as covered in this paper.

2 A Specification Ontology 37

In order to describe domains we postulate the following related specification components: *entities*, *actions*, *events* and *behaviours*. Although not part of a proper domain description the examples of this section are necessary in order for the reader to better envisage what the domain descriptions and requirements prescriptions must "ultimately" cover.

2.1 Entities 38

By an entity we shall understand a phenomenon we can point to in the domain or a concept formed from such phenomena. 39

Example 1 – Entities
The example is that of aspects of a transportation net. You may think of such a net as being either a road net, a rail net, a shipping net or an air traffic net. Hubs are then street intersections, train stations, harbours, respectively airports. Links are then street segments between immediately adjacent intersections, rail tracks between train stations, sea lanes between harbours, respectively air lanes between airports. 40

1 There are hubs and links.

2 There are nets, and a net consists of a set of two or more hubs and one or more links.

3 There are hub and link identifiers.

4 Each hub (and each link) has an own, unique hub (respectively link) identifier (which can be observed (ω) from the hub [respectively link]).

41

```
type
   [1]  H, L,
   [2]  N = H-set × L-set
axiom [nets–hubs–links–1]
   [2]  ∀ (hs,ls):N • card hs≥2 ∧ card ls≥1
type
   [3]  HI, LI
value
   [4]  ωHI: H → HI, ωLI: L → LI
axiom [nets–hubs–links–2]
   [4]  ∀ h,h′:H, l,l′:L • h≠h′ ⇒ ωHI(h)≠ωHI(h′) ∧  l≠l′⇒ωLI(l)≠ωLI(l′)
```

42

In order to model the physical (i.e., domain) fact that links are delimited by two hubs and that one or more links emanate from and are, at the same time, incident upon a hub we express the following: 43

5 From any link of a net one can observe the two hubs to which the link is connected. We take this ‘observing’ to mean the following: from any link of a net one can observe the two distinct identifiers of these hubs.

6 From any hub of a net one can observe the identifiers of one or more links which are connected to the hub.

7 Extending Item [5]: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.

8 Extending Item [6]: the observed link identifiers must be identifiers of links of the net to which the hub belongs.

44

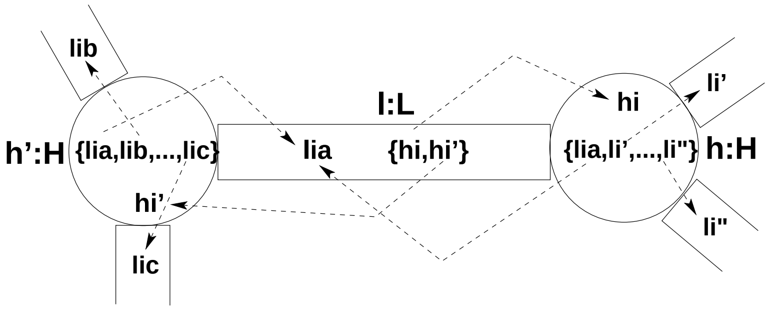


Figure 1: Connected links and hubs with observable identifiers

45

value

$[\,5\,]\ \ \omega\mathsf{HIs}\colon\ \mathsf{L} \rightarrow \mathsf{HI}\text{-}\mathbf{set},$

$[\,6\,]\ \ \omega\mathsf{LIs}\colon\ \mathsf{H} \rightarrow \mathsf{LI}\text{-}\mathbf{set},$

axiom $[\,\mathsf{net{-}hub{-}link{-}identifiers{-}1}\,]$

$[\,5\,]\ \ \forall\ \mathsf{l{:}L} \bullet \mathbf{card}\ \omega\mathsf{HIs(l)}{=}2\ \wedge$

$[\,6\,]\ \ \forall\ \mathsf{h{:}H} \bullet \mathbf{card}\ \omega\mathsf{LIs(h)}{\geq}1\ \wedge$

$\forall\ \mathsf{(hs,ls){:}N} \bullet$

$[\,5\,]\ \ \quad \forall\ \mathsf{h{:}H} \bullet \mathsf{h} \in \mathsf{hs} \Rightarrow \forall\ \mathsf{li{:}LI} \bullet \mathsf{li} \in \omega\mathsf{LIs(h)}$

$\qquad\qquad\qquad \Rightarrow \exists\ \mathsf{l'{:}L} \bullet \mathsf{l'} \in \mathsf{ls} \wedge \mathsf{li}{=}\omega\mathsf{LI(l')} \wedge \omega\mathsf{HI(h)} \in \omega\mathsf{HIs(l')} \wedge$

$[\,6\,]\ \ \quad \forall\ \mathsf{l{:}L} \bullet \mathsf{l} \in \mathsf{ls} \Rightarrow \exists\ \mathsf{h',h''{:}H} \bullet \{\mathsf{h',h''}\}{\subseteq}\mathsf{hs} \wedge \omega\mathsf{HIs(l)}{=}\{\omega\mathsf{HI(h')},\omega\mathsf{HI(h'')}\}$

$[\,7\,]\ \ \forall\ \mathsf{h{:}H} \bullet \mathsf{h} \in \mathsf{hs} \Rightarrow \omega\mathsf{LIs(h)} \subseteq \mathsf{iols(ls)}$

$[\,8\,]\ \ \forall\ \mathsf{l{:}L} \bullet \mathsf{l} \in \mathsf{ls} \Rightarrow \omega\mathsf{HIs(h)} \subseteq \mathsf{iohs(hs)}$

value

$\mathsf{iohs}\colon\ \mathsf{H}\text{-}\mathbf{set} \rightarrow \mathsf{HI}\text{-}\mathbf{set},\ \mathsf{iols}\colon\ \mathsf{L}\text{-}\mathbf{set} \rightarrow \mathsf{LI}\text{-}\mathbf{set}$

$\mathsf{iohs(hs)} \equiv \{\omega\mathsf{HI(h)}|\mathsf{h{:}H}{\bullet}\mathsf{h} \in \mathsf{hs}\}$

$\mathsf{iols(ls)} \equiv \{\omega\mathsf{LI(l)}|\mathsf{l{:}L}{\bullet}\mathsf{l} \in \mathsf{ls}\}$

Property Assertion: No Isolated Hubs.

- ..
- ..

assertion:

…
…

End of Property Assertion

Property Assertion: No Dangling Links.

- ..
- ..

assertion:

…
…

End of Property Assertion 46

In the above extensive example we have focused on just five entities: nets, hubs, links and their identifiers. The nets, hubs and links can be seen as separable phenomena. The hub and link identifiers are conceptual models of the fact that hubs and links are connected — so the identifiers are abstract models of 'connection', i.e., the mereology of nets, that is, of how nets are composed. These identifiers are attributes of entities.

Links and hubs have been modelled to possess link and hub identifiers. A link's "own" link identifier enables us to refer to the link, A link's two hub identifiers enables us to refer to the connected hubs. Similarly for the hub and link identifiers of hubs and links. 47

9 A hub, h_i, state, $h\sigma$, is a set of hub traversals.

10 A hub traversal is a triple of link, hub and link identifiers $(l_{i_{in}}, h_{i_i}, l_{i_{out}})$ such that $l_{i_{in}}$ and $l_{i_{out}}$ can be observed from hub h_i and such that h_{i_i} is the identifier of hub h_i.

11 A hub state space is a set of hub states such that all hub states concern the same hub.

48

```
type
   [9]  HT = (LI×HI×LI)
   [10] HΣ = HT-set
   [11] HΩ = HΣ-set
value
   [10] ωHΣ: H → HΣ
   [11] ωHΩ: H → HΩ
axiom [hub−states]
```

```
    ∀ n:N,h:H•h ∈ ωHs(n)⇒wf_HΣ(ωHΣ(h))∧wf_HΩ(h,ωHΩ(h))
value
    wf_HΣ: HΣ → Bool, wf_HΩ: H×HΩ → Bool
    wf_HΣ(hσ) ≡ ∀ (li,hi,li′),(_,hi′,_):HT•(li,hi,li′)∈ hσ ⇒ {li,li′}⊆ωLIs(h)∧hi=ωHI(h)∧hi′=hi
    wf_HΩ(h,hω) ≡ ∀ hσ:HΣ•hσ ∈ hω⇒wf_HΣ(hσ)∧hσ≠{} ⇒
                    let (li,hi,li′):HT•(li,hi,li′)∈ hσ in hi=ωHI(h) end
```

Property Assertion: Hub States and Hub State Spaces.

- ..
- ..

assertion:

```
    ...
    ...
```

End of Property Assertion

■ End of Example 1

We refer to Appendix Sect. B (Pages 115–125) and Appendix Sect. C.2 (Pages 134–134) for a more substantial coverage of the concept of entities.

2.2 Actions 49

A set of entities form a domain state. It is the domain engineer which decides on such states. A function is an action if, when applied to zero, one or more arguments and a state, it then results in a state change. (Arguments could be other entities or just values of entity attributes.) 50

Example 2 – Deterministic Hub State Setting

12 Our example action is that of setting the state of hub.

13 The setting applies to a hub

14 and a hub state in the hub state space

13 and yields a “new” hub.

15A The before and after hub identifier remains the same.

15B The before and after link identifiers remain the same.

16 The before and after hub state space remains the same.

17 The result hub state is that being set (i.e., the argument hub state).

51

```
value
  [12]  set_HΣ: H × HΣ → H
  [13]  set_HΣ(h,hσ) as h′
  [14]    pre hσ ∈ ωHΩ(h)
  [15A]   post ωHI(h)=ωHI(h′)∧
  [15B]        ωLIs(h)=ωLIs(h′)∧
  [16]         ωHΩ(h)=ωHΩ(h′)∧
  [17]         ωHΣ(h′)=hσ
```

Property Assertion: Deterministic Hub States.

- ..
- ..

assertion:

...
...

End of Property Assertion

■ End of Example 2

52 Example 2 illustrated a deterministic action: one that always succeeded in carrying out the prescribed operation. But, as we shall see in Sect. 3.3, the domain technology may be faulty and an action, as carried out by such a technology, may fail to have the desired effect. 53

Example 3 – Non-Deterministic Hub State Setting

17 The result hub state is one of the hub states of the hub state space.

```
value
  [12]  set_HΣ: H × HΣ → H
  [13]  set_HΣ(h,hσ) as h′
  [14]    pre hσ ∈ ωHΩ(h)
  [15A]   post ωHI(h)=ωHI(h′)∧
  [15B]        ωLIs(h)=ωLIs(h′)∧
  [16]         ωHΩ(h)=ωHΩ(h′)∧
  [17]         ωHΣ(h′) ∈ ωHΩ(h)
```

But Example 3 is still only an approximation. Appendix Sect. B.3.7 shall present the proper way to compare entities.

We refer to Appendix Sect. D (Pages 154–156) for a more substantial coverage of the concept of actions.

2.3 Events 54

Any domain state change is an event. A situation in which a (specific) state change was expected but none (or another) occurred is an event. Some events are more "interesting" than other events. Not all state changes are caused by actions of the domain. 55

Example 4 – Events: Failure State Transitions

18 A hub is in some state, $h\sigma$.

19 An action directs it to change to state $h\sigma'$ where $h\sigma' \neq h\sigma$.

20 But after that action the hub remains either in state $h\sigma$ or is possibly in a third state, $h\sigma''$ where h$\sigma'' \not\in$ {hσ,hσ'}.

21 Thus an "interesting event" has occurred !

56

```
∃ n:N,h:H,hσ,hσ′:HΣ•h ∈ ωHs(n)∧
  [19,20]  {hσ,hσ′}⊆ωHΩ(h)∧card{hσ,hσ′}=2 ∧
  [18]     ωHΣ(h)=hσ ;
  [19]     let h′ = set_HΣ(h,hσ′) in
  [20]     ωHΣ(h′)∈ ωHΣ(h′)\{hσ′} ⇒
  [21]     "interesting event" end
```

It only makes sense to change hub states if there are more than just one single such state. ■ End of Example 4

We refer to Appendix Sect. E (Pages 157–159) for a more substantial coverage of the concept of events.

2.4 Behaviours 57

A behaviour is a set of zero, one or more sequences of sets of actions or behaviours, including events. 58

Example 5 – Behaviours: Blinking Semaphores

22 Let h be a hub of a net n.

23 Let $h\sigma$ and $h\sigma'$ be two distinct states of h.

24 Let $ti : TI$ be some time interval.

25 Let h start in an initial state $h\sigma$.

26 Now let hub h undergo an ongoing sequence of n changes

26a from $h\sigma$ to $h\sigma'$ and

26b then, after a wait of ti seconds,

26c and then back to $h\sigma$.

26d After n blinks a pause, $tp : TI$, is made and blinking restarts.

59

```
type
   TI
value
   ti,tj:TI [ axiom tj>>ti ]
   n:Nat,
   [ 26 ]  blinking: H × HΣ × HΣ → Unit
   [ 26 ]  blinking(h,hσ,hσ′,m) in
   [ 25 ]     let h′ = set_HΣ(h,hσ) in
   [ 26c ]    wait ti ;
   [ 26a ]    let h″ = set_HΣ(h′,hσ′) in
   [ 26c ]    wait ti ;
   [ 26 ]     if m=1
   [ 26 ]         then skip
   [ 26 ]         else blinking(h,hσ,hσ′,m−1) end end end
   [ 26 ]      wait tj ;
   [ 26d ]     blinking(h,hσ,hσ′,n)
   [ 23 ]     pre {hσ,hσ′}⊆ωHΩ(h)∧hσ≠hσ′
   [ 26 ]          ∧ initial m=n
```

■ End of Example 5

We refer to Appendix Sect. F (Pages 160–162) for a more substantial coverage of the concept of behaviours.

3 Domain Engineering 60

We focus on the *facet* components of a domain description and leave it to other publications, for ex. [12, Vol. 3, Part IV, Chaps. 8–10] to cover such aspects of domain engineering as stakeholder identification and liaison, domain acquisition and analysis, terminologisation, verification, testing, model-checking, validation and domain theory formation. 61

By understanding, first, the *facet* components the domain engineer is in a better position to effectively establish the regime of stakeholders, pursue acquisition and analysis, and construct a necessary and sufficient terminology. The domain description components each cover their domain facet. 62

We outline six such facets: intrinsics, support technology, rules and regulations, scripts (licenses and contracts), management and organisation, and human behaviour. But first we cover a notion of business processes.

3.1 Business Processes 63

By a business process we understand a set of one or more, possibly interacting behaviours which fulfill a business objective. We advocate that domain engineers, typically together with domain stakeholder groups, rough-sketch their individual business processes. 64

Example 6 – Some Transport Net Business Processes With respect to one and the same underlying road net we suggest some business-processes and invite the reader to rough-sketch these.

27 **Private citizen automobile transports:** Private citizens use the road net for pleasure and for business, for sightseeing and to get to and from work.

A private citizen automobile transport "business process rough-sketch" might be:

> A car owner drives to work: *Drives out, onto the street, turns left, goes down the street, straight through the next three intersections, then turns left, two blocks straight, etcetera, finally arrives at destination, and finally turns into a garage.*

65

28 **Public bus (&c.) transport:** Province and city councils contract bus (&c.) companies to provide regular passenger transports according to timetables and at cost or free of cost.

A public bus transport "business process rough-sketch" might be:

> A bus drive from station of origin to station of final destination: *Bus driver starts from station of origin at the designated time for this drive; drives to first passenger stop; open doors to let passenger in; leaves stop at time table designated time; drives to next stop adjusting speed to traffic conditions and to "keep time" as per the time table; repeats this process: "from stop*

to stop", letting passengers off and on the bus; after having (thus, i.e., in this manner) completed last stop "turns" bus around to commence a return drive. 66

29 **Road maintenance and repair:** Province and city councils hire contractors to monitor road (link and hub) surface quality, to maintain set standards of surface quality, and to "emergency" re-establish sudden occurrences of low quality.

30 **Toll road traffic:** State and province governments hire contractors to run toll road nets with toll booth plazas.

31 **Net revision: road (&c.) building:** State government and province and city councils contract road building contractors to extend (or shrink) road nets.

The detailed description of the above rough-sketched business process synopses now becomes part of the domain description as partially exemplified in the previous and the next many examples. ■ End of Example 6

67 Rough-sketching such business processes helps bootstrap the process of domain acquisition. We shall return to the notion of business processes in Sect. 4.1 where we introduce the concept of *business process re-engineering*.

We refer to Appendix Sect. H (Pages 179–182) for a more substantial coverage of the concept of business processes.

3.2 Intrinsics

By intrinsics we shall understand the very basics, that without which none of the other facets can be described, i.e., that which is common to two or more, usually all of these other facets. 68

Example 7 – Intrinsics Most of the descriptions of Sect. 2 model intrinsics. We add a little more:

32 A link traversal is a triple of a (from) hub identifier, an along link identifier, and a (towards) hub identifier

33 such that these identifiers make sense in any given net.

34 A link state is a set of link traversals.

35 And a link state space is a set of link states.

69

```
value
   n:N
type
   [32]  LT′ = HI × LI × HI
   [33]  LT = {|lt:LT′•wfLT(lt)(n)|}
   [34]  LΣ′ = LT-set
   [34]  LΣ = {|lσ:LΣ′•wf_LΣ(lσ)(n)|}
   [35]  LΩ′ = LΣ-set
   [35]  LΩ = {|lω:LΩ′•wf_LΩ(lω)(n)|}
value
   [33]  wfLT: LT → N → Bool
   [33]  wfLT(hi,li,hi′)(n) ≡
   [33]    ∃ h,h′:H•{h,h′}⊆ωHs(n)∧
   [33]      ωHI(h)=hi∧ωHI(h′)=hi′∧
   [33]      li ∈ ωLIs(h)∧li ∈ ωLIs(h′)
```

The wf_LΣ and wf_LΩ can be defined like the corresponding functions for hub states and hub state spaces. ■ End of Example 7

We refer to Appendix Sect. I (Pages 183–185) for a more substantial coverage of the concept of intrinsics.

3.3 Support Technologies 70

By support technologies we shall understand the ways and means by which humans and/or technologies support the representation of entities and the carrying out of actions. 71

Example 8 – Support Technologies
Some road intersections (i.e., hubs) are controlled by semaphores alternately shining **red**–**yellow**–**green** in carefully interleaved sequences in each of the in-directions from links incident upon the hubs. Usually these signalings are initiated as a result of road traffic sensors placed below the surface of these links. We shall model just the signaling: 72

36 There are three colours: **red**, **yellow** and **green**.

37 Each hub traversal is extended with a colour and so is the hub state.

38 There is a notion of time interval.

39 Signaling is now a sequence, $\langle(h\sigma',t\delta'),\ (h\sigma'',\ t\delta''),\ \ldots,\ (h\sigma'^{\cdots'},\ t\delta'^{\cdots'})\rangle$ such that the first hub state $h\sigma'$ is to be set first and followed by a time delay $t\delta'$ whereupon the next state is set, etc.

40 A semaphore is now abstracted by the signalings that are prescribed for any change from a hub state $h\sigma$ to a hub state $h\sigma'$.

73

```
type
  [36] Colour == red | yellow | green
  [37] X = LI×HI×LI×Colour [crossings of a hub]
  [37] $H\Sigma$ = X-set [hub states]
  [38] TI [time interval]
  [39] Signalling = $(H\Sigma \times TI)^*$
  [40] Semaphore = $(H\Sigma \times H\Sigma) \ \overrightarrow{m}$ Signalling
value
  [37] $\omega H\Sigma$: H → $H\Sigma$
  [40] $\omega$Semaphore: H → Sema,
  [41] chg_$H\Sigma$: H × $H\Sigma$ → H
  [41] chg_$H\Sigma$(h,h$\sigma$) as h′
  [41]   pre h$\sigma$ ∈ $\omega H\Omega$(h) post $\omega H\Sigma$(h′)=h$\sigma$
```

74

```
  [39] chg_$H\Sigma$_Seq: H × $H\Sigma$ → H
  [39] chg_$H\Sigma$_Seq(h,h$\sigma$) ≡
  [39]   let sigseq = ($\omega$Semaphore(h))($\omega\Sigma$(h),h$\sigma$) in
  [39]   sig_seq(h)(sigseq) end
  [39] sig_seq: H → Signalling → H
  [39] sig_seq(h)(sigseq) ≡
  [39]   if sigseq=⟨⟩ then h else
  [39]   let (h$\sigma$,t$\delta$) = hd sigseq in let h′ = chg_$H\Sigma$(h,h$\sigma$);
  [39]   wait t$\delta$;
  [39]   sig_seq(h′)(tl sigseq) end end end
```

■ End of Example 8

We refer to Appendix Sect. J (Pages 186–189) for a more substantial coverage of the concept of support technologies.

3.4 Rules and Regulations 75

By a **rule** we shall understand a text which describe how the domain is — i.e., how people and technology are — expected to behave. The meaning of a rule is a predicate over "before/after" states of actions (simple, one step behaviours): if the predicate holds then the rule has been obeyed. 76

By a **regulation** we shall understand a text which describes actions to be performed should its corresponding rule fail to hold. The meaning of a regulation is therefore a state-to-state transition, one that brings the domain into a rule-holding "after" state. 77

Example 9 – **Rules** We give two examples related to railway systems where train stations are the hubs and the rail tracks between train stations are the links:

41 Trains arriving at or leaving train stations:

  a) (In China:) No two trains

  b) must arrive at or leave a train station

  c) in any two minute time interval. 78

42 Trains travelling "down" a railway track. We must introduce a notion of links being a sequence of adjacent sectors.

  a) Trains must travel in the same direction;

  b) and there must be at least one "free-from-trains" sector

  c) between any two such trains.

We omit showing somewhat "lengthy" formalisations. ■ End of Example 9

We omit exemplification of regulations.

We refer to Appendix Sect. K (Pages 190–191) for a more substantial coverage of the concepts of rules and regulations.

3.5 Scripts, Licenses and Contracts 79

3.5.1 Scripts

By a script we understand a usually structured set of pairs of rules and regulations — structured, for example, as a simple "algorithm description". 80

Example 10 – **Timetable Scripts**

43 Time is considered discrete. Bus lines and bus rides have unique names (across any set of time tables).

44 A *T*ime*T*able associates *B*us *L*ine *Id*entifiers (*blid*) to sets of *Journies*.

45 *Journies* are designated by a pair of a *BusRoute* and a set of *BusRides*.

46 A *BusRoute* is a triple of the *Bus Stop* of origin, a list of zero, one or more intermediate *Bus Stop*s and a destination *Bus Stop*.

47 A set of *BusRides* associates, to each of a number of *B*us *Id*entifiers (*bid*) a *Bus Sched*ule.

48 A *Bus Sched*ule is a triple of the initial departure *T*ime, a list of zero, one or more intermediate bus stop *T*imes and a destination arrival *T*ime. 81

49 A *Bus Stop* (i.e., its position) is a *Frac*tion of the distance along a link (identified by a *L*ink *I*dentifier) *f*rom an *i*dentified *h*ub *to* an *i*dentified *h*ub.

50 A *Frac*tion is a **Real** properly between 0 and 1.

51 The *Journies* must be *w*ell_*f*ormed in the context of some net.

52 A set of journies is well-formed if

> 53 the bus stops are all different,
>
> 54 a bus line is embedded in some line of the net, and
>
> 55 all defined bus trips of a bus line are equivalent.

82

type
  [43] T, BLId, BId
  [44] TT = BLId $\underset{m}{\rightarrow}$ Journies
  [45] Journies′ = BusRoute × BusRides
  [46] BusRoute = BusStop × BusStop* × BusStop
  [47] BusRides = BId $\underset{m}{\rightarrow}$ BusSched
  [49] BusSched = T × T* × T
  [50] BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
  [51] Frac = {|r:**Real**•0<r<1|}
  [45] Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}
value
  [52] wf_Journies: Journies → N → **Bool**
  [52] wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
  [53]   diff_bus_stops(bs1,bsl,bsn) ∧
  [54]   is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
  [55]   commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

■ End of Example 10

We refer to Appendix Sect. L.1 (Pages 192–198) for a more substantial coverage of the concept of scripts. 83

Timetables are used in Example 11 on the following page.

3.5.2 Licenses and Contracts

By a **license** (a **contract**) language we understand a pair of languages of licenses and of the set of actions allowed by the license — such that non-allowable license (contract) actions incur moral obligations (respectively legal responsibilities). 84

Example 11 – Public Bus Transport Contracts

An example contract can be ‘schematised’:

cid: **contractor** cor **contracts sub-contractor** cee
  to perform operations
   {"conduct","cancel","insert","subcontract"}
    with respect to timetable tt.

We assume a context (a global state) in which all contract actions (including contracting) takes
85 place and in which the implicit net is defined.

Concrete examples of actions can be schematised:

(a) cid: **conduct bus ride** (blid,bid) **to start at time** t
(b) cid: **cancel bus ride** (blid,bid) **at time** t
(c) cid: **insert bus ride like** (blid,bid) **at time** t

The schematised license shown earlier is almost like an action; here is the action form:

(d) cid: **contractor** cnm′ **is granted a contract** cid′
  to perform operations
    {”conduct”,”cancel”,”insert”,sublicense”}
  with respect to timetable tt′.

86 All actions are being performed by a sub-contractor in a context which defines that sub-
contractor *cnm*, the relevant net, say *n*, the base contract, referred here to by *cid* (from
which this is a sublicense), and a timetable *tt* of which *tt*′ is a subset. contract name *cnm*′ is
new and is to be unique. The subcontracting action can (thus) be simply transformed into a
87 contract as shown on Page 34.

```
type
   Action = CNm × CId × (SubCon | SmpAct) × Time
   SmpAct = Conduct | Cancel | Insert
   Conduct == μConduct(s_blid:BLId,s_bid:BId)
   Cancel == μCancel(s_blid:BLId,s_bid:BId)
   Insert = μInsert(s_blid:BLId,s_bid:BId)
   SubCon == μSubCon(s_cid:CId,s_cnm:CNm,s_body:body)
                where body = (s_ops:Op-set,s_tt:TT)
```

We omit formalising the semantics of these syntaxes. A formalisation could be expressed (in CSP [56]) with each bus, each licensee (and licensor), time and the road net bus traffic being processes, etc. ■ End of Example 11

We refer to Appendix Sect. L.2 (Pages 198–206) for a more substantial coverage of the concepts of licenses and contracts.

3.6 Management and Organisation 88

By management we shall understand the set of behaviours which perform strategic, tactical
89 and operational actions. By organisation we shall understand the decomposition of these

behaviours into, for example, clearly separate strategic, tactical and operational "areas", possibly further decomposed by geographical and/or "subject matter" concerns. 90

To explain differences between strategic, tactical and operational issues we introduce notions of *strategic, tactical* and *operational funds*, $\mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$, and other *resources*, $\mathbb{R}$, a notion of *contexts*, $\mathbb{C}$, and a notion of *states*, $\mathbb{S}$. Contexts bind resources to bindings from locations to disjoint time intervals (allocation and scheduling), states bind resource identifiers to resource values. 91

Simplified types of the strategic, tactical and operational actions are now of the following types: executive functions apply to contexts, states and funds and obtain and redistribute funds; strategic functions apply to contexts and strategic funds and create new contexts and states and consume some funds; tactical functions apply to resources, contexts, states tactical funds and create new contexts while consuming some tactical funds; etcetera. 92

type
  $\mathbb{R}, \mathbb{RID}, \mathbb{RVAL}, \mathbb{F}_{\mathcal{S}}, \mathbb{F}_{\mathcal{T}}, \mathbb{F}_{\mathcal{O}}$
  $\mathbb{C} = \mathbb{R} \underset{m}{\overrightarrow{\ }} ((\mathbb{T} \times \mathbb{T}) \underset{m}{\overrightarrow{\ }} \mathbb{L})$
  $\mathbb{S} = \mathbb{RID} \underset{m}{\overrightarrow{\ }} \mathbb{RVAL}$

value
  $\omega\mathbb{RID}$: $\mathbb{R} \rightarrow \mathbb{RID}$
  $\omega\mathbb{RVAL}$: $\mathbb{R} \rightarrow \mathbb{RVAL}$
  Executive_functions: $\mathbb{C} \times \mathbb{S} \times \mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}} \rightarrow \mathbb{F}_{\mathcal{S},\mathcal{T},\mathcal{O}}$
  Strategic_functions: $\mathbb{C} \times \mathbb{F}_{\mathcal{S}} \rightarrow \mathbb{F}_{\mathcal{S}} \times \mathbb{R} \times \mathbb{C} \times \mathbb{S}$
  Tactic_functions: $\mathbb{R} \times \mathbb{C} \times \mathbb{S} \times \mathbb{F}_{\mathcal{T}} \rightarrow \mathbb{C} \times \mathbb{F}_{\mathcal{T}}$
  Operational_functions: $\mathbb{C} \times \mathbb{S} \times \mathbb{F}_{\mathcal{O}} \rightarrow \mathbb{S} \times \mathbb{F}_{\mathcal{O}}$

93

Example 12 – **Public Bus Transport Management** We relate to Example 11:

56 The **conduct, cancel** and **insert bus ride** actions are operational functions.

57 The actual **subcontract** actions are tactical functions;

58 but the decision to carry out such a tactical function may very well be a strategic function as would be the acquisition or disposal of busses.

59 Forming new timetables, in consort with the contractor, is a strategic function.

We omit formalisations. ■ End of Example 12

We refer to Appendix Sect. M (Pages 229–230) for a more substantial coverage of the concepts of management and organisation.

3.7 Human Behaviour 94

By human behaviour we shall understand those aspects of the behaviour of domain stakeholders which have a direct bearing on the "functioning" of the domain Behaviours "fall" in a spectrum from diligent via sloppy to delinquent and outright criminal neglect in the
95 observance of maintaining entities, carrying our actions and responding to events.

Example 13 – **Human Behaviour** Cf. Examples 11–12:

60 no failures to conduct a bus ride must be classified as diligent;

61 rare failures to conduct a bus ride must be classified as sloppy if no technical reasons were the cause;

62 occasional failures ··· as delinquent;

63 repeated patterns of failures ··· as criminal.

We omit showing somewhat "lengthy" formalisations. ■ End of Example 13

We refer to Appendix Sect. N (Pages 231–234) for a more substantial coverage of the concept of human behaviour.

3.8 Discussion 96

We have briefly outlined six concepts of domain facets and we have exemplified each of these. Real-scale domain descriptions are, of course, much larger than what we can show. Typically, say for the domain of logistics, a basic description is approximately 30 pages; for "small" parts of railway systems we easily get up to 100–200 pages – both including formalisations. The reader should now have gotten a reasonably clear idea as to what
97 constitutes a domain description. As mentioned, in the introduction to this section, i.e., Sect. 3, we shall not cover post-modelling activities such a validation and domain theory formation. The latter is usually part of the verification (theorem proving, model checking and formal testing) of the formal domain description. Final validation of a domain description is with respect to the narrative part of the narrative/formalisation pairs of descriptions. The reader should also be able to form a technical opinion about what can be formalised, and that not all can be formalised within the framework of a single formal specification language, cf. Sect. 1.5.

4 Requirements Engineering 98

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. We distinguish 99 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. 100

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 exemply the latter, then the former.

101

Example 14 – Goals of a Toll Road System

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

■ End of Example 14

102

Example 15 – Goals of Toll Road System Software

- 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.

■ End of Example 15

Goals are usually expressed in terms of properties. Requirements can then be proved to satisfy the $\mathcal{G}$oals: $\mathcal{D}, \mathcal{R} \models \mathcal{G}$. [62, Lamsweerde] focus on goals. 103

Example 16 – Arguing Goal-satisfaction of a Toll Road System

- By endowing links and hubs with average traversal times for both ordinary road and for toll road links and hubs
  - 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.

■ End of Example 16

104

Example 17 – Arguing Goal-satisfaction of Toll Road System Software

- By recording
  - tickets issued and collected at toll boths and
  - toll road hubs and links entered and left
  - as per the requirements specification brought in (forthcoming) Examples 19-23,
- we can eventually argue that
  - the requirements of (the forthcoming) Examples 19-23
  - help satisfy the goal of Example 15 on the previous page.

■ End of Example 17

105

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

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

4.1 Business Process Re-engineering 106

In Sect. 3.1 we very briefly covered a notion of business processes. These were the business processes of the domain before installation of the required computing systems. The potential of installing computing systems invariably requires revision of established business processes. Business process re-engineering (BPR) is a development of new business processes – whether or not complemented by computing and communication. BPR, such as we advocate it, proceeds on the basis of an existing domain description and outlines needed changes (additions, deletions, modifications) to entities, actions, events and behaviours following the six domain facets outlined in Sects. 3.2–3.7. The goals help us formulate the BPR prescriptions. 107

Example 18 – Rough-sketching a Re-engineered Road Net Our sketch centers 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 (Sects. 3.2–3.7). 108

64 **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. (Example 20 brings more details.) 109

65 **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. 110

66 **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. 111

67 **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). 112

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

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

70 **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. 113

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

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

73 **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".

■ End of Example 18

We refer to Appendix Sect. O (Pages 235–239) for a more substantial coverage of the concept of business process re-engineering.

4.2 Domain Requirements 114

For the phase of domain requirements the requirements stakeholders "sit together" with the domain cum requirements engineers and read the domain description, line-by-line, in order to "derive" the domain requirements. They do so in five rounds (in which the BPR rough sketch is both regularly referred to and possibly, i.e., most likely regularly updated). Domain requirements are "derived" from the domain description as covered in Sects. 4.2.1–4.2.5. The goals then determine the derivations: which projections, instantiations, determinations, etcetera, to perform.

4.2.1 Projection 115

By *domain projection* we understand an operation that applies to a domain description and yields a domain requirements prescription. The latter represents a projection of the former in which only those parts of the domain are present that shall be of interest in the ongoing requirements development 116

Example 19 – Projection
Our requirements is for a simple toll road: a linear sequence of links and hubs outlined in Example 18: see Items [1–11] of Example 1 on page 21 and Items [32–35] of Example 7 on page 29.

■ End of Example 19

4.2.2 Instantiation 117

By *domain instantiation* we understand an operation that applies to a (projected) domain description, i.e., a requirements prescription, and yields a domain requirements prescription, where the latter has been made more specific, usually by constraining a domain description. 118

Example 20 – Instantiation
Here the toll road net topology as outlined in Example 18 on page 38 is introduced: a straight sequence of toll road hubs pairwise connected with pairs of one way links and with each hub two way link connected to a toll road plaza. 119

type
$\mathsf{H, L, P = H}$
$\mathsf{N' = (H \times L) \times H \times ((L \times L) \times H \times (H \times L))^*}$

```
    N″ = {|n:N′•wf(n)|}
value
    wf_N″: N′ → Bool
    wf_N″((h,l),h′,llhpl) ≡ ... 6  lines ... !
    αN: N″ → N
    αN((h,l),h′,llhpl) ≡ ... 2 lines ... !
```

wf_N″ secures linearity; αN allows abstraction from more concrete N″ to more abstract N.

■ End of Example 20

4.2.3 Determination 120

By *domain determination* we understand an operation that applies to a (projected and possibly instantiated) domain description, i.e., a requirements prescription, and yields a domain requirements prescription, where (attributes of) entities, actions, events and behaviours have been made less indeterminate. 121

Example 21 – Determination

Pairs of links between toll way hubs are open in opposite directions; all hubs are open in all directions; links between toll way hubs and toll plazas are open in both directions. 122

```
type
    LΣ = (HI×HI)-set, LΩ = LΣ-set
    HΣ = (LI×LI)-set, HΩ = HΣ-set
    N′ = (H × L) × H × ((L × L) × H × (H × L))*
value
    ωLΣ: L → LΣ, ωLΩ: L → LΩ
    ωHΣ: H → HΣ, ωHΩ: H → HΩ
axiom
    ∀ ((h,l),h′,llhhl:⟨(l′,l″),h″,(h‴,l‴)⟩⌢llhhl′):N″ •
        ωLΣ(l)={(ωHI(h),ωHI(h′)),(ωHI(h′),ωHI(h))}∧
        ωLΣ(l‴)={(ωHI(h″),ωHI(h‴)),(ωHI(h‴),ωHI(h″))}∧
        ∀ i,i+1:Nat • {i,i+1}⊆inds llhhl ⇒
            let ((li,li′),hi,(hi″,li″))=llhhl(i), (_,hj,(hj″,lj″))=llhhl(i+1) in
            ωLΩ(li)= {{(ωHI(hi),ωHI(hj))}}∧ωLΩ(li′)={{(ωHI(hj),ωHI(hi))}}∧
            ωHΩ(hi)= { ... } ... 3 lines end
```

■ End of Example 21

4.2.4 Extension 123

By *domain extension* we understand an operation that applies to a (projected and possibly determined and instantiated) domain description, i.e., a (domain) requirements prescription, and yields a (domain) requirements prescription. The latter prescribes that a software system is to support, partially or fully, entities, operations, events and/or behaviours that were not feasible (or not computable in reasonable time or space) in a domain without computing support, but which are now are not only feasible but also computable in reasonable time and space. 124

Example 22 – **Extension** We extend the domain by introducing toll road entry and exit booths as well as electronic ticket hub sensors and actuators. There should now follow a careful narrative and formalisation of these three machines: the car driver/machine "dialogues" upon entry and exit as well as the sensor/car/actuator machine "dialogues" when cars enter hubs. The description should first, we suggest, be ideal; then it should take into account failures of booth equipment, electronic tickets, car drivers, and of sensors and actuators.

■ End of Example 22

4.2.5 Fitting 125

By *domain requirements fitting* we understand an operation which takes two or more (say n) domain requirements prescriptions, d_{r_i}, that are claimed to share entities, actions, events and/or behaviours and map these into n+1 domain requirements prescriptions, δ_{r_i}, where one of these, $\delta_{r_{n+1}}$ capture the shared phenomena and concepts and the other n prescriptions, δ_{r_i}, 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 $\delta_{r_{n+1}}$, prescribe interfaces between δ_{r_i} and $\delta_{r_{n+1}}$ for $i : \{1..n\}$. 126

Example 23 – **Fitting** We assume three ongoing requirements development projects, all focused around road transport net software systems: (i) road maintenance, (ii) toll road car monitoring and (iii) bus services on ordinary plus toll road nets. The main shared phenomenon is the road net, i.e., the links and the hubs. The consolidated, shared road net domain requirements prescription, $\delta_{r_{n+1}}$, is to become a prescription for the domain requirements for shared hubs and links. Tuples of these relations then prescribe representation of all hub, respectively all link attributes – common to the three applications. Functions (including actions) on hubs and links become database queries and updates. Etc. ■ End of Example 23

4.2.6 Discussion: 127

This section has very briefly surveyed and illustrated domain requirements. The reader should take cognizance of the fact that these are indeed "derived" from the domain description. They are not domain descriptions, but, once the business process re-engineering has been adopted and the required software has been installed, then the domain requirements become part of a revised domain description !

4.3 Interface Requirements 128

By interface requirements we understand such requirements which are concerned with the phenomena and concepts *shared* between the domain and the machine. Thus such requirements can only be expressed using terms from both the domain and the machine. We tackle the problem of "deriving", i.e., constructing interface requirements by tackling four "smaller" problems:those of "deriving" interface requirements for entities, actions, events and behaviours respectively. Again goals help state which phenomena and concepts are to be shared.

4.3.1 Entity Interfaces 129

Entities that are shared between the domain and the machine must initially be input to the machine. Dynamically arising or attribute value changing entities must likewise be input and all such machine entities must have their attributes updated, when need arise. Requirements for shared entities thus entail requirements for their representation and for their human/machine and/or machine/machine transfer-dialogues. 130

Example 24 – Shared Entities

Main shared entities are those of hubs and links. We suggest that eventually a relational database be used for representing hubs links in relations. 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. ■ End of Example 24

4.3.2 Action Interfaces 131

By a shared action we mean an action that can only be partly computed by the machine. That is, the machine, in order to complete an action, may have to inquire with the domain (some measurable, time-varying entity attribute value, or some domain stakeholder) in order to proceed in its computation. 132

Example 25 – Shared Actions In order for a car driver to leave an exit toll both the following component actions must take place: the driver inserts the electronic pass in the exit toll booth machine; the machine scans and accepts the ticket and calculates the fee for the car journey from entry booth via the toll road net to the exit booth; the driver is alerted to the cost and is requested to pay this amount; once paid the exit booth toll gate is raised. *Notice that a number of details of the new support technology is left out. It could either be elaborated upon here, or be part of the system design.* ■ End of Example 25

4.3.3 Event Interfaces 133

By a shared event we mean an event whose occurrence in the domain need be communicated to the machine – and, vice-versa, an event whose occurrence in the machine need be communicated to the domain. 134

Example 26 – Shared Events 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 at a toll plaza exit booth 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. ■ End of Example 26

4.3.4 Behaviour Interfaces 135

By a shared behaviour we understand a sequence of zero, one or more shared actions and shared events. 136

Example 27 – **Shared Behaviour** 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 moretoll 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. Example 26. ■ End of Example 27

4.3.5 Discussion 137

The discussion of Sect. 4.2.6 carries over to this section. That is, once the machine has been installed it, the machine, is part of the new domain !

4.4 Machine Requirements 138

We shall not cover this stage of requirements development other than saying that it consists of the following concerns: performance requirements (storage, speed, other resources), dependability requirements (availability, accessibility, integrity, reliability, safety, security), maintainability requirements (adaptive, extensional, corrective, perfective, preventive), portability requirements (development platform, execution platform, maintenance platform, demo platform) and documentation requirements. 139 Only dependability seems to be subjectable to rigorous, formal treatment. We refer to [12, Vol. 3, Part V, Chap. 19, Sect. 19.6] for an extensive (30 page) survey.

The **discussions** of Sects. 4.2.6 and 4.3.5 carry over to this paragraph. That is, once the machine has been installed it, the machine, is part of the new domain !

5 Discussion 140

We discuss a number of issues that were left open above.

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

5.2 Domain Descriptions Are Not Normative 141

The 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 those of clearing ('matching') sell offers and buy bids. These rules and regulations appears to be quite different from those of the *Tokyo Stock Exchange* [81]. 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" 142

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". 143

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 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 144

The issue of *"what can be described"* has been a constant challenge to philosophers. In
[76, 1919] Russell covers his first *Theory of Descriptions*, and in [75] a revision, as *The
Philosophy of Logical Atomism*. The issue is not that straightforward. In [17, 20] we try
to broach the topic from the point of view of the kind of domain engineering presented in
145 this paper.

Our approach is simple; perhaps too simple ! We can describe what can be observed.
We do so, first by 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
146 tions 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 147

Section 1.6 made some claims. We think we have substantiated them all, albeit ever so
148 briefly.

Each of the domain facets(intrinsics, support technologies, rules and regulations, scripts
149 [licenses and contracts], management and organisation and human behaviour) and each of
the requirements facets(projection, instantiation, determination, extension and fitting) provide
vide rich grounds for both specification methodology studies and and for more theoretical
studies [15, ICTAC 2007].

5.6 Relation to Other Work 150

The most obvious 'other' work is that of [60, Problem Frames]. In [60] Jackson, like is done
here, departs radically from conventional requirements engineering. In his approach understandings
derstandings of the domain, the requirements and possible software designs are arrived at,
151 not hierarchically, but in parallel, interacting streams of decomposition. Thus the 'Problem
lem Frame' development approach iterates between concerns of domains, requirements and
software design. "Ideally" our approach pursues domain engineering prior to requirements
152 engineering, and, the latter, prior to software design. But see next.

The recent book [62, 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 [62] apply to determining which projections, instantiation, determination and extension operations to perform on domain descriptions.

5.7 "Ideal" Versus Real Developments 153

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. 154

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 155

We have used the `RSL` specification language, [48, 12], for the formalisations of this report, but any of the model-oriented approaches and languages offered by `Alloy` [59], `Event B` [1], `RAISE` [49], `VDM` [46] and `Z` [84], should work as well. 156

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` [73], `CSP` [56], `MSC` [57], `Statecharts` [54], and/or some temporal logic, for example either `DC` [85] or `TLA+` [61]. Research into how such diverse textual and diagrammatic languages can be combined is ongoing [3].

5.9 $\mathcal{D},\mathcal{S} \models \mathcal{R}$ 157

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 [27].

5.10 Domain Versus Ontology Engineering 158

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 needes, cf. Sect. 1.5.

6 Conclusion 159

These lecture notes are based on the paper:

```
From Domains to Requirements
Submitted for publication
December 7, 2009
```

Versions of that paper are found on the Internet”

```
www.imm.dtu.dk/~db/short-from-domains-to-requirements.pdf
www.imm.dtu.dk/~db/long-from-domains-to-requirements.pdf
```

The examples of the short version are without formulas. The examples of the long version are with formulas. 160

The idea of extending that (8-11 page two column) paper into a brief set of lectures notes and slides arose in connection with the author’s April 2010 lectures at the Technical University of Vienna. In addition to a normal format paper a full-fledged “RSL primer”, a number of clarifying methodology sections and further examples have been added as appendices. 161

The formalisations of these lecture notes (and slides) which are expressed in RSL, the RAISE Specification Languange, can be expressed in either of Alloy, Event B, VDM-SL or Z.

The present author would like to work with “enthusiasts” (i.e., “followers”) of the above-listed specification languages to achieve versions of these lecture notes (and slides) for any and all of these other formal specification languages.

7 Bibliographical Notes

Section 1.5 gives most relevant references to formal specification languages (techniques and tools) that cover the spectrum of domain and requirements specification, refinement and verification. The recent book on Logics of Specification Languages [24] covers `ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM` and `Z`.

References

[1] J.-R. Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[2] B. Aichernig and T. Maibaum, editors. *UNU-IIST 10th Anniversary Colloquium on Formal Methods at the Crossroads, from Panacea to Foundational Support; Lisboa, Portugal*, volume 2757 of *Lecture Notes in Computer Science*. Springer, March 2002.

[3] 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.

[4] Y. Arimoto and D. Bjørner. Hospital Healthcare: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.

[5] P. Bernus and L. Nemes, editors. *Modelling and Methodologies for Enterprise Integration*, International Federation for Information Processing, London, UK, 1996 1995. IFIP TC5, Chapman & Hall. Working Conference, Queensland, Australia, November 1995.

[6] D. Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. In *2nd Asia-Pacific Software Engineering Conference (APSEC '95)*. IEEE Computer Society, 6–9 December 1995. Brisbane, Queensland, Australia.

[7] D. Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press.

[8] D. Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki.

[9] D. Bjørner. New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems. In *FORMS2003: Symposium on Formal Methods for Railway*

Operation and Control Systems. Institut für Verkehrssicherheit und Automatisierungstechnik, Techn.Univ. of Braunschweig, Germany, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.

[10] D. Bjørner. Documents: A Domain Analysis. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] D. Bjørner. Domain Engineering. In *The 2007 Lipari PhD Summer School*, Lecture Notes in Computer Science (eds. E. Börger and A. Ferro), pages 1–102, Heidelberg, Germany, 2009. Springer. To appear. Meanwhile check with http://www2.imm.dtu.dk/~db/container-paper.pdf.

[19] 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, 2009. Springer.

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

[21] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. JAIST Press, March 2009. The monograph contains the following chapters: [36, 37, 38, 39, 35, 40, 41, 42, 43, 44].

[22] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. JAIST Press, March 2009. JAIST Research Monograph #4, 536 pages: http://www2.imm.dtu.dk/˜db/jaistmono.pdf.

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

[24] D. Bjørner and M. C. Henson, editors. *Logics of Specification Languages*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.

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

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

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

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

[29] W. Clancey. The knowledge–level reinterpreted: modeling socio–technical systems. *International Journal of Intelligent Systems*, 8:33–49, 1993.

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

[31] B. L. Clarke. Individuals and Points. *Notre Dame J. Formal Logic*, 26(1):61–75, 1985.

[32] G. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, pages 430–440. IEEE, 1984.

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

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

[35] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*, chapter 5: The Triptych Process Model – Process Assessment and Improvement, pages 107–138. JAIST Press, March 2009.

[36] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 1: On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management, pages 3–38. JAIST Press, March 2009.

[37] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 2: Possible Collaborative Domain Projects – A Management Brief, pages 39–56. JAIST Press, March 2009.

[38] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 3: The Rôle of Domain Engineering in Software Development, pages 57–72. JAIST Press, March 2009.

[39] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 4: Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal, pages 73–106. JAIST Press, March 2009.

[40] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 6: Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm, pages 139–175. JAIST Press, March 2009.

[41] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 7: Documents – A Rough Sketch Domain Analysis, pages 179–200. JAIST Press, March 2009.

[42] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 8: Public Government – A Rough Sketch Domain Analysis, pages 201–222. JAIST Press, March 2009.

[43] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 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.

[44] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering [21]*, chapter 10: Towards a Family of Script Languages – – Licenses and Contracts – Incomplete Sketch, pages 283–328. JAIST Press, March 2009.

[45] D. J. Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.

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

[47] 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.

[48] 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.

[49] 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.

[50] T. R. Gruber and G. R. Olsen. An Ontology for Engineering Mathematics. In J. Doyle, P. Torasso, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1994. Fourth International Conference. Gustav Stresemann Institut, Bonn, Germany.[4].

[51] M. Gruninger and M. Fox. The Logic of Enterprise Modelling. In *Modelling and Methodologies for Enterprise Integration, see [5]*, pages 141–157, November 1995.

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

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

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

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

[56] T. 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).

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

[4] http://www-ksl.stanford.edu/knowledge-sharing/papers/engmath.html

[58] D. Jackson. A Direct Path to Dependable Software. *CACM: Communications of the ACM*, 52(4):78–88, April 2009.

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

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

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

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

[63] J. McCarthy. Towards a Mathematical Science of Computation. In C. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.

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

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

[66] A. Prior. *Changes in Events and Changes in Things*, chapter in [65]. Oxford University Press, 1993.

[67] A. N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.

[68] A. N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.

[69] A. N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.

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

[71] M. Pěnička and D. Bjørner. From Railway Resource Planning to Train Operation — a Brief Survey of Complementary Formalisations. In *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart*, pages 629–636. Kluwer Academic Publishers, August 2004.

[72] M. Pěnička, A. K. Strupchanska, and D. Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.

[73] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Institut für Informatik, Humboldt Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany, 1 Oktober 2009. 276 pages. http://www2.informatik.hu-berlin.de/top/pnene_buch/pnene_buch.pdf.

[74] G. Rochelle. *Behind time: The incoherence of time and McTaggart's atemporal replacement*. Avebury series in philosophy. Ashgate, Brookfield, Vt., USA, 1998. vii + 221 pages.

[75] 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.

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

[77] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pws Pub Co, August 17, 1999. ISBN: 0534949657, 512 pages, Amazon price: US $ 70.95.

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

[79] 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.

[80] A. K. Strupchanska, M. Pěnička, and D. Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.

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

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

[83] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

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

[85] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

Part II

5 More Lectures Days

A An RSL Primer 162

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 value are type, 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.

163

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″, ...

164

Example 28 – Basic Net Attributes:

- For safe, uncluttered traffic, hubs and links can 'carry' a maximum of vehicles.

- Links have lengths. (We ignore hub (traveersal) lengths.)
- One can calculate whether a link is a two-way link.

165

```
type
   MAX = Nat
   LEN = Real
   is_Two_Way_Link = Bool
value
   ωMax: (H|L) → MAX
   ωLen: L → LEN
   is_two_way_link: L → is_Two_Way_Link
   is_two_way_link(l) ≡ ∃ lσ:LΣ • lσ ∈ ωHΣ(l)∧card lσ=2
```

■ End of Example 28

Composite Types 166

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.

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

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω
[12] $A \underset{m}{\rightarrow} B$
[13] A → B
[14] $A \overset{\sim}{\rightarrow} B$
[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 × C × ... × D, in which case it is identical to type expression kind 9,

- or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\xrightarrow[m]{}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\xrightarrow[m]{}$ (C|D|(E $\xrightarrow[m]{}$ F)), etc.

16. The postulated disjoint union of types A, B, . . . , and C.

17. The record type of mk_id-named record values mk_id(av,...,bv), where av, . . . , bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

18. The record type of unnamed record values (av,...,bv), where av, . . . , bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

167

Example 29 – **Composite Net Type Expressions:**

The type clauses of function signatures:

```
value
   f: A → B
```

often have the type expressions *A* and/or *B* be composite type expressions: 168

```
value
   ωHIs: L → HI-set          Example 1 Item [5]
   ωLIs: H → LI-set          Example 1 Item [6]
   ωHΣ: H → HT-set           Example 1 Item [10]
   set_HΣ: H × HΣ → H        Example 2 Item [12]
```

169Right-hand sides of type definitions often have composite type expressions:

```
type
   N = H-set × L-set          Example 1 Item [2]
   HT = LI × HI × LI          Example 1 Item [9]
   LT′ = HI × LI × HI         Example 7 Item [32]
```

■ End of Example 29

A.1.2 Type Definitions 170

Concrete Types

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

```
type
   A = Type_expr
```

171

Example 30 – 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. 172

  type
   [sorts] N_α, H, L, HI, LI
  value
   ωHs: $N_\alpha \rightarrow$ H-**set**
   ωLs: $N_\alpha \rightarrow$ L-**set**
  axiom
   $\forall$ n:N_α • **card** ωHs(n)$\geq$2 $\wedge$ **card** ωLs(n)$\geq$1 $\wedge$ ...

  where the ... shall account for what has been in expressed in axioms [5–8] of Example 1 on page 21. 173

- **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: $N_\beta \rightarrow$ **Bool**
   wf_N_β(hs,ls) $\equiv$ **card** hs$\geq$2 $\wedge$ **card** ls$\geq$1 ....
   inject_N_β: $N_\alpha \overset{\sim}{\rightarrow} N_\beta$ **pre**: wf_N_β(hs,ls)
   inject_N_β(n_α) $\equiv$ (ωHs(n_α),ωLs(n_α))

  where the ... shall account for what has been in expressed in axioms [5–8] of Example 1 on page 21.

  174

- **Cartesians + Maps + Wellformedness:** Or a net can be modelled as a triple of

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

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

175

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

- [a,b] *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 39 on page 80 defines the well-formedness predicates for the above map types.

■ End of Example 30

Some schematic type definitions are: 176

[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) |}

where a form of [2–3] is provided by combining the types: 177

Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
   ∀ a1:A_1, a2:A_2, ..., ai:Ai •
      s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
      ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
   ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
      a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

178

Example 31 – 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.[5] The removal command specifies a link identifier.

179

```
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.
  - ◊ 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 ?? on page ?? presents the semantics functions for *int_Insert* and *int_Remove*.
. ■ End of Example 31

180

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

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

181

Example 32 – Net Subtypes:

In Example 30 on page 60 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. 182

- In the **Sorts + Observers + Axioms** part of Example 30
  - 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:

183

```
type
   [sorts] N′, H, L, HI, LI
                N = {|n:N′ • wf_N(n)|}
value
   wf_N: N′ → Bool
   wf_N(n) ≡
      ∀ n:N • card ωHs(n)≥2 ∧ card ωLs(n)≥1 ∧
      [5−−8] of example 1
```

184

- In the **Cartesians + Wellformedness** part of Example 30
  - 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)|}
```

185

- In the **Cartesians + Maps + Wellformedness** part of Example 30
  - a net was a triple of hubs, links and a graph,
  - each with their wellformednes predicates.

  We now redefine this as follows:

186

```
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 $\overrightarrow{m}$ H
   HUBS = {|hs:HUBS′ • wf_HUBS(hs)|}
   LINKS′ = LI → L
   LINKS = {|ls:LINKS′ • wf_LINKS(ls)|}
   GRAPH′ = HI $\overrightarrow{m}$ (LI $\overrightarrow{m}$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 39 on page 80 presents a definition of *wf_GRAPH*. ■ End of Example 32

187

Sorts — Abstract Types

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

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

188

Example 33 – Net Sorts:

In formula lines of Examples 30–32 we have indicated those **type** clauses which define *sorts*, by bracketed [sorts] literals. ■ End of Example 33

A.2 Concrete RSL Types: Values and Operations 189

A.2.1 Arithmetic

type
  $\mathbf{Nat}, \mathbf{Int}, \mathbf{Real}$
value
  $+,-,*\!: \mathbf{Nat}\times\mathbf{Nat}\to\mathbf{Nat} \mid \mathbf{Int}\times\mathbf{Int}\to\mathbf{Int} \mid \mathbf{Real}\times\mathbf{Real}\to\mathbf{Real}$
  $/\!: \mathbf{Nat}\times\mathbf{Nat}\stackrel{\sim}{\to}\mathbf{Nat} \mid \mathbf{Int}\times\mathbf{Int}\stackrel{\sim}{\to}\mathbf{Int} \mid \mathbf{Real}\times\mathbf{Real}\stackrel{\sim}{\to}\mathbf{Real}$
  $<,\leq,=,\neq,\geq,>\ (\mathbf{Nat}|\mathbf{Int}|\mathbf{Real}) \times (\mathbf{Nat}|\mathbf{Int}|\mathbf{Real}) \to \mathbf{Bool}$

190

A.2.2 Set Expressions

Set Enumerations

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

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

191

Example 34 – 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. 192

Solely using the set data type and the concept of subtypes, we can model the above:

type
  C
  $\mathsf{G}' = \mathsf{C}\text{-}\mathbf{set},\ \ \mathsf{G} = \{|\ \mathsf{g}{:}\mathsf{G}' \bullet \mathsf{g}\neq\{\}\ |\}$
  $\mathsf{S} = \mathsf{G}\text{-}\mathbf{set}$
  $\mathsf{L}' = \mathsf{C}\text{-}\mathbf{set},\ \ \mathsf{L} = \{|\ \ell{:}\mathsf{L}' \bullet \ell\neq\{\}\ |\}$
  $\mathsf{NW}' = \mathsf{S},\ \ \mathsf{NW} = \{|\ \mathsf{s}{:}\mathsf{S} \bullet \mathsf{wf\_S(s)}\ |\}$
value
  $\mathsf{wf\_S}\!: \mathsf{S} \to \mathbf{Bool}$
  $\mathsf{wf\_S(s)} \equiv \forall\ \mathsf{g}{:}\mathsf{G} \bullet \mathsf{g} \in \mathsf{s} \Rightarrow \exists\ \mathsf{g}'{:}\mathsf{G} \bullet\ \ \mathsf{g}' \in \mathsf{s} \wedge \mathsf{share(g,g')}$
  $\mathsf{share}\!:\ \mathsf{G}\times\mathsf{G} \to \mathbf{Bool}$
  $\mathsf{share(g,g')} \equiv \mathsf{g}\neq\mathsf{g}' \wedge \mathsf{g} \cap \mathsf{g}' \neq \{\}$

```
    liaisons: G×G → L
    liaisons(g,g′) = g ∩ g′ pre share(g,g′)
```

193

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! 194

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 34

195

Set Comprehension

The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm. 196

```
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)}
```

197

Example 35 – Set Comprehensions:

Example 30 on page 60 illustrates, in the **Cartesians + Maps + Wellformedness** part the following set comprehensions in the *wf_N(hs,ls,g)* wellformedness predicate definition:

$[d] \cup \{\mathbf{dom}\ g(hi) | hi{:}HI \bullet hi \in \mathbf{dom}\ g\}$

It expresses the distributed union of sets (**dom** *g(hi)*) of link identfiers (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** *g*). 198

[e] ∪ {**rng** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*}
It expresses the distributed union of sets (**rng** *g(hi)*) of hub identfiers (for each of the *hi* indexed maps from (definition set, **dom**) link identiers to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **deom** *g*). ■ End of Example 35

199

A.2.3 Cartesian Expressions

Cartesian Enumerations

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

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

200

Example 36 – **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. 201

- 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.

202

type
LN, HN, LEN, LOC
$\mathsf{cL} = \mathsf{LI} \times \mathsf{LN} \times (\mathsf{HI} \times \mathsf{HI}) \times \mathsf{LOC} \times ...$
$\mathsf{cH} = \mathsf{HI} \times \mathsf{HN} \times \mathsf{LI}\text{-}\mathbf{set} \times \mathsf{LOC} \times ...$

■ End of Example 36

203

A.2.4 List Expressions

List Enumerations

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

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

$\langle$ 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. 204

List Comprehension

The last line below expresses list comprehension.

type
A, B, P = A → **Bool**, Q = A $\xrightarrow{\sim}$ B
value
comprehend: $A^\omega \times P \times Q \xrightarrow{\sim} B^\omega$
comprehend(l,P,Q) ≡
$\langle$ Q(l(i)) | i **in** $\langle$1..**len** l$\rangle$ • P(l(i))$\rangle$

205

Example 37 – 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.

206

```
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 ⇒
         ωHIs(l(i)) ∩ ωHIs(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 ⇒ ωLI(pr(i))=ar(i)
```

207

- 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 $r\widehat{\ }r'$ 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.

208

```
value
  routes: N → PR-infset
  routes(n) ≡
    let prs = {⟨l⟩|l:L•ωLs(n)} ∪
                  ∪ {pr⌢pr′|pr,pr′:PR•{pr,pr′}⊆prs∧ωHIs(r(len pr))∩ωHIs(pr′(1))≠{}}
    prs end

  is_circular: PR → Bool
  is_circular(pr) ≡ ωHIs(pr(1))∩ωHIs(pr(len pr))≠{}

  is_looped: PR → Bool
  is_looped(pr) ≡ ∃ i,j:Nat • i≠j∧{i,j}⊆index pr ⇒ ωHIs(pr(i))∩ωHIs(pr(j))≠{}
```

209

- 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 37

210

A.2.5 Map Expressions

Map Enumerations

Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

type

$$\begin{array}{l}
\quad \mathrm{T1,\ T2} \\
\quad \mathrm{M = T1} \underset{m}{\rightarrow} \mathrm{T2} \\
\textbf{value} \\
\quad \mathrm{u,u1,u2,...,un{:}T1,\ v,v1,v2,...,vn{:}T2} \\
\quad \{[\,],\ [\mathrm{u}\mapsto\mathrm{v}],\ ...,\ [\mathrm{u1}\mapsto\mathrm{v1},\mathrm{u2}\mapsto\mathrm{v2},...,\mathrm{un}\mapsto\mathrm{vn}],...\} \subseteq \mathrm{M}
\end{array}$$

211

Map Comprehension

The last line below expresses map comprehension:

$$\begin{array}{l}
\textbf{type} \\
\quad \mathrm{U,\ V,\ X,\ Y} \\
\quad \mathrm{M = U} \underset{m}{\rightarrow} \mathrm{V} \\
\quad \mathrm{F = U} \overset{\sim}{\rightarrow} \mathrm{X} \\
\quad \mathrm{G = V} \overset{\sim}{\rightarrow} \mathrm{Y} \\
\quad \mathrm{P = U} \rightarrow \textbf{Bool} \\
\textbf{value} \\
\quad \mathrm{comprehend{:}\ M{\times}F{\times}G{\times}P} \rightarrow (\mathrm{X} \underset{m}{\rightarrow} \mathrm{Y}) \\
\quad \mathrm{comprehend(m,F,G,P)} \equiv \\
\qquad [\ \mathrm{F(u)} \mapsto \mathrm{G(m(u))} \mid \mathrm{u{:}U} \bullet \mathrm{u} \in \textbf{dom}\ \mathrm{m} \wedge \mathrm{P(u)}\,]
\end{array}$$

212

Example 38 – Concrete Net Type Construction:

- We Define a function *con[struct]_N*$_\gamma$ (of the **Cartesians + Maps + Wellformedness** part of Example 30.
  - The base of the construction is the fully abstract sort definition of N_α in the **Sorts + Observers + Axioms** part of Example 30 – 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 30.
  - First we recall the ssential types of that N_γ.

213

$$\begin{array}{l}
\textbf{type} \\
\quad \mathsf{N}_\gamma = \mathsf{HUBS} \times \mathsf{LINKS} \times \mathsf{GRAPH} \\
\quad \mathsf{HUBS} = \mathsf{HI} \underset{m}{\rightarrow} \mathsf{H} \\
\quad \mathsf{LINKS} = \mathsf{LI} \underset{m}{\rightarrow} \mathsf{L} \\
\quad \mathsf{GRAPH} = \mathsf{HI} \underset{m}{\rightarrow} (\mathsf{LI} \underset{m}{\rightarrow} \mathsf{HI}) \\
\textbf{value}
\end{array}$$

```
$\mathsf{con\_N}_\gamma\text{: } \mathsf{N}_\alpha \to \mathsf{N}_\gamma$
$\mathsf{con\_N}_\gamma(\mathsf{n}_\alpha) \equiv$
    $\textbf{let } \mathsf{hubs} = [\,\omega\mathsf{HI(h)} \mapsto \mathsf{h} \mid \mathsf{h{:}H} \bullet \mathsf{h} \in \omega\mathsf{Hs}(\mathsf{n}_\alpha)\,],$
        $\mathsf{links} = [\,\omega\mathsf{LI(h)} \mapsto \mathsf{l} \mid \mathsf{l{:}L} \bullet \mathsf{l} \in \omega\mathsf{Ls}(\mathsf{n}_\alpha)\,],$
        $\mathsf{graph} = [\,\omega\mathsf{HI(h)} \mapsto [\,\omega\mathsf{LI(l)} \mapsto \iota(\omega\mathsf{HIs(l)}\backslash\{\omega\mathsf{HI(h)}\})$
                                                  $\mid \mathsf{l{:}L} \bullet \mathsf{l} \in \omega\mathsf{Ls}(\mathsf{n}_\alpha) \wedge \mathsf{li} \in \omega\mathsf{LIs(h)}\,]$
                                    $\mid \mathsf{H{:}h} \bullet \mathsf{h} \in \omega\mathsf{Hs}(\mathsf{n}_\alpha)\,]\textbf{ in}$
        $(\mathsf{hubs.links,graph})\textbf{ end}$

$\iota\text{: } \mathsf{A}\text{-}\mathbf{set} \xrightarrow{\sim} \mathsf{A}\ [\,\mathsf{A\ could\ be\ LI}\text{-}\mathbf{set}\,]$
$\iota(\mathsf{as}) \equiv \textbf{if card}\,\mathsf{as}{=}1\textbf{ then let } \{\mathsf{a}\}{=}\mathsf{as}\textbf{ in } \mathsf{a}\textbf{ else chaos end end}$
```

214

theorem:

n_α **satisfies axioms** [2,5–8] for N of Example 1 $\Rightarrow \mathsf{wf\_N}_\gamma\mathsf{con\_N}_\gamma(\mathsf{n}_\alpha)$

■ End of Example 38

215

A.2.6 Set Operations

Set Operator Signatures

value

21 $\in$: A × A-**infset** → **Bool**
22 $\notin$: A × A-**infset** → **Bool**
23 $\cup$: A-**infset** × A-**infset** → A-**infset**
24 $\cup$: (A-**infset**)-**infset** → A-**infset**
25 $\cap$: A-**infset** × A-**infset** → A-**infset**
26 $\cap$: (A-**infset**)-**infset** → A-**infset**
27 $\backslash$: A-**infset** × A-**infset** → A-**infset**
28 $\subset$: A-**infset** × A-**infset** → **Bool**
29 $\subseteq$: A-**infset** × A-**infset** → **Bool**
30 $=$: A-**infset** × A-**infset** → **Bool**
31 $\neq$: A-**infset** × A-**infset** → **Bool**
32 **card**: A-**infset** $\xrightarrow{\sim}$ **Nat**

216

Set Examples

examples

$a \in \{a,b,c\}$
$a \notin \{\}$, $a \notin \{b,c\}$

$\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
$\cup\{\{a\},\{a,b\},\{a,d\}\} = \{a,b,d\}$
$\{a,b,c\} \cap \{c,d,e\} = \{c\}$
$\cap\{\{a\},\{a,b\},\{a,d\}\} = \{a\}$
$\{a,b,c\} \setminus \{c,d\} = \{a,b\}$
$\{a,b\} \subset \{a,b,c\}$
$\{a,b,c\} \subseteq \{a,b,c\}$
$\{a,b,c\} = \{a,b,c\}$
$\{a,b,c\} \neq \{a,b\}$
card $\{\} = 0$, **card** $\{a,b,c\} = 3$

217

Informal Explication

21. $\in$: The membership operator expresses that an element is a member of a set.

22. $\notin$: The nonmembership operator expresses that an element is not a member of a set.

23. $\cup$: 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. $\cup$: 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. $\cap$: 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. $\cap$: 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. 218

27. $\setminus$: 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. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.

29. $\subset$: 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. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.

32. **card**: The cardinality operator gives the number of elements in a finite set.

219

Set Operator Definitions

The operations can be defined as follows (≡ is the definition symbol):

```
value
   s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
   s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
   s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
   s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
   s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
   s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
   s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
   card s ≡
      if s = {} then 0 else
      let a:A • a ∈ s in 1 + card (s \ {a}) end end
      pre s /∗ is a finite set ∗/
   card s ≡ chaos /∗ tests for infinity of s ∗/
```

220

A.2.7 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
```

221

A.2.8 List Operations

List Operator Signatures

```
value
   hd: Aω →̃ A
   tl: Aω →̃ Aω
   len: Aω →̃ Nat
   inds: Aω → Nat-infset
   elems: Aω → A-infset
   .(.): Aω × Nat →̃ A
```

$\widehat{}$: $A^* \times A^\omega \rightarrow A^\omega$
$=$: $A^\omega \times A^\omega \rightarrow$ **Bool**
$\neq$: $A^\omega \times A^\omega \rightarrow$ **Bool**

222

List Operation Examples

examples
hd⟨a1,a2,...,am⟩=a1
tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
len⟨a1,a2,...,am⟩=m
inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩⁀⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

223

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list. 224
- ⁀: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows: 225

List Operator Definitions

```
value
    is_finite_list: $A^\omega$ → 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 }
```

226

```
    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″)
```

227

A.2.9 Map Operations

Map Operator Signatures and Map Operation Examples

```
value
    m(a): M → A $\overset{\sim}{\rightarrow}$ B, m(a) = b

    dom: M → A-infset [domain of map]
```

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

rng: M → B-**infset** [range of map]
$\mathbf{rng}\,[\,a1\mapsto b1,a2\mapsto b2,...,an\mapsto bn\,] = \{b1,b2,...,bn\}$

†: M × M → M [override extension]
$[\,a\mapsto b,a'\mapsto b',a''\mapsto b''\,] \dagger [\,a'\mapsto b'',a''\mapsto b'\,] = [\,a\mapsto b,a'\mapsto b'',a''\mapsto b'\,]$

228

∪: M × M → M [merge ∪]
$[\,a\mapsto b,a'\mapsto b',a''\mapsto b''\,] \cup [\,a'''\mapsto b'''\,] = [\,a\mapsto b,a'\mapsto b',a''\mapsto b'',a'''\mapsto b'''\,]$

\: M × A-**infset** → M [restriction by]
$[\,a\mapsto b,a'\mapsto b',a''\mapsto b''\,]\backslash\{a\} = [\,a'\mapsto b',a''\mapsto b''\,]$

/: M × A-**infset** → M [restriction to]
$[\,a\mapsto b,a'\mapsto b',a''\mapsto b''\,]/\{a',a''\} = [\,a'\mapsto b',a''\mapsto b''\,]$

=,≠: M × M → **Bool**

°: $(A \underset{m}{\rightarrow} B) \times (B \underset{m}{\rightarrow} C) \rightarrow (A \underset{m}{\rightarrow} C)$ [composition]
$[\,a\mapsto b,a'\mapsto b'\,] \circ [\,b\mapsto c,b'\mapsto c',b''\mapsto c''\,] = [\,a\mapsto c,a'\mapsto c'\,]$

229

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps. 230
- \: 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.

- ≠: The nonequal operator expresses that the two operand maps are *not* identical.

- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

231

Example 39 – **Miscellaneous Net Expressions: Maps:**

Example 30 on page 60 left out defining the well-formedness of the map types:

```
value
      wf_HUBS: HUBS → Bool
   [a] wf_HUBS(hubs) ≡ ∀ hi:HI • hi ∈ dom hubs ⇒ ωHIhubs(hi)=hi
      wf_LINKS: LINKS → Bool
   [b] wf_LINKS(links) ≡ ∀ li:LI • li ∈ dom links ⇒ ωLIlinks(li)=li
   wf_N_γ: N_γ → Bool
   wf_N_γ(hs,ls,g) ≡
      [c] dom hs = dom g ∧
      [d] ∪ {dom g(hi)|hi:HI • hi ∈ dom g} = dom links ∧
      [e] ∪ {rng g(hi)|hi:HI • hi ∈ dom g} = dom g ∧
      [f] ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒ (g(hi))(li)≠hi
      [g] ∀ 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
```

232

- [c] *HUBS* record the same hubs as do the net corresponding *GRAPHS* (**dom** *hs* = **dom** *g* ∧).

- [d] *GRAPHS* record the same links as do the net corresponding *LINKS* (∪ {**dom** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *links*).

- [e] 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 !

- [f] 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*).

- [g] 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 39

Map Operation Redefinitions 233

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) ]
```

234

```
   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 235

A.3.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

A.3.2 Simple Predicate Expressions 236

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, . . ., k designate number values, then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

A.3.3 Quantified Expressions 237

Let X, Y, . . ., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds. 238

Example 40 – Predicates Over Net Quantities:

From earlier examples we show some predicates:

- Example 28: Right hand side of function definition *is_two_way_link(l):*

  $\exists\ l\sigma{:}L\Sigma \bullet l\sigma \in \omega H\Sigma(l) \wedge \mathbf{card}\ l\sigma{=}2$ 239

- Example 30:
  - The **Sorts + Observers + Axioms** part:
    - Right hand side of the wellformedness function *wf_N(n)* definition:
      $\forall\ n{:}N \bullet \mathbf{card}\,\omega Hs(n){\geq}2 \wedge \mathbf{card}\,\omega Ls(n){\geq}1 \wedge [\,5{-}{-}8\,]$ *of example 1*
    - Right hand side of the wellformedness function *wf_N(hs,ls)* definition:
      $\mathbf{card}\ hs{\geq}2 \wedge \mathbf{card}\ ls{\geq}1 \ldots$

  240

  - The **Cartesians + Maps + Wellformedness** part:
    - Right hand side of the *wf_HUBS* wellformedness function definition:
      $\forall\ hi{:}HI \bullet hi \in \mathbf{dom}\ hubs \Rightarrow \omega HIhubs(hi){=}hi$
    - Right hand side of the *wf_LINKS* wellformedness function definition:
      $\forall\ li{:}LI \bullet li \in \mathbf{dom}\ links \Rightarrow \omega LIlinks(li){=}li$
    - Right hand side of the *wf_N(7 hs,ls,g)* wellformedness function definition:
      $[\,c\,]\ \mathbf{dom}\ hs = \mathbf{dom}\ g \wedge$
      $[\,d\,] \cup \{\mathbf{dom}\ g(hi) | hi{:}HI \bullet hi \in \mathbf{dom}\ g\} = \mathbf{dom}\ links \wedge$
      $[\,e\,] \cup \{\mathbf{rng}\ g(hi) | hi{:}HI \bullet hi \in \mathbf{dom}\ g\} = \mathbf{dom}\ g \wedge$
      $[\,f\,]\ \forall\ hi{:}HI \bullet hi \in \mathbf{dom}\ g \Rightarrow \forall\quad li{:}LI \bullet li \in \mathbf{dom}\ g(hi) \Rightarrow (g(hi))(li){\neq}hi$
      $[\,g\,]\ \forall\ hi{:}HI \bullet hi \in \mathbf{dom}\ g \Rightarrow \forall\ li{:}LI \bullet li \in \mathbf{dom}\ g(hi) \Rightarrow$
      $\quad\exists\ hi'{:}HI \bullet hi' \in \mathbf{dom}\ g \Rightarrow \exists\ !\ li{:}LI \bullet li \in \mathbf{dom}\ g(hi) \Rightarrow$
      $\quad\quad(g(hi))(li) = hi' \wedge (g(hi'))(li) = hi$

■ End of Example 40

A.4 λ-Calculus + Functions 241

A.4.1 The λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
    ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

A.4.2 Free and Bound Variables 242

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in λy •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).

A.4.3 Substitution 243

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;

- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;

- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

A.4.4 α-Renaming and β-Reduction 244

- 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) $\equiv$ **subst**([N/x]M)

A.4.5 An Example 245

Example 41 – 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. 246
- 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: ...).

247

```
value
    n:N
type
    T, V
    TS = T-infset
axiom
    $\forall$ ts:TS • $\exists$ tmin,tmax:T: tmin $\in$ ts $\wedge$ tmax $\in$ ts $\wedge$ $\forall$ t:T • t $\in$ ts $\Rightarrow$ tmin $\leq$ t $\leq$ tmax
    [that is: ts = {$\mathcal{MIN}$(ts)..$\mathcal{MAX}$(ts)}]
type
    VP = V $\xrightarrow[m]{}$ Pos
    TF$'$ = T $\rightarrow$ 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$\rightarrow$ N $\rightarrow$ Bool
    wf_TF(tf)(n) $\equiv$ ...
    $\mathcal{DOMAIN}$: TF $\rightarrow$ TS
    $\mathcal{MIN}$, $\mathcal{MAX}$: TS $\rightarrow$ T
```

248

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 $\times$ T $\times$ TF $\rightarrow$ TF $\rightarrow$ TF
    insert_V(v,t,vtf)(tf) as tf$'$
```

249

- 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: $\mathcal{MIN}(\mathcal{DOMAIN}(\mathsf{tf}) \leq \mathsf{t} \leq \mathcal{MAX}(\mathcal{DOMAIN}(\mathsf{tf})) \wedge$
     $\forall\ \mathsf{t}':\mathsf{T} \bullet \mathsf{t}' \in \mathcal{DOMAIN}(\mathsf{tf}) \Rightarrow \mathsf{v} \notin \mathbf{dom}\ \mathsf{tf}(\mathsf{t}') \wedge$
     $\mathcal{MIN}(\mathcal{DOMAIN}(\mathsf{vtf})) = \mathsf{t} \wedge$
     $\forall\ \mathsf{t}':\mathsf{T}\bullet\mathsf{t}' \in \mathcal{DOMAIN}(\mathsf{vtf}) \Rightarrow \mathbf{dom}\ \mathsf{vtf}(\mathsf{t}')=\{\mathsf{v}\}$
```

250

- 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 $(vtf(t))(v)$.

251

```
post: $\mathsf{tf}' = \lambda \mathsf{t}''\bullet$
            let $\mathsf{ts} = \mathcal{DOMAIN}(\mathsf{tf}) \cup \mathcal{DOMAIN}(\mathsf{vtf})$ in
            if $\mathcal{MIN}(\mathsf{ts}) \leq \mathsf{t}'' \leq \mathcal{MAX}(\mathsf{ts})$
               then
                  $((\mathsf{t}''<\mathsf{t}) \rightarrow \mathsf{tf}(\mathsf{t}''),$
                  $(\mathsf{t}''\geq\mathsf{t}) \rightarrow [\,\mathsf{v}'\mapsto$ if $\mathsf{v}'\neq\mathsf{v}$ then $(\mathsf{tf}(\mathsf{t}))(\mathsf{v}')$ else $(\mathsf{vtf}(\mathsf{t}))(\mathsf{v})$ end
                                    $|\mathsf{v}':\mathsf{V}\bullet\mathsf{v}' \in \mathsf{vehicles}(\mathsf{tf})\,])$
               else chaos end
            end
assumption: $\mathsf{wf\_TF}(\mathsf{vtf})\wedge\mathsf{wf\_TF}(\mathsf{tf})$
theorem: $\mathsf{wf\_TF}(\mathsf{tf}')$

value
   vehicles: TF $\rightarrow$ V-set
```

vehicles(tf) ≡ {v|t:T,v:V•t ∈ $\mathcal{DOMAIN}$(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 41

A.4.6 Function Signatures 252

For sorts we may want to postulate some functions:

```
type
   A, B, ..., C
value
   ωB: A → B
   ...
   ωC: 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. 253

Example 42 – Hub and Link Observers:

Let a net with several hubs and links be presented. Now observer functions ωHs and ωLs 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. 254

The observer functions ωHI and ωLI 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 42

A.4.7 Function Definitions 255

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
```

a_expr, b_expr are A, respectively B valued expressions of any of the kinds illustrated in earlier and later sections of this primer. 256

Or functions can be defined implicitly:

```
value
    f: $A \rightarrow B$
    f(a_expr) as b
    post P(a_expr,b)
    P: $A \times B \rightarrow$ Bool
```

```
g: $A \overset{\sim}{\rightarrow} B$
g(a_expr) as b
pre P′(a_expr)
post P(a_expr,b)
P′: $A \rightarrow$ Bool
```

where *b* is just an identifier.

The symbol $\overset{\sim}{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function. 257

Finally functions, f, g, ..., can be defined in terms of axioms over function identifiers, f, g, ..., and over identiers of function arguments and results.

```
type
    A, B, C, D, ...
value
    f: $A \rightarrow B$
    g: $C \rightarrow D$
    ...
axiom
    $\forall$ a:A, b:B, c:C, d:D, ...
        $\mathcal{P}_1$(f,a,b) $\wedge$ ... $\wedge$ $\mathcal{P}_m$(f,a,b)
        ...
        $\mathcal{Q}_1$(g,c,d) $\wedge$ ... $\wedge$ $\mathcal{Q}_n$(g,c,d)
```

where $\mathcal{P}_1, \ldots, \mathcal{P}_m$ and $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ designate suitable predicate expressions. 258

Example 43 – Axioms over Hubs, Links and Their Observers:

Example 1 on page 21 Items [4]–[8] clearly demonstrates how a number of entities and observer functions are constrained (that is, partially defined) by function signatures and axioms.
. ■ End of Example 43

A.5 Other Applicative Expressions 259

A.5.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

$$\mathbf{let}\ \mathrm{a} = \mathcal{E}_d\ \mathbf{in}\ \mathcal{E}_b(\mathrm{a})\ \mathbf{end}$$

is an “expanded” form of:

$$(\lambda \mathrm{a}.\mathcal{E}_b(\mathrm{a}))(\mathcal{E}_d)$$

A.5.2 Recursive let Expressions 260

Recursive **let** expressions are written as:

$$\begin{array}{l}
\mathbf{let}\ \mathrm{f} = \lambda \mathrm{a}{\bullet}\mathrm{E}(\mathrm{f},\mathrm{a})\ \mathbf{in}\ \mathrm{B}(\mathrm{f},\mathrm{a})\ \mathbf{end} \\
\mathbf{let}\ \mathrm{f} = (\lambda \mathrm{g} \lambda \mathrm{a}{\bullet}\mathrm{E}(\mathrm{g},\mathrm{a}))(\mathrm{f})\ \mathbf{in}\ \mathrm{B}(\mathrm{f}.\mathrm{a})\ \mathbf{end} \\
\mathbf{let}\ \mathrm{f} = \mathrm{F}(\mathrm{f})\ \mathbf{in}\ \mathrm{E}(\mathrm{f},\mathrm{a})\ \mathbf{end\ where}\ \mathrm{F} \equiv \lambda \mathrm{g} \lambda \mathrm{a}{\bullet}\mathrm{E}(\mathrm{g},\mathrm{a}) \\
\mathbf{let}\ \mathrm{f} = \mathbf{Y}\mathrm{F}\ \mathbf{in}\ \mathrm{B}(\mathrm{f},\mathrm{a})\ \mathbf{end\ where}\ \mathbf{Y}\mathrm{F} = \mathrm{F}(\mathbf{Y}\mathrm{F})
\end{array}$$

We read $\mathsf{f} = \mathbf{Y}\mathsf{F}$ as “*f is a fix point of F*”.

A.5.3 Non-deterministic let Clause 261

The non-deterministic **let** clause:

$$\mathbf{let}\ \mathrm{a}{:}\mathrm{A} \bullet \mathcal{P}(\mathrm{a})\ \mathbf{in}\ \mathcal{B}(\mathrm{a})\ \mathbf{end}$$

expresses the non-deterministic selection of a value a of type A which satisfies a predicate $\mathcal{P}(\mathsf{a})$ for evaluation in the body $\mathcal{B}(\mathsf{a})$. If no $\mathsf{a{:}A} \bullet \mathsf{P(a)}$ the clause evaluates to **chaos**.

A.5.4 Pattern and “Wild Card” let Expressions 262

Patterns and *wild cards* can be used:

$$\begin{array}{l}
\mathbf{let}\ \{\mathrm{a}\} \cup \mathrm{s} = \mathrm{set}\ \mathbf{in}\ ...\ \mathbf{end} \\
\mathbf{let}\ \{\mathrm{a},\_\} \cup \mathrm{s} = \mathrm{set}\ \mathbf{in}\ ...\ \mathbf{end} \\
\\
\mathbf{let}\ (\mathrm{a},\mathrm{b},...,\mathrm{c}) = \mathrm{cart}\ \mathbf{in}\ ...\ \mathbf{end} \\
\mathbf{let}\ (\mathrm{a},\_,...,\mathrm{c}) = \mathrm{cart}\ \mathbf{in}\ ...\ \mathbf{end} \\
\\
\mathbf{let}\ \langle \mathrm{a} \rangle ^\frown \ell = \mathrm{list}\ \mathbf{in}\ ...\ \mathbf{end} \\
\mathbf{let}\ \langle \mathrm{a},\_,\mathrm{b} \rangle ^\frown \ell = \mathrm{list}\ \mathbf{in}\ ...\ \mathbf{end} \\
\\
\mathbf{let}\ [\,\mathrm{a} \mapsto \mathrm{b}\,] \cup \mathrm{m} = \mathrm{map}\ \mathbf{in}\ ...\ \mathbf{end} \\
\mathbf{let}\ [\,\mathrm{a} \mapsto \mathrm{b},\_\,] \cup \mathrm{m} = \mathrm{map}\ \mathbf{in}\ ...\ \mathbf{end}
\end{array}$$

A.5.5 Conditionals 263

Various kinds of conditional expressions are offered by RSL:

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /* same as: */
    if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
    choice_pattern_1 → expr_1,
    choice_pattern_2 → expr_2,
    ...
    choice_pattern_n_or_wild_card → expr_n end
```

264

Example 44 – 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. 265

35. If the command is of the form 2oldH(hi′,l,hi′) then

    ⋆1 hi′ must be the identifier of a hub in hs,

    ⋆s2 l must not be in ls and its identifier must (also) not be observable in ls, and

    ⋆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

    ⋆1 hi must be the identifier of a hub in hs,

    ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and

    ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.

266

37. If the command is of the form 2newH(h′,l,h″) then

    ⋆1 h′ — left to the reader as an exercise (see formalisation !),

    ⋆2 l — left to the reader as an exercise (see formalisation !), and

    ⋆3 h″ — left to the reader as an exercise (see formalisation !).

Conditions concerning the new link (second ⋆s, ⋆2, in the above three cases) can be expressed independent of the insert command category. 267

```
value
    33  int_Insert: Insert → N $\xrightarrow{\sim}$ 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 $\xrightarrow{\sim}$ N names a value, int_Insert, and defines its type to be Insert → N $\xrightarrow{\sim}$ N, that is, a partial function ($\xrightarrow{\sim}$) 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 74 Item 25 on page 75.

End of RSL Explanation

268

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

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: $H \times LI \to H$, rLI: $H \times LI \xrightarrow{\sim} H$
   40:  aLI(h,li) as $h'$
           pre li $\notin$ obs_LIs(h)
           post obs_LIs($h'$) = {li} $\cup$ obs_LIs(h) $\wedge$ non_l_eq(h,$h'$)
   41:  rLI($h'$,li) as h
           pre li $\in$ obs_LIs($h'$) $\wedge$ card obs_LIs($h'$)$\geq$2
           post obs_LIs(h) = obs_LIs($h'$) \ {li} $\wedge$ 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 $\notin$ 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} $\wedge$ 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 $\in$ obs_LIs(h') $\wedge$ **card** obs_LIs(h')$\geq$2 expresses that the link identifier li must not be observable h.
  - post-condition clause **post** obs_LIs(h) = obs_LIs(h') \ {li} $\wedge$ 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

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, $\cup$, 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.

271

```
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 hsδ = {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)}∪ hsδ,ls ∪{l}) end
⋆j    end
⋆k    pre pre_int_Insert(op)(hs,ls)
```

RSL Explanation

- $\star_i$–$\star_j$: The clause **case** op **of** $p_1 \to c_1$, $p_2 \to c_2$, $\ldots p_n \to c_n$ **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

272

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.

273

```
value
  45   int_Remove: Rmv → N ∼→ 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 44

A.5.6 Operator/Operand Expressions 274

$$
\begin{array}{ll}
\langle\text{Expr}\rangle ::= & \\
& \ \langle\text{Prefix\_Op}\rangle\ \langle\text{Expr}\rangle \\
& \mid \langle\text{Expr}\rangle\ \langle\text{Infix\_Op}\rangle\ \langle\text{Expr}\rangle \\
& \mid \langle\text{Expr}\rangle\ \langle\text{Suffix\_Op}\rangle \\
& \mid \ldots \\
\langle\text{Prefix\_Op}\rangle ::= & \\
& - \mid \sim \mid \cup \mid \cap \mid \mathbf{card} \mid \mathbf{len} \mid \mathbf{inds} \mid \mathbf{elems} \mid \mathbf{hd} \mid \mathbf{tl} \mid \mathbf{dom} \mid \mathbf{rng} \\
\langle\text{Infix\_Op}\rangle ::= & \\
& = \mid \neq \mid \equiv \mid + \mid - \mid * \mid \uparrow \mid / \mid < \mid \leq \mid \geq \mid > \mid \wedge \mid \vee \mid \Rightarrow \\
& \mid \in \mid \notin \mid \cup \mid \cap \mid \backslash \mid \subset \mid \subseteq \mid \supseteq \mid \supset \mid \widehat{\ } \mid \dagger \mid \circ \\
\langle\text{Suffix\_Op}\rangle ::= & \ !
\end{array}
$$

A.6 Imperative Constructs 275

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 276

```
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

277

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

A.6.6 Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

A.7 Process Constructs 278

A.7.1 Process Channels

Let A, B and D stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

```
channel
    c,c′:A
channel
    {k[ i ]|i:KIdx}:B
    {ch[ i ]i:KIdx}:B
```

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B). 279

Example 45 – Modelling Connected Links and Hubs:

Examples (45–48) 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. 280

- 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.

```
type
    N, V, VI
value
    n:N, vs:V-set
    ωVI: V → VI
type
    H, L, HI, LI, M
    IVH = VI×HI, IVL = VI×LI, IHL = HI×LI
```

281

- 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

$\mathsf{hs{:}H\text{-}\mathbf{set}} = \omega\mathsf{Hs(n)},\ \mathsf{ls{:}L\text{-}\mathbf{set}} = \omega\mathsf{Ls(n)}$

$\mathsf{his{:}HI\text{-}\mathbf{set}} = \{\omega\mathsf{HI(h)}|\mathsf{h{:}H}\bullet\mathsf{h} \in \mathsf{hs}\},\ \mathsf{lis{:}LI\text{-}\mathbf{set}} = \{\omega\mathsf{LI(h)}|\mathsf{l{:}L}\bullet\mathsf{l} \in \mathsf{ls}\},$

$\mathsf{ivhs{:}IVH\text{-}\mathbf{set}} = \{(\omega\mathsf{VI(v)},\omega\mathsf{HI(h)})|\mathsf{v{:}V,h{:}H}\bullet\mathsf{v} \in \mathsf{vs}\wedge\mathsf{h} \in \mathsf{hs}\}$

$\mathsf{ivls{:}IVL\text{-}\mathbf{set}} = \{(\omega\mathsf{VI(v)},\omega\mathsf{LI(l)})|\mathsf{v{:}V,l{:}L}\bullet\mathsf{v} \in \mathsf{vs}\wedge\mathsf{l} \in \mathsf{ls}\}$

$\mathsf{ihls{:}IHL\text{-}\mathbf{set}} = \{\mathsf{(hi,li)}|\mathsf{h{:}H,(hi,li){:}IHL}\bullet\ \mathsf{h} \in \mathsf{hs}\wedge\mathsf{hi}=\omega\mathsf{HI(h)}\wedge\mathsf{li} \in \omega\mathsf{LIs(h)}\}$

282

- We are now ready to declare the channels:
  - a set of channels, $\{\mathsf{vh[\,i\,]}|\mathsf{i{:}IVH}\bullet\mathsf{i}\in\mathsf{ivhs}\}$ between vehicles and all potentially traversable hubs;
  - a set of channels, $\{\mathsf{vh[\,i\,]}|\mathsf{i{:}IVH}\bullet\mathsf{i}\in\mathsf{ivhs}\}$ between vehicles and all potentially traversable links; and
  - a set of channels, $\{\mathsf{hl[\,i\,]}|\mathsf{i{:}IHL}\bullet\mathsf{i}\in\mathsf{ihls}\}$, between hubs and connected links.

channel

$\{\mathsf{vh[\,i\,]}\ |\ \mathsf{i{:}IVH}\ \bullet\ \mathsf{i} \in \mathsf{ivhs}\}\ :\ \mathsf{M}$

$\{\mathsf{vl[\,i\,]}\ |\ \mathsf{i{:}IVL}\ \bullet\ \mathsf{i} \in \mathsf{ivls}\}\ :\ \mathsf{M}$

$\{\mathsf{hl[\,i\,]}\ |\ \mathsf{i{:}IHL}\ \bullet\ \mathsf{i} \in \mathsf{ihls}\}\ :\ \mathsf{M}$

■ End of Example 45

A.7.2 Process Definitions 283

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. 284

```
value
   P: Unit → in c out k[ i ] Unit
   Q: i:KIdx →  out c in k[ i ] Unit
   P() ≡ ... c ? ... k[ i ] ! e ... ; P()
   Q(i) ≡ ... k[ i ] ? ... c ! e ... ; Q(i)

   R:  Unit →  out c in k[ i ]  B
   S:  i:KIdx →  out c in k[ i ]  D
   R() ≡ ... c′ ? ... ch[ i ] ! e ... ; B_Val_Expr
   S(i) ≡ ... ch[ i ] ? ... c ! e ...; D_Val_Expr
```

285

Example 46 – **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.
- Vehicles interact with hubs and links:
  - with potentially all hubs.
  - and with potentially all links.

286

```
value
   hub: hi:HI × h:H → in,out {hl[ (hi,li)|li:LI•li ∈ ωLIs(h) ]}
                              in,out {vh[ (vi,hi)|vi:VI•vi ∈ vis ]}  Unit
   link: li:LI × l:L → in,out {hl[ (hi,li)|hi:HI•hi ∈ ωHIs(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 46

A.7.3 Process Composition 287

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
$\mathcal{P} \;[\!]\; \mathcal{Q}$	Nondeterministic external choice (either/or)
$\mathcal{P} \sqcap \mathcal{Q}$	Nondeterministic internal choice (either/or)
$\mathcal{P}$ ╫ $\mathcal{Q}$	Interlock parallel composition
$\mathcal{O} \; \{ \; \mathcal{P}_i \mid \text{i:Idx} \; \}$	Distributed composition, $\mathcal{O} = \parallel, [\!], \sqcap,$ ╫

express the parallel ($\parallel$) of two processes, or the nondeterministic choice between two processes: either external ($[\!]$) or internal ($\sqcap$). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them
288 terminates.

Example 47 – 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( ωHI(h))(h)|h:H•h ∈ ωHs(n)} ‖
        ‖ {link( ωLI(l))(l)|l:L•l ∈ ωLs(n)} ‖
        ‖ {vehicle(ωVI(v))(ωPN(v))(v)|v:V•v ∈ vs}

    ωPN: V → (Pos×Net)
```

289

- We illustrate a schematic definition of simplified hub processes.
- The hub process alternates, internally non-deterministically, $\sqcap$, between three subprocesses

- 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)($\mathcal{E}_{h_\ell}$(li)(m)(h)) end|i:LI•li ∈ ωLI(h)}
   ⌈⌉ ⌈⌉{let m = vh[(vi,hi)] ? in hub(vi)($\mathcal{E}_{h_v}$(vi)(m)(h)) end|vi:VI•vi ∈ vis}
   ⌈⌉ hub(hi)($\mathcal{E}_{h_{own}}$(h))
```

290

- 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.

```
$\mathcal{E}_{h_\ell}$:  LI → M → H → H
$\mathcal{E}_{h_v}$:  VI → M → H → H
$\mathcal{E}_{h_{own}}$: H → H
```

The reader is encouraged to sketch/define similarly schematic link and vehicle processes.
. ■ End of Example 47

A.7.4 Input/Output Events 291

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. 292

Example 48 – 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.
- 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

293

- 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 $\overrightarrow{m}$ (LI $\overrightarrow{m}$ HI)
  Pos = atH | onL
  atH == μatH(hi:HI)
  onL == μonL(fhi:HI,li:LI,f:F,thi:HI)
  F = {|f:**Real**•0≤f≤1|}

294

- 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.

295

- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
   M == μL_H(li:LI,hi:HI) | μH_L(hi:HI,li:LI)
value
   vehicle: VI → (Pos × Net) → V → Unit
   vehicle(vi)(μatH(hi),net)(v) ≡
   [1]  (wait ;
   [2]   vehicle(vi)(μatH(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)]!μH_L(hi,li)∥vl[(vi,li)]!μH_L(hi,li));
   [8]   vehicle(vi)(μonL(hi,li,0,hi′),net)(v)
   [9]   end end end)
```

296

- 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]).

297

```
type
   M == μL_H(li:LI,hi:HI) | μH_L(hi:HI,li:LI)
value
   δ:Real = move(h,f) axiom 0<δ≪1
   vehicle(vi)( μonL(hi,li,f,hi′),net)(v) ≡
   [1′]  (wait ;
   [2′]   vehicle(vi)(μonL(hi,li,f,hi′),net)(v))
   [3′]  ⌈⌉
   [4′]  (case f of
   [5′]     1 → ((vl[vi,hi′]!μL_H(li,hi′)‖vh[vi,li]!μL_H(li,hi′));
   [6′]           vehicle(vi)(μatH(hi′),net)(v)),
   [7′]     _ → vehicle(vi)(μonL(hi,li,f+δ,hi′),net)(v)
   [8′]   end)
   move: H × F → F
```

■ End of Example 48

A.8 Simple RSL Specifications 298

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: 299

```
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 == μHf(as:Hf)
          | μKf(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)
```

300

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. 301

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
A, B

value
f: A → B

302 which non-deterministically binds f to a function value of type A→B.

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 $(\overline{A_k},\overline{f_\ell},\overline{v\ell})$ is an abbreviation for A_{ℓ_1}, A_{ℓ_2}, $\ldots$, A_t, f_{ℓ_1}, f_{ℓ_2}, $\ldots$, f_{ℓ_f}, v_{ℓ_1}, v_{ℓ_2}, $\ldots$, 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) $\mathcal{P}$. 303

Example 49 – 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. 304

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. 305

- *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.

306

type

```
    N, H, HI, LI, VI
    RPM == μEnter_L(vi:VI,li:LI) | μLeave_L(vi:VI,li:LI)
value
    n:N
channel
    {ves[ωHI(h),li]|h:H•h ∈ ωHs(n)∧li ∈ ωLIs(h)}:VI
    {vls[li,ωHI(h)]|h:H•h ∈ ωHs(n)∧li ∈ ωLIs(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
```

307

To understand the sensor behaviours let us review the vehicle behaviour. In the *vehicle* behaviour defined in Example 48, in two parts, Page 107 and Page 108 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
    vehicle(vi)(pos,net)(v) ≡                                308
    [1] (wait ;
    [2]  vehicle(vi)(pos,net)(v))
    [3]  ⌈⌉
         case pos of
             μatH(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)(μonL(hi,li,0,hi′),net)(v)
    [9]        end end end)
             μonL(hi,li,f,hi′) →
    [4′]      (case f of
    [5′−6′]     1 → (vls[li,hi]!vi; vehicle(vi)(μatH(hi′),net)(v)),
[7′]          _ → vehicle(vi)(μonL(hi,li,f+δ,hi′),net)(v)
[8′]        end)
      end
```

309

- As mentioned on Page 110 *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′} = ωHIs(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!μEnter_LI(vi,li); enter_sensor(hi,li) end
   leave_sensor(li,hi) ≡
      let vi = ves[li,hi]? in rp!μLeave_LI(vi,li); enter_sensor(li,hi) end
```

310

- 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.

```
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]
```

311

```
   road_pricing(lvs,fee,acc) ≡ in rp
      let m = rp? in
      case m of
          μEnter_LI(vi,li) →
                road_pricing(lvs†[li↦lvs(li)∪{vi}],fee,acc),
          $\mu$Leave_LI(vi,li) $\longrightarrow$
                **let** lvs$'$ = **if** vi $\in$ lvs(li) **then** lvs$\dagger$[ li$\mapsto$lvs(li)$\setminus${vi} ] **else** lvs **end**,
                   acc$'$ = payment(vi,li)(fee,acc) **in**
               road_pricing(lvs$'$,fee,acc$'$)
      **end end end**
```

■ End of Example 49

Part III
3 More Lectures Days

B Domain Entities 312

B.1 Entities

The reason for our interest in 'simple entities' is that assemblies and units of systems possess static and dynamic properties which become contexts and states of the processes into which we shall "transform" simple entities.

B.1.1 Observable Phenomena 313

We shall just consider 'simple entities'.[6] By a simple entity we shall here understand a phenomenon that we can designate, viz. see, touch, hear, smell or taste, or measure by some instrument (of physics, incl. chemistry). A simple entity thus has properties. A simple entity is either continuous or is discrete, and then it is either atomic or composite.

Attributes: Types and Values 314

By an attribute we mean a simple property of an entity. *A simple entity has properties* $p_i, p_j, \ldots, p_k$. Typically we express attributes by a pair of a type designator: *the attribute is of type* V, and a value: *the attribute has value* v (of type V, i.e., $v : V$). A simple entity may have many simple properties. A continuous entity, like 'oil', may have the following attributes: type: *petroleum*, kind: *Brent-crude*, amount: *6 barrels*, price: *45 US $/barrel*. An ***atomic*** entity, like a 'person', may have the following attributes: gender: *male*, name: 315 *Dines Bjørner*, birth date: *4. Oct. 1937*, marital status: *married*. A ***composite*** entity, like a railway system, may have the following attributes: country: *Denmark*, name: *DSB*, electrified: *partly*, owner: *independent public enterprise owned by Danish Ministry of Transport*.

Continuous Simple Entities 316

A simple entity is said to be continuous if, within limits, reasonably sizable amounts of the simple entity, can be arbitrarily decomposed into smaller parts each of which still remain simple continuous entities of the same simple entity kind. Examples of continuous entities are: oil, i.e., any fluid, air, i.e., any gas, time period and a measure of fabric.

Discrete Simple Entities 317

A simple entity is said to be discrete if its immediate structure is not continuous. A simple discrete entity may, however, contain continuous sub-entities. Examples of discrete entities are: persons, rail units, oil pipes, a group of persons, a railway line and an oil pipeline.

[6] We use use the name 'simple entities' in contrast to 'entities' which we see as comprising all of simple entities, functions, events and behaviours. "Interesting" functions and normal events involve all forms of entities.

Atomic Simple Entities 318

A simple entity is said to be atomic if it cannot be meaningfully decomposed into parts where these parts has a useful "value" in the context in which the simple entity is viewed and while still remaining an instantiation of that entity. Thus a 'physically able person', which we consider atomic, can, from the point of physical ability, not be decomposed into meaningful parts: a leg, an arm, a head, etc. Other atomic entities could be a rail unit, an oil pipe, or a hospital bed. The only thing characterising an atomic entity are its attributes.

Composite Simple Entities 319

A simple entity, c, is said to be composite if it can be meaningfully decomposed into sub-entities that have separate meaning in the context in which c is viewed. We exemplify some composite entities. (1) A *railway net* can be decomposed into a set of one or more *train lines* and a set of two or more *train stations*. Lines and stations are themselves composite
320 entities. (2) An *Oil industry* whose decomposition include: one or more *oil fields*, one or more *pipeline systems*, one or more *oil refineries* and one or more *one or more oil product distribution systems*. Each of these sub-entities are also composite. Composite simple entities are thus characterisable by their attributes, their sub-entities, and the mereology of how these sub-entities are put together.

B.1.2 Discussion 321

In Sect. B.2.2 we interpreted the model of mereology in six examples. The units of Sect. C.2 which in that section were left uninterpreted now got individuality — in the form of aircraft, building rooms, rail units and oil pipes. Similarly for the assemblies of Sect. C.2.
322 They became pipeline systems, oil refineries, train stations, banks, etc.In conventional modelling the mereology of an infrastructure component, of the kinds exemplified in Sect. B.2.2, was modelled by modelling that infrastructure component's special mereology together, "in line", with the modelling of unit and assembly attributes. With the model of Sect. C.2 now available we do not have to model the mereological aspects, but can, instead, instantiate the model of Sect. C.2 appropriately. We leave that to be reported upon elsewhere. In many conventional infrastructure component models it was often difficult to separate what was mereology from what were attributes.

B.2 Examples of Composite Structures 323

Before a semantic treatment of the concept of mereology let us review what we have done and let us interpret our abstraction (i.e., relate it to actual societal infrastructure components).

B.2.1 What We have Done So Far ? 324

We have presented a model that is claimed to abstract essential mereological properties of machine assemblies, railway nets, the oil industry, oil pipelines, buildings and their installations, hospitals, etcetera.

B.2.2 Six Interpretations 325

Let us substantiate the claims made in the previous paragraph. We will do so, albeit informally, in the next many paragraphs. Our substantiation is a form of diagrammatic reasoning. Subsets of diagrams will be claimed to represent parts, while Other subsets will be claimed to represent connectors. The reasoning is incomplete.

Air Traffic 326

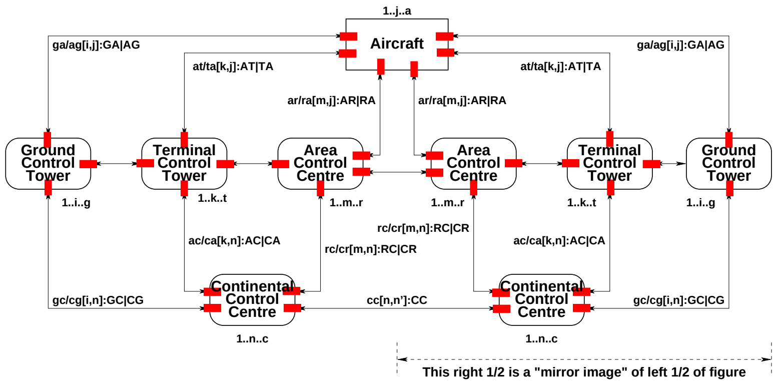


Figure 2: An air traffic system. Black (rounded or edged) boxes and lines are units; **red filled** **boxes** are connections

327

Figure 2 shows nine (9) boxes and eighteen (18) lines. Together they form an assembly. Individually boxes and lines represent units. The rounded corner boxes denote buildings. The sharp corner box denote an aircraft. Lines denote radio telecommunication. Only where lines touch boxes do we have connections. These are shown as red horisontal or vertical boxes at both ends of the double-headed arrows, overlapping both the arrows and the boxes. The index ranges shown attached to, i.e., labelling each unit, shall indicate that there are a multiple of the "single" (thus representative) unit shown. Notice that the 328 'box' units are fixed installations and that the double-headed arrows designate the ether where radio waves may propagate. We could, for example, assume that each such line is characterised by a combination of location and (possibly encrypted) radio communication frequency. That would allow us to consider all line for not overlapping. And if they were overlapping, then that must have been a decision of the air traffic system.

Buildings 329

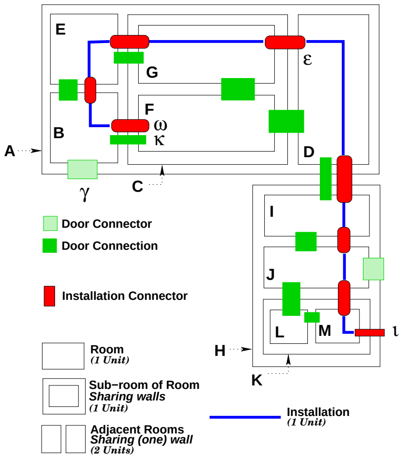


Figure 3: A building plan with installation

330 Figure 3 shows a building plan — as an assembly of two neighbouring, common wall-sharing buildings, A and H, probably built at different times; with room sections B, C, D and E contained within A, and room sections I, J and K within H; with room sections L and
331 M within K, and F and G within C. Connector γ provides means of a connection between A and B. Connection κ provides "access" between B and F. Connectors ι and ω enable input, respectively output adaptors (receptor, resp. outlet) for electricity (or water, or oil), connection ϵ allow electricity (or water, or oil) to be conducted through a wall. Etcetera.

Financial Service Industry 332

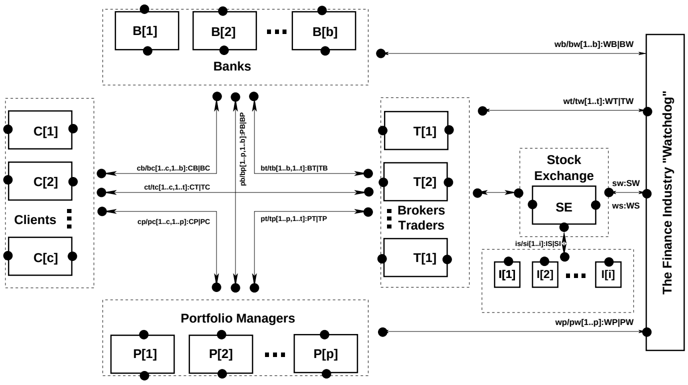


Figure 4: A financial service industry

333

Figure 4 on the facing page shows seven (7) larger boxes [6 of which are shown by dashed lines] and twelve (12) double-arrowed lines. Where double-arrowed lines touch upon (dashed) boxes we have connections (also to inner boxes). Six (6) of the boxes, the dashed line boxes, are assemblies, five (5) of them consisting of a variable number of units; five (5) are here shown as having three units each with bullets "between" them to designate "variability". People, not shown, access the outermost (and hence the "innermost" boxes, but the latter is not shown) through connectors, shown by bullets, •.

Machine Assemblies 334

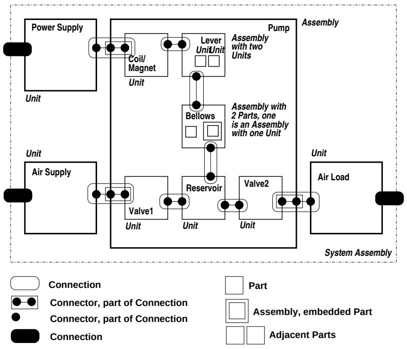


Figure 5: An air pump, i.e., a physical mechanical system

335

Figure 5 shows a machine assembly. Square boxes show assemblies or units. Bullets,
•, show connectors. Strands of two or three bullets on a thin line, encircled by a rounded
box, show connections. The full, i.e., the level 0, assembly consists of four parts and three
internal and three external connections. The Pump unit is an assembly of six (6) parts,
five (5) internal connections and three (3) external connectors. Etcetera. One connector 336
and some connections afford "transmission" of electrical power. Other connections convey
torque. Two connectors convey input air, respectively output air.

Oil Industry 337

338

"The" Overall Assembly Figure 6 on the following page shows an assembly consisting of fourteen (14) assemblies, left-to-right: one oil field, a crude oil pipeline system, two refineries and one, say, gasoline distribution network, two seaports, an ocean (with oil and ethanol tankers and their sea lanes), three (more) seaports, and three, say gasoline and ethanol distribution networks. Between all of the assembly units there are connections, and from some of the assembly units there are connectors (to an external environment). The crude oil pipeline system assembly unit will be concretised next.

339

340

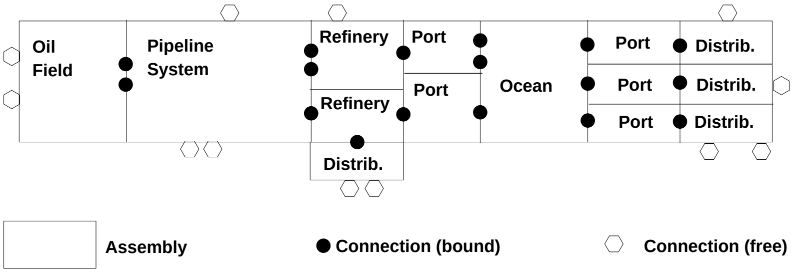


Figure 6: A Schematic of an Oil Industry

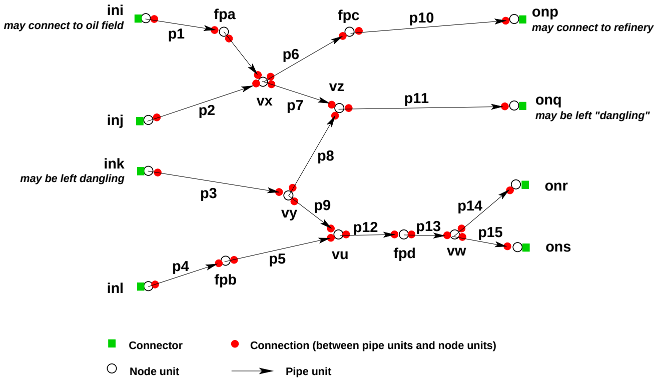


Figure 7: A Pipeline System

A Concretised Assembly Unit Figure 7 on the next page shows a pipeline system. It consists of 32 units: fifteen (15) pipe units (shown as directed arrows and labelled p1–p15), four (4) input node units (shown as small circles, ∘, and labelled ini–inℓ), four (4) flow pump units (shown as small circles, ∘, and labelled fpa–fpd), five (5) valve units (shown as small circles, ∘, and labelled vx–vw), and four (4) output node units (shown as small circles, ∘, and labelled onp–ons). In this example the routes through the pipeline system start with node units and end with node units, alternates between node units and pipe units, and are connected as shown by fully filled-out **red**[7] disc connections. Input and output nodes have input, respectively output connectors, one each, and shown with **green**[8]

Railway Nets

Figure 8 on the facing page diagrams four rail units, each with their two, three or four connectors. Multiple instances of these rail units can be assembled as shown on Fig. 9 on the next page into proper rail nets.

Figure 9 on the facing page diagrams an example of a proper rail net. It is assembled

[7] This paper is most likely not published with colours, so **red** will be shown as **darker colour**.
[8] Shown as **lighter coloure**d connections.

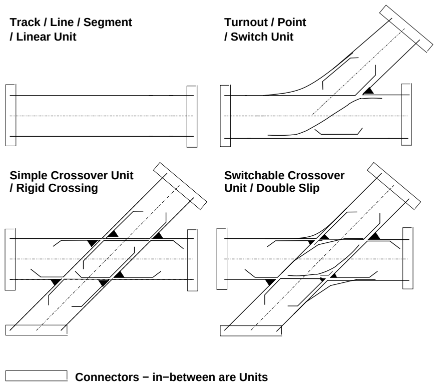


Figure 8: Four example rail units

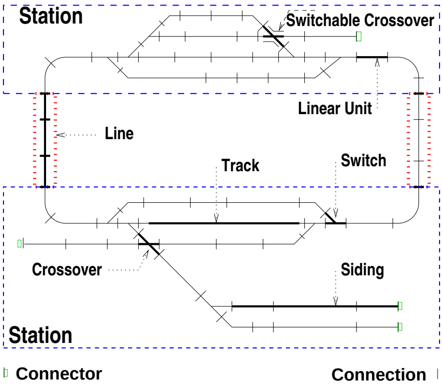


Figure 9: A "model" railway net. An Assembly of four Assemblies:
Two stations and two lines; Lines here consist of linear rail units;
stations of all the kinds of units shown in Fig. 8.
There are 66 connections and four "dangling" connectors

from the kind of units shown in Fig. 8. In Fig. 9 consider just the four dashed boxes: The dashed boxes are assembly units. Two designate stations, two designate lines (tracks) between stations. We refer to to the caption four line text of Fig. 8 for more "statistics". We could have chosen to show, instead, for each of the four "dangling' connectors, a composition of a connection, a special "end block" rail unit and a connector.

B.2.3 Discussion 346

It requires a somewhat more laborious effort, than just "flashing" and commenting on these diagrams, to show that the modelling of essential aspects of their structures can indeed be done by simple instantiation of the model given in the previous section. We can refer to a number of documents which give rather detailed domain models of air traffic [6], container 347 line industry [18][9], financial service industry (banks, credit card companies, brokers, traders and securities and commodities exchanges, insurance companies, etc.)[10], health-care [44, Sects. 10.2.2 + 10.4.2], IT security [43], "the market" (consumers, retailers, wholesalers, producers and distribution chains) [7], "the" oil industry[11], transportation nets[12], railways [71, 80, 72, 8, 9] and [44, Sect. 10.6][13], etcetera, etcetera. Seen in the perspective of the present paper we claim that much of the modelling work done in those references can now be considerably shortened and trust in these models correspondingly increased.

B.3 Attributes and Sub-entities of Sort Values 348

B.3.1 General

Entities are defined in terms of either sorts, that is, abstract types for whose values we do not define mathematical models, or concrete types whose values are sets, Cartesians, lists, maps, functions or other. Entities are either atomic,[14] in which case they are characterised solely in terms of all their attributes (types and values), or are composite, in which case they are characterised in terms of all their attributes (types and values) and all their sub-
349 entities. For both atomic and composite sorts we introduce, as need be, observer functions, whether of attributes or (possibly, if composite) of sub-entities.[15]

In this section we shall introduce and define an equality operator that compares entities modulo some attribute: the name of the equality operator is $\simeq_{\omega_{\mathcal{A}_{attr}}}$, and application of the equality operator to a pair of entities to be compared and the attribute for which comparison is left is expressed: $\simeq_{\mathcal{A}_{attr_A}}$ $(a', a'')(\omega_\alpha)$. To explain this "modulo attribute" equality operator we first $\iota\ell\ell$ustrate[16] the concepts of functions that observe attributes and sub-entities.

B.3.2 Constant and Variable Valued Attributes 350

There are two kinds of attributes to be considered. constant valued attributes, and variable valued attributes. Attributes with variable values are also called entity state components.

[9] http://www2.imm.dtu.dk/~db/container-paper.pdf

[10] http://www2.imm.dtu.dk/~db/fsi.pdf

[11] http://www2.imm.dtu.dk/~db/pipeline.pdf

[12] http://www2.imm.dtu.dk/~db/transport.pdf

[13] http://www.railwaydomain.org/

[14] As dealt with elsewhere (Appendix Sect. B.1.1, Pages 115–116) in these lecture notes: attributes of atomic or composite entities are (type and value) properties of entities (save those of being a composite entity and of such composite entities sub-entities). Atomic entities are atomic in that they have no sub-entities. Sub-entities of composite entities are proper entities.

[15] Till now, in these lecture notes, we have used"the same kind" of observer functions (ωB_i, ωC_j) for observing attributes (B_i) of atomic or composite entities and for observing sub-entities (C_j) of composite entities. In this section we shall distinguish between ωbserving αttributes ($\omega_\alpha B$) and ωbserving sub-ϵntities ($\omega_\epsilon C$). Maybe we shall have an opportunity to do so in a next version of these lecture notes.

[16] In this section we distinguish between $\iota\ell\ell$ustrations (formally marked with $\iota\ell\ell$s) and $\delta\epsilon\phi$initions (read: definitions, marked with $\delta\epsilon\phi$s). $\iota\ell\ell$ustrations are like schematic examples, but they are just that: rough-sketched generic examples. $\delta\epsilon\phi$initions are valid throughout these lecture notes.

Let A be (the type name of) a set of entities, let $\mathsf{B}_1, \ldots, \mathsf{B}_m$ be all the (distinct names of) types of constant valued attributes of A and let $\Sigma_1, \ldots, \Sigma_n$ be all the (distinct names of) types of variable valued attributes of A. We $\iota\ell\ell$ustrate these: 351

type
  $[\,\iota\ell\ell\,]$ A, B$_1$, ..., B$_m$, Σ_1, ..., Σ_n, C$_1$, ..., C$_k$

B.3.3 Sub-Entities 352

Let $\mathsf{C}_1, \ldots, \mathsf{C}_k$ be all the (distinct names of) types of sub-entities of A. We $\iota\ell\ell$ustrate these:

type
  $[\,\iota\ell\ell\,]$ C$_1$, ..., C$_k$

B.3.4 Attribute and Sub-Entity Observers 353

Let $\{\omega_\alpha \mathrm{B}_1, \ldots, \omega_\alpha \mathrm{B}_m\}$ be the corresponding set of all the constant valued observers of A, Let $\{\omega_\alpha \Sigma_1, \ldots, \omega_\alpha \Sigma_n\}$ be the corresponding set of all the variable valued observers of A and let $\{\omega_\epsilon \mathrm{C}_1, \ldots, \omega_\epsilon \mathrm{C}_k\}$ be the corresponding set of all the sub-entity observers of A. We $\iota\ell\ell$ustrate these:

value
  $[\,\iota\ell\ell\,]$ ω_αB$_1$: A $\to$ B$_1$, ..., ω_αB$_n$: A $\to$ B$_m$
  $[\,\iota\ell\ell\,]$ $\omega_\alpha\Sigma_1$: A $\to$ Σ_1, ..., $\omega_\alpha\Sigma_n$: A $\to$ Σ_n,
  $[\,\iota\ell\ell\,]$ ω_ϵC$_1$: A $\to$ C$_1$, ..., ω_ϵC$_k$: A $\to$ C$_2$

B.3.5 Attribute and Sub-entity Meta-Observers 354

Let $\mathcal{A}_{ttr_A}$ name the general type of a attribute observer function for sort A. Let $\mathcal{E}_{subs_A}$ name the general type of a sub-entity observer functions for sort A. We $\iota\ell\ell$ustrate, with respect to the above $\iota\ell\ell$ustrations, these general types:

type
  $[\,\iota\ell\ell\,]$ $\mathcal{A}_{ttr_A} = \omega_\alpha\mathrm{B}_1 \mid ... \mid \omega_\alpha\mathrm{B}_m \mid \omega_\alpha\Sigma_1 \mid ... \mid \omega_\alpha\Sigma_n$
  $[\,\iota\ell\ell\,]$ $\mathcal{E}_{subs_A} = \omega_\epsilon\mathrm{C}_1 \mid ... \mid \omega_\epsilon\mathrm{C}_k$

355

Let $\omega\mathcal{A}_{attr_A}$ denote the function which from a type (A) observes all it attribute observer functions. Let $\omega\mathcal{E}_{subs}$ denote the function which from a type observes all it possible sub-entity observer functions. We $\delta\epsilon\varphi$ne these:

value
  $[\,\delta\epsilon\phi\,]$ $\omega\mathcal{A}_{ttr_A}s$: A $\to$ $\mathcal{A}_{ttr_A}$**-set**
  $[\,\delta\epsilon\phi\,]$ $\omega\mathcal{E}_{subs_A}s$: A $\to$ $\mathcal{E}_{subs_A}$**-set**

B.3.6 Meta-Observer Properties 356

Let $\mathbb{A}_{ttr_A}$ $\iota\ell\ell$ustrate the set of all attribute observers for type A, and let $\mathbb{E}_{subs_A}$ $\iota\ell\ell$ustrate the set of all sub-entity observers for type A, then the two axioms $\iota\ell\ell_{attr}$ and $\iota\ell\ell_{subs}$ holds for the $\iota\ell\ell$ustrated type A and its observer functions:

value
$[\,\iota\ell\ell_{attr}\,]$ $\mathbb{A}_{ttr_A}$:$\mathcal{A}_{ttr_A}$-**set** $= \{\omega_\alpha \mathrm{B}_1,\ldots,\omega_\alpha \mathrm{B}_m,\omega_\alpha \Sigma_1,\ldots,\omega_\alpha \Sigma_n\}$,
$[\,\iota\ell\ell_{subs}\,]$ $\mathbb{E}_{subs_A}$:$\mathcal{E}_{subs_A}$-**set** $= \{\omega_\epsilon \mathrm{C}_1,\ldots,\omega_\epsilon \mathrm{C}_k\}$
axiom
$[\,\iota\ell\ell_{attr}\,]$ $\forall$ a:A • $\omega\mathcal{A}_{ttr_A}s(\mathrm{a}) = \mathbb{A}_{ttr_A} \wedge$
$[\,\iota\ell\ell_{subs}\,]$ $\forall$ a:A • $\omega\mathcal{E}_{subs_A}s(\mathrm{a}) = \mathbb{E}_{subs_A}$

B.3.7 Sort Value Equality 357

Now to register a possible change in but one attribute of A we meta-linguistically $\delta\epsilon\phi$ine the following equality operator:

value
$[\,\delta\epsilon\phi\,]$ $\simeq_{\mathcal{A}_{attr_A}}$: A×A $\rightarrow \mathcal{A}_{ttr_A} \rightarrow$ **Bool**
$[\,\delta\epsilon\phi\,]$ $\simeq_{\mathcal{A}_{attr_A}}(\mathrm{a}',\mathrm{a}'')(\omega_\alpha) \equiv$
$[\,\delta\epsilon\phi\,]$ $\quad\forall\ \omega F{:}\omega\mathcal{A}_{ttr_A}s(\mathrm{a}')\backslash\{\omega_\alpha\}\Rightarrow\omega F(\mathrm{a}')=\omega F(\mathrm{a}'') \wedge \forall\ \omega'_\epsilon{:}\mathcal{E}_{subs_A}\Rightarrow\omega'_\epsilon(\mathrm{a}')=\omega'_\epsilon(\mathrm{a}'')$
$[\,\delta\epsilon\phi\,]$ $\quad$**pre** $\omega\mathcal{A}_{ttr_A}s(\mathrm{a}') = \omega\mathcal{A}_{ttr_A}s(\mathrm{a}'')$

The $\simeq_{\omega\mathcal{A}_{attr}}$ 'equality' operator applies to two values a′,a″:A and an attribute observer function, $\omega\mathrm{B}_i$ (given as ω_α), and yields **true** if a′ and a″ have all but the same attribute
358 values except for attribute B_i, and have all exactly the same and equal sub-entities.

Example 50 – Equality of Hubs Modulo Hub States:

Please review Examples 2 on page 24 and 3 on page 25. In Example 3 on page 25 on Page 125, formula line item [17], a comparison is made between two values of a sort: $\omega\mathsf{H}\Sigma(\mathsf{h}')=(\lceil\rceil\{\mathsf{h}\sigma'|\mathsf{h}\sigma'{:}\mathsf{H}\Sigma\bullet\mathsf{h}\sigma'\in\ \omega\Omega(\mathsf{h})\backslash\{\mathsf{h}\sigma\}\})_{\overline{p}}\lceil\rceil_p\mathsf{h}\sigma$. We now redefine this comparion – which really does not capture all the value aspects of the compared hubs! 359

value
p:**Real**, **axiom** 0<p≤1, typically p$\simeq 1 - 10^{-7}$
$\overline{p}$:**Real**, **axiom** $\overline{p}$=1−p

[12] set_HΣ: H × HΣ → H
[13] set_HΣ(h,hσ) **as** h′
[14] $\quad$**pre** h$\sigma \in \omega$HΩ(h)
[15] $\quad$**post** $\simeq_{\omega_{\mathcal{A}_{attr_H}}}$(h,h′)($\omega$HΣ) $\wedge$

[17] $\omega\mathsf{H}\Sigma(\mathsf{h}')=(\lceil\rceil\{\mathsf{h}\sigma'|\mathsf{h}\sigma':\mathsf{H}\Sigma\bullet\mathsf{h}\sigma'\in\omega\Omega(\mathsf{h})\backslash\{\mathsf{h}\sigma\}\})_{\overline{p}}\lceil\rceil_{p}\mathsf{h}\sigma$

■ End of Example 50

B.4 Unique Entity Identifiers 360

In many domain and requirements modelling situations we make use of the concept of *unique entitiy identifiers*. For any type A for which we introduce unique identifiers of all a:A values we consider such unique identifiers as of sort AI[17]. The AI attribute shall be considered a constant-valued attribute.

[17] We may, in some immediate future, decide to instead of using the sort name AI using, for example, the sort name $\Im$A or $\Im_{\mathsf{A}}$.

C Mereology 361

C.1 Opening

C.1.1 Definition

By mereology we understand the study and knowledge about parts and wholes and the relationships between parts and between parts and holes.

C.1.2 Examples 362

Example 51 – Simple and Composite Net Entities:

We repeat some of the material from Example 1 on page 21.

- [1] A road, train, airlane (air traffic) or sea lane (shipping) net
- [2] consists, amongst other things, of hubs and links.

```
type
   [1] N
   [2] H, L
value
   [2] ωHs: N → H-set, ωLs: N → L-set,
```

We can consider nets as composite and, for the time being, hubs and links as simple.
. ■ End of Example 51

363 Example 51 illustrated that entities can be either atomic of composite. But also functions,
364 events and behaviours can be either atomic or composite.

Example 52 – Simple and Composite Net Functions:

- [3] With every link we associate a length.
- [4] A journey is a pair of a link and a continuation.
- [5] A continuation is either "nil" or is a journey.
- [6] Journies have lengths:
  - [6.1] the length of the link of the journey pair,
  - [6.2] and the length of the continuation – where a "nil" continuation has length 0.

365

```
type
    [3] LEN
    [4] Journey = L × C
    [5] C = ″nil″ | Journey
value
    [3] zero_LEN:LEN
    [3] ωLEN: L → LEN
    [6] length: Journey → LEN
    [6] length(l,c) ≡
    [6.1] let ll = ωLEN(l),
    [6.2]      cl = if c=″nil″ then zero_LEN else length(c) end in
    [6]   sum(ll,cl) end
    sum: LEN × LEN → LEN
```

366

Both the journey and continuation entities, *j* and *c* , and the *length* function are composite Both the link entities, *ll*, the ωLEN function are atomic. ■ End of Example 52

367

Example 53 – Simple and Composite Net Events:

- [7] The isolated crash of two vehicles, at time t, in a traffic, at a hub or along a link can be construed as a single atomic event.
- [8] The crash, within a few seconds $(t, t', t \sim t')$, in a traffic, of three or more vehicles,
  - [8.1] in a hub,
  - [8.2] or along a short segment of a link,

  can be considered a composite event.

We shall model this event by the predicates which holds of vehicles in a traffic at given times.

368

```
type
    TF = T → (V →m Pos)
    Pos == μatH(hi:HI) | μonL(πhi:HI,πli:LI,πf:F,πhi′:HI)
type
value
    [7] atomic_crash: V × V → TF → T → Bool
    [7] atomic_crash(v,v′)(tf)(t) ≡ (tf(t))(v)=(tf(t))(v′)
    [7] pre t ∈ 𝒟𝒪ℳ𝒜ℐ𝒩tf ∧ {v,v′}⊆dom(tf(t))∧v≠v′
```

```
[8] composite_crash: V-set → TF → (T×T) → Bool
[8] composite_crash(vs)(tf)(t,t′) ≡
[8.1]  ∃ hi:HI • card{v|v:V• ∈ vs∧(tf(t″))(v)=μatH(hi)∧t≤t″≤t′}≥3∨
[8.2]  ∃ hi′,hi″:HI,li:LI,fs:F-set •
[8.2]     fs={r..r′} where 0≤r≃r′≤1 ∧
[8.2]     card{(tf(t″))(v)=μonL(hi′,li,f,hi″)|v:V,f:F•v ∈ vs∧f ∈ fs∧t≤t″≤t′}≥3
[8]   pre {t,t′}⊆𝒟𝒪ℳ𝒜ℐ𝒩tf ∧ t∼t′ ∧ ∧ vs⊆dom(tf(t)) ∧ card vs≥3
```

■ End of Example 53

369 In the next, long example we consider a pipeline system (or either oil or gas pipes).

Example 54 – Simple and Composite Net Behaviours:

Pipeline Systems and Their Units

47. We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.

48. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.

49. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WElls and SKs.[18]

370

```
type
   47  N, PI, VA, PU, FO, JO, WE, SK
   48  U = Π | V | P | F | J | S| W
   48  Π == mkΠ(pi:PI)
   48  V == mkV(va:VA)
   48  P == mkP(pu:PU)
   48  F == mkF(fo:FO)
   48  J == mkJ(jo:JO)
   48  W == mkW(we:WE)
   48  S == mkS(sk:SK)
```

Unit Identifiers and Unit Type Predicates 371

50. We associate with each unit a unique identifier, $ui : UI$.

51. From a unit we can observe its unique identifier.

52. From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

```
type
   50  UI
value
   51  obs_UI: U → UI
   52  is_Π: U → Bool, is_V: U → Bool, ..., is_J: U → Bool
       is_Π(u) ≡ case u of mkPI(_) → true, _ → false end
       is_V(u) ≡ case u of mkV(_) → true, _ → false end
        ...
       is_S(u) ≡ case u of mkS(_) → true, _ → false end
```

Unit Connections 372

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from "the other side".

53. With a pipe, a valve and a pump we associate exactly one input and one output connection.

54. With a fork we associate a maximum number of output connections, m, larger than one.

55. With a join we associate a maximum number of input connections, m, larger than one.

56. With a well we associate zero input connections and exactly one output connection.

57. With a sink we associate exactly one input connection and zero output connections.

373

```
value
   53 obs_InCs,obs_OutCs: Π|V|P → {|1:Nat|}
   54 obs_inCs: F → {|1:Nat|}, obs_outCs: F → Nat
   55 obs_inCs: J → Nat, obs_outCs: J → {|1:Nat|}
   56 obs_inCs: W → {|0:Nat|}, obs_outCs: W → {|1:Nat|}
   57 obs_inCs: S → {|1:Nat|}, obs_outCs: S → {|0:Nat|}
axiom
   54 ∀ f:F • obs_outCs(f) ≥ 2
   55 ∀ j:J • obs_inCs(j) ≥ 2
```

374

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed. If a fork is output-connected to n units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: "the other way around".

Net Observers and Unit Connections

375

58. From a net one can observe all its units.

59. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:

    a) Wells can be connected to zero or one output unit — a pump.

    b) Sinks can be connected to zero or one input unit — a pump or a valve.

    c) Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.

    d) Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.

    e) Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

376

```
value
   58  obs_Us: N → U-set
   59  obs_cUIs: U → UI-set × UI-set
       wf_Conns: U → Bool
       wf_Conns(u) ≡
           let (iuis,ouis) = obs_cUIs(u) in iuis ∩ ouis = {} ∧
           case u of
   59a  mkW(_) → card iuis ∈ {0} ∧ card ouis ∈ {0,1},
   59b  mkS(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0},
   59c  mkΠ(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
   59c  mkV(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
   59c  mkP(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
   59d  mkF(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0}∪{2..obs_inCs(j)},
   59e  mkJ(_) → card iuis ∈ {0}∪{2..obs_inCs(j)} ∧ card ouis ∈ {0,1}
           end end
```

Well-formed Nets, Actual Connections 377

60. The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

```
axiom
    60  ∀ n:N,u:U • u ∈ obs_Us(n) ⇒
    60    let (iuis,ouis) = obs_cUIs(u) in
    60    ∀ ui:UI • ui ∈ iuis ∪ ouis ⇒
    60      ∃ u′:U • u′ ∈ obs_Us(n) ∧ u′≠u ∧ obs_UI(u′)=ui end
```

Well-formed Nets, No Circular Nets 378

61. By a route we shall understand a sequence of units.

62. Units form routes of the net.

```
type
    61  R = UI^ω
value
    62  routes: N → R-infset
    62  routes(n) ≡
    62    let us = obs_Us(n) in
    62    let rs = {⟨u⟩|u:U•u ∈ us} ∪ {r^r′|r,r′:R• {r,r′}⊆rs∧adj(r,r′)} in
    62    rs end end
```

379

63. A route of length two or more can be decomposed into two routes

64. such that the least unit of the first route "connects" to the first unit of the second route.

```
value
    63    adj: R × R → Bool
    63    adj(fr,lr) ≡
    63      let (lu,fu)=(fr(len fr),hd lr) in
    64      let (lui,fui)=(obs_UI(lu),obs_UI(fu)) in
    64      let ((_,luis),(fuis,_))=(obs_cUIs(lu),obs_cUIs(fu)) in
    64      lui ∈ fuis ∧ fui ∈ luis end end end
```

65. No route must be circular, that is, the net must be acyclic.

```
value
    65 acyclic: N → Bool
    65   let rs = routes(n) in
    65   ∼∃ r:R•r ∈ rs⇒∃ i,j:Nat•{i,j}⊆inds r∧i≠j∧r(i)=r(j) end
```

Pipeline Processes

380

We now add connectors to our model:

66. From an oil pipeline system one can observe units and connectors.
67. Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.
68. Units and connectors have unique identifiers.
69. From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

381

```
type
66 OPLS, U, K
68 UI, KI
value
66 obs_Us: OPLS → U-set, obs_Ks: OPLS → K-set
67 is_WeU, is_PiU, is_PuU, is_VaU, is_JoU, is_FoU, is_SiU: U → Bool [mutually exclusive]
68 obs_UI: U → UI, obs_KI: K → KI
69 obs_UIp: K → (UI|{nil}) × (UI|{nil})
```

382

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

383

70. There is given an oil pipeline system, opls.
71. To every unit we associate a CSP behaviour.
72. Units are indexed by their unique unit identifiers.
73. To every connector we associate a CSP channel.

    Channels are indexed by their unique "k"onnector identifiers.
74. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes,

represented by u.

75. Channels, in this model, have no state.

76. Unit behaviours communicate with neighbouring units — those with which they are connected.

77. Unit functions, $\mathcal{U}_i$, change the unit state.

78. The pipeline system is now the parallel composition of all the unit behaviours.

384

Editorial Remark: Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise. The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink. The literal **Unit**, in a function signature, before the → "announces" that the function takes no argument.[19] The literal **Unit**, in a function signature, after the → "announces", as used here, that the function never terminates.

385

```
value
70  opls:OPLS
channel
73  {ch[ ki ]|k:KI,k:K•k ∈ obs_Ks(opls)∧ki=obs_KI(k)} M
value
78  pipeline_system: Unit → Unit
78  pipeline_system() ≡
71     ∥ {unit(ui)(u)|u:U•u ∈ obs_Us(opls)∧ui=obs_UI(u)}

72  unit: ui:UI → U →
76        in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
76                    let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  Unit
74  unit(ui)(u) ≡ let u′ = $\mathcal{U}_i$(ui)(u) in unit(ui)(u′) end

77  $\mathcal{U}_i$: ui:UI → U →
77        in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
77                    let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  U
```

■ End of Example 54

C.1.3 Discussion

386

In this Appendix section, Sect. C, we shall mainly cover atomic and composite entities.

C.2 A Conceptual Model of Composite Entities 387

C.2.1 Systems, Assemblies, Units

We speak of systems as assemblies. From an assembly we can immediately observe a set of parts. Parts are either assemblies or units. We do not further define what assemblies and units are.

```
type
   S = A, A, U, P = A | U
value
   obs_Ps: (S|A) → P-set
```

Parts observed from an assembly are said to be immediately embedded in, that is, within, that assembly. Two or more different parts of an assembly are said to be immediately adjacent to one another. 388

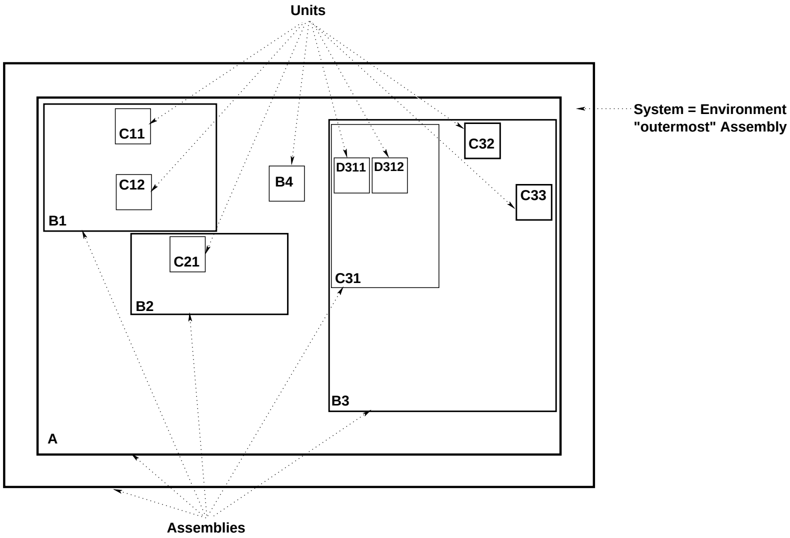


Figure 10: Assemblies and Units “embedded” in an Environment

A system includes its environment. And we do not worry, so far, about the semiotics of all this ! 389

Embeddedness and adjacency generalise to transitive relations.

Given obs_Ps we can define a function, xtr_Ps, which applies to an assembly a and which extracts all parts embedded in a and including a. The functions obs_Ps and xtr_Ps define the meaning of embeddedness.

```
value
   xtr_Ps: (S|A) → P-set
   xtr_Ps(a) ≡
      let ps = {a} ∪ obs_Ps(a) in ps ∪ union{xtr_Ps(a′)|a′:A•a′ ∈ ps} end
```

union is the distributed union operator. Parts have unique identifiers. All parts observable from a system are distinct. 390

```
type
   AUI
value
   obs_AUI: P → AUI
axiom
   ∀ a:A •
      let ps = obs_Ps(a) in
      ∀ p′,p″:P • {p′,p″}⊆ps ∧ p′≠p″ ⇒ obs_AUI(p′)≠obs_AUI(p″) ∧
      ∀ a′,a″:A • {a′,a″}⊆ps ∧ a′≠a″ ⇒ xtr_Ps(a′)∩ xtr_Ps(a″)={} end
```

C.2.2 ‘Adjacency’ and ‘Within’ Relations 391

Two parts, p,p′, are said to be *immediately next to*, i.e., i_next_to(p,p′)(a), one another in an assembly a if there exists an assembly, *a′* equal to or embedded in *a* such that p and p′ are observable in that assembly a′.

```
value
   i_next_to: P × P → A ⥲ Bool, pre i_next_to(p,p′)(a): p≠p′
   i_next_to(p,p′)(a) ≡ ∃ a′:A • a′=a ∨ a′ ∈ xtr_Ps(a) • {p,p′}⊆obs_Ps(a′)
```

392 One part, p, is said to be *immediately within* another part, p′in an assembly a if there exists an assembly, a′ equal to or embedded in a such that p is observable in a′.

```
value
   i_within: P × P → A ⥲ Bool
   i_within(p,p′)(a) ≡
      ∃ a′:A •  (a=a′ ∨ a′ ∈ xtr_Ps(a)) • p′=a′ ∧ p ∈ obs_Ps(a′)
```

393 We can generalise the immediate ‘within’ property. A part, p, is (transitively) within a part p′, within(p,p′)(a), of an assembly, a, either if p, is immediately within p′ of that assembly, a, or if there exists a (proper) part p″ of p′ such that within(p″,p)(a).

```
value
   within: P × P → A ⥲ Bool
   within(p,p′)(a) ≡
      i_within(p,p′)(a) ∨ ∃ p″:P • p″ ∈ obs_Ps(p) ∧ within(p″,p′)(a)
```

394 The function within can be defined, alternatively, using xtr_Ps and i_within instead of obs_Ps and within :

```
value
    within′: P × P → A ~→ Bool
    within′(p,p′)(a) ≡
        i_within(p,p′)(a) ∨ ∃ p″:P • p″ ∈ xtr_Ps(p) ∧ i_within(p″,p′)(a)
```

lemma: within ≡ within′

395 We can generalise the immediate ‘next to’ property. Two parts, p, p′ of an assembly, a, are adjacent if they are either ‘next to’ one another or if there are two parts p_o, p'_o such that p, p′ are embedded in respectively p_o and p'_o and such that p_o, p'_o are immediately next to one another.

```
value
    adjacent: P × P → A ~→ Bool
    adjacent(p,p′)(a) ≡
        i_next_to(p,p′)(a) ∨
        ∃ p″,p‴:P • {p″,p‴}⊆xtr_Ps(a) ∧ i_next_to(p″,p‴)(a) ∧
            ((p=p″)∨within(p,p″)(a)) ∧ ((p′=p‴)∨within(p′,p‴)(a))
```

C.2.3 Mereology, Part I 396

So far we have built a *ground mereology* model, $\mathcal{M}_{\mathcal{G}\text{round}}$. Let $\sqsubseteq$ denote *parthood, x is part of y*, $x \sqsubseteq y$.

$$\forall x(x \sqsubseteq x)^{20} \tag{1}$$
$$\forall x, y(x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow (x = y) \tag{2}$$
$$\forall x, y, z(x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z) \tag{3}$$

Let $\sqsubset$ denote *proper parthood, x is part of y*, $x \sqsubset y$. Formula 4 defines $x \sqsubset y$. Equivalence 5 can be proven to hold.

$$\forall x \sqsubset y \quad =_{\text{def}} \quad x(x \sqsubseteq y) \wedge \neg(x = y) \tag{4}$$
$$\forall\forall x, y(x \sqsubseteq y) \quad \Leftrightarrow \quad (x \sqsubset y) \vee (x = y) \tag{5}$$

397 The *proper part* $(x \sqsubset y)$ relation is a strict partial ordering:

$$\forall x \neg(x \sqsubset x) \tag{6}$$
$$\forall x, y(x \sqsubset y) \Rightarrow \neg(y \sqsubset x) \tag{7}$$
$$\forall x, y, z(x \sqsubset y) \wedge (y \sqsubset z) \Rightarrow (x \sqsubset z) \tag{8}$$

[20] Our notation now is not RSL but some conventional first-order predicate logic notation.

Overlap, $\bullet$, is also a relation of parts: Two individuals overlap if they have parts in common:

$$x \bullet y \;=_{\text{def}}\; \exists z(z \sqsubset x) \wedge (z \sqsubset y) \tag{9}$$
$$\forall x(x \bullet x) \tag{10}$$
$$\forall x, y(x \bullet y) \Rightarrow (y \bullet x) \tag{11}$$

Proper overlap, $\circ$, can be defined: 398

$$x \circ y \;=_{\text{def}}\; (x \bullet x) \wedge \neg(x \sqsubseteq y) \wedge \neg(y \sqsubseteq x) \tag{12}$$

Whereas Formulas (1-11) holds of the model of mereology we have shown so far, Formula (12) does not. In the next section we shall repair that situation.

The *proper part* relation, $\sqsubset$, reflects the *within* relation. The *disjoint* relation, $\oint$, reflects the *adjacency* relation.

$$x \oint y \;=_{\text{def}}\; \neg(x \bullet y) \tag{13}$$

Disjointness is symmetric: 399

$$\forall x, y(x \oint y) \Rightarrow (y \oint x) \tag{14}$$

The *weak supplementation* relation, Formula 15, expresses that if y is a proper part of x then there exists a part z such that z is a proper part of x and z and y are disjoint That is, whenever an individual has one proper part then it has more than one.

$$\forall x, y(y \sqsubset x) \;\Rightarrow\; \exists z(z \sqsubset x) \wedge (z \oint y) \tag{15}$$

Formulas 1–3 and 15 together determine the *minimal mereology*, $\mathcal{M}_{\mathcal{M}\text{inimal}}$. Formula 15 400
does not hold of the model of mereology we have shown so far..

Formula 15 expresses that whenever an individual has one proper part then it has more
than one. We mentioned there, Page 137, that we would comment on the fact that our
model appears to allow that assemblies may have just one proper part. We now do so. We 401
shall still allow assemblies to have just one proper part — in the sense of a sub-assembly
or a unit — but we shall interpret the fact that an assembly always have at least one
attribute. Therefore we shall "generously" interpret the set of attributes of an assembly
to constitute a part. In Sect. C.4 we shall see how attributes of both units and assemblies 402
of the interpreted mereology contribute to the state components of the unit and assembly
processes.

C.2.4 Connectors 403

So far we have only covered notions of parts being next to other parts or within one another. We shall now add to this a rather general notion of parts being otherwise related. 404 That notion is one of connectors. Connectors provide for connections between parts. A connector is an ability be be connected. A connection is the actual fulfillment of that ability. Connections are relations between pairs of parts. Connections "cut across" the "classical" *parts being part of the (or a) whole* and *parts being related by embeddedness* 405 *or adjacency*.

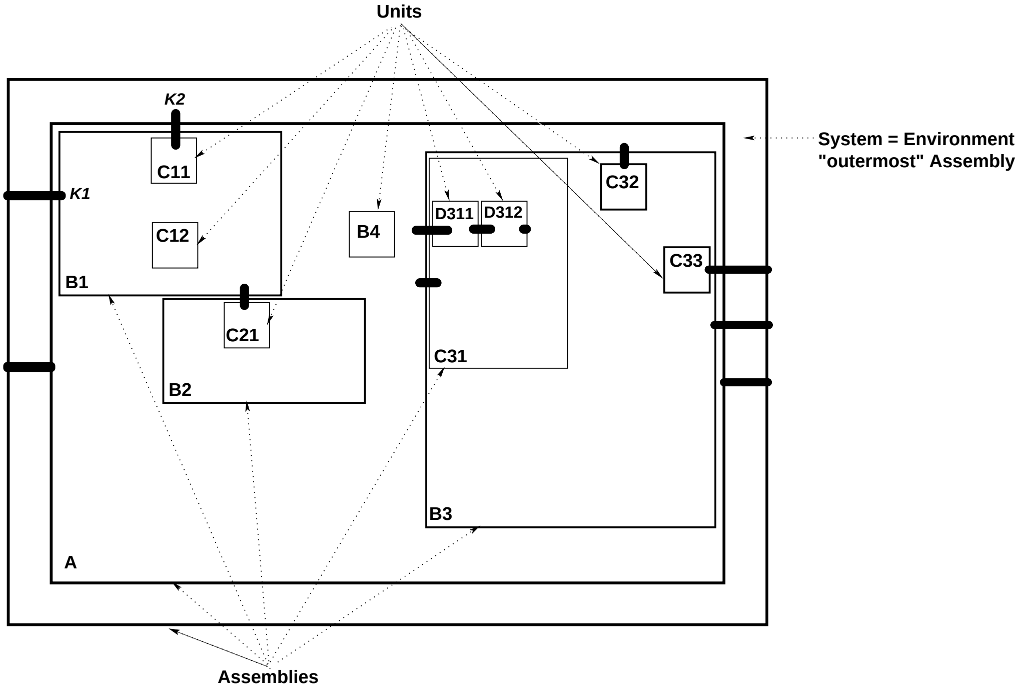


Figure 11: Assembly and Unit Connectors: Internal and External

406 For now, we do not "ask" for the meaning of connectors !

Figure 11 "adds" connectors to Fig. 10 on page 134. The idea is that connectors allow an assembly to be connected to any embedded part, and allow two adjacent parts to be connected.

In Fig. 11 the environment is connected, by *K2*, (without, as we shall later see, interfering with assemblies A and B1), to part C11; the "external world" is connected, by K1, 407 to B1; etcetera. Later we shall discuss more general forms of connectors.

From a system we can observe all its connectors. From a connector we can observe its unique connector identifier and the set of part identifiers of the parts that the connector connects. All part identifiers of system connectors identify parts of the system. All 408 observable connector identifiers of parts identify connectors of the system.

type

```
    K
value
    obs_Ks: S → K-set
    obs_KI: K → KI
    obs_Is: K → AUI-set
    obs_KIs: P → KI-set
axiom
    ∀ k:K • card obs_Is(k)=2,
    ∀ s:S,k:K • k ∈ obs_Ks(s) ⇒
        ∃ p:P • p ∈ xtr_Ps(s) ⇒ obs_AUI(p) ∈ obs_Is(k),
    ∀ s:S,p:P • ∀ ki:KI • ki ∈ obs_KIs(p) ⇒
        ∃! k:K • k ∈ obs_Ks(s) ∧ ki=obs_KI(k)
```

409 This model allows for a rather “free-wheeling” notion of connectors one that allows internal connectors to “cut across” embedded and adjacent parts; and one that allows external connectors to “penetrate” from an outside to any embedded part.

We need define an auxiliary function. xtr∀KIs(p) applies to a system and yields all its connector identifiers.

```
value
    xtr∀KIs: S → KI-set
    xtr∀Ks(s) ≡ {obs_KI(k)|k:K•k ∈ obs_Ks(s)}
```

C.2.5 Mereology, Part II 410

(See Sect. C.2.3 (Page 136) for Mereology, Part I.)

We shall interpret connections as follows: A connection between parts p_i and p_j that enjoy a p_i adjacent to p_j relationship, means $p_i \circ p_j$, that is, although parts p_i and p_j are adjacent they do *share* “something”, i.e., have something *in common*. What that “something” is we shall comment on in Sect. C.4.3. A connection between parts p_i and p_j that enjoy a p_i within p_j relationship, does not add other meaning than commented upon in Sect. C.4.3 on page 151. 411

With the above interpretation we may arrive at the following, perhaps somewhat “awkward-looking” case: a connection connects two adjacent parts p_i and p_j where part p_i is within part p_{i_o} and part p_j is within part p_{j_o} where parts p_{i_o} and p_{j_o} are adjacent but not otherwise connected. How are we to explain that ! Since we have not otherwise interpreted the meaning of parts, we can just postulate that “so it is” ! We shall, in Sect. C.4.3 on page 151, give a more satisfactory explanation. 412

On Pages 136–137 we introduced the following operators: $\sqsubseteq$, $\sqsubset$, $\bullet$, $\circ$, and $\oint$ In some of the mereology literature [30, 31, 28] these operators are symbolised with caligraphic letters: $\sqsubseteq$: $\mathcal{P}$: part, $\sqsubset$: $\mathcal{PP}$: proper part, $\bullet : \mathcal{O}$: overlap and $\oint : \mathcal{U}$: underlap.

C.2.6 Discussion 413

Summary: This ends our first model of a concept of mereology. The parts are those of assemblies and units. The relations between parts and the whole are, on one hand, those of embeddedness i.e. within, and adjacency, i.e., adjacent, and on the other hand, those expressed by connectors: relations between arbitrary parts and between arbitrary parts and the exterior. 414

Extensions: A number of extensions are possible: one can add "mobile" parts and "free" connectors, and one can further add operations that allow such mobile parts to move from one assembly to another along routes of connectors. Free connectors and mobility assumes static versus dynamic parts and connectors: a free connector is one which allows a mobile part to be connected to another part, fixed or mobile; and the potentiality of a move of a mobile part introduces a further dimension of dynamics of a mereology. 415

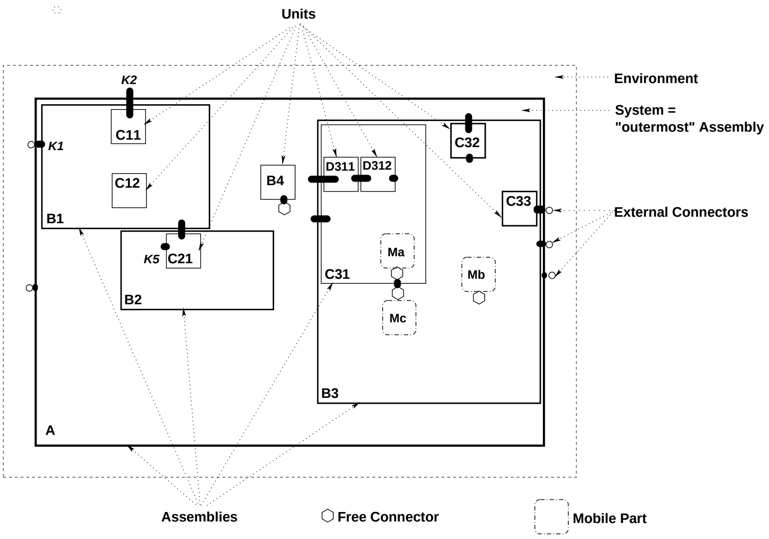


Figure 12: Mobile Parts and Free Connectors

416

Comments: We shall leave the modelling of free connectors and mobile parts to another time. Suffice it now to indicate that the mereology model given so far is relevant: that it applies to a somewhat wide range of application domain structures, and that it thus affords a uniform treatment of proper formal models of these application domain structures.

C.3 Functions and Events 417

418

Example 55 – Pipeline Transport Functions and Events:

We need introduce a number of auxiliary concepts in order to show examples of atomic and composite functions and events.

Well-formed Nets, Special Pairs, wfN_SP

419

79. We define a “special-pairs” well-formedness function.

    a) Fork outputs are output-connected to valves.

    b) Join inputs are input-connected to valves.

    c) Wells are output-connected to pumps.

    d) Sinks are input-connected to either pumps or valves.

420

```
value
    79  wfN_SP: N → Bool
    79  wfN_SP(n) ≡
    79    ∀ r:R • r ∈ routes(n) in
    79      ∀ i:Nat • {i,i+1}⊆inds r ⇒
    79        case r(i) of ∧
    79a         mkF(_) → ∀ u:U•adj(⟨r(i)⟩,⟨u⟩) ⇒ is_V(u),_→true end ∧
    79        case r(i+1) of
    79b         mkJ(_) → ∀ u:U•adj(⟨u⟩,⟨r(i)⟩) ⇒ is_V(u),_→true end ∧
    79        case r(1) of
    79c         mkW(_) → is_P(r(2)),_→true end ∧
    79        case r(len r) of
    79d         mkS(_) → is_P(r(len r−1))∨is_V(r(len r−1)),_→true end
```

The **true** clauses may be negated by other **case** distinctions’ is_V or is_V clauses.

Special Routes, I

421

80. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.

81. A simple pump-pump route is a pump-pump route with no forks and joins.

82. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.

83. A simple pump-valve route is a pump-valve route with no forks and joins.

84. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.

85. A simple valve-pump route is a valve-pump route with no forks and joins.

86. A valve-valve route is a route of length two or more whose first and last units are valves

and whose intermediate units are pipes or forks or joins.

87. A simple valve-valve route is a valve-valve route with no forks and joins.

422

```
value
    80-87  ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
               pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n≥2

    80  ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
    81  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
    82  pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
    83  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
    84  vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
    85  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
    86  vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
    87  sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

    is_πfjr,is_πr: R → Bool
    is_πfjr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)∨is_F(u)∨is_J(u)
    is_πr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)
```

Special Routes, II

423

Given a unit of a route,

88. if they exist (∃),

89. find the nearest pump or valve unit,

90. “upstream” and

91. “downstream” from the given unit.

424

```
value
    88 ∃UpPoV: U × R → Bool
    88 ∃DoPoV: U × R → Bool
    90 find_UpPoV: U × R →̃ (P|V), pre find_UpPoV(u,r): ∃UpPoV(u,r)
    91 find_DoPoV: U × R →̃ (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
    88 ∃UpPoV(u,r) ≡
    88   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
```

```
88 ∃DoPoV(u,r) ≡
88    ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
90 find_UpPoV(u,r) ≡
90    let i,j:Nat•{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) in r(i) end
91 find_DoPoV(u,r) ≡
91    let i,j:Nat•{i,j}⊆indsr∧i≤j∧u=r(i)∧{is_V|is_P}(r(j)) in r(j) end
```

State Attributes of Pipeline Units 425

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units. 426

92. Oil flow, $\phi : \Phi$, is measured in volume per time unit.
93. Pumps are either pumping or not pumping, and if not pumping they are closed.
94. Valves are either open or closed.
95. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.
96. At any time any unit is sustaining a current input flow of oil (at its input(s)).
97. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

427

```
type
    92  Φ
    93  PΣ == pumping | not_pumping
    93  VΣ == open | closed
value
          −,+: Φ × Φ → Φ, <,=,>: Φ × Φ → Bool
    93     obs_PΣ: P → PΣ
    94     obs_VΣ: V → VΣ
    95–97  obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
    is_Open: U → Bool
       case u of
          mkΠ(_)→true,mkF(_)→true,mkJ(_)→true,mkW(_)→true,mkS(_)→true,
          mkP(_)→obs_PΣ(u)=pumping,
          mkV(_)→obs_VΣ(u)=open
```

```
        end
    acceptable_LeakΦ, excessive_LeakΦ: U → Φ
axiom
    ∀ u:U • excess_LeakΦ(u) > accept_LeakΦ(u)
```

Flow Laws

428

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws.

98. When, in Item 96, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

429

```
value
    98    obs_inΦ: U → Φ
    98    obs_inΦ(u) ≡ obs_CurrΦ(u)
    98    obs_outΦ: U → Φ
law:
    98    ∀ u:U • obs_outΦ(u) = obs_CurrΦ(u)−obs_LeakΦ(u)
```

430

99. Two connected units enjoy the following flow relation:
    a) If
        i. two pipes, or
        ii. a pipe and a valve, or
        iii. a valve and a pipe, or
        iv. a valve and a valve, or
        v. a pipe and a pump, or
        vi. a pump and a pipe, or
        vii. a pump and a pump, or
        viii. a pump and a valve, or
        ix. a valve and a pump

       are immediately connected
    b) then
        i. the current flow out of the first unit's connection to the second unit
        ii. equals the current flow into the second unit's connection to the first unit

431

law:

99a $\forall\ u,u':U \bullet \{is\_\Pi,is\_V,is\_P,is\_W\}(u'|u'') \wedge adj(\langle u\rangle,\langle u'\rangle)$
99a $\quad is\_\Pi(u) \vee is\_V(u) \vee is\_P(u) \vee is\_W(u) \wedge$
99a $\quad is\_\Pi(u') \vee is\_V(u') \vee is\_P(u') \vee is\_S(u')$
99b $\quad \Rightarrow obs\_out\Phi(u)=obs\_in\Phi(u')$

432

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

Possibly Desirable Properties

433

100. Let r be a route of length two or more, whose first unit is a pump, *p*, whose last unit is a valve, *v* and whose intermediate units are all pipes: if the pump, *p* is pumping, then we expect the valve, *v*, to be open.

101. Let r be a route of length two or more, whose first unit is a pump, *p*, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, *p* is pumping, then we expect pump p'', to also be pumping.

102. Let r be a route of length two or more, whose first unit is a valve, *v*, whose last unit is a pump, *p* and whose intermediate units are all pipes: if the valve, *v* is closed, then we expect pump *p*, to not be pumping.

103. Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

434

desirable properties:

100 $\forall\ r:R \bullet spvr(r) \wedge$
100 $\quad$ **spvr_prop(r):** $obs\_P\Sigma(\textbf{hd}\ r)=pumping \Rightarrow obs\_P\Sigma(r(\textbf{len}\ r))=open$

101 $\forall\ r:R \bullet sppr(r) \wedge$
101 $\quad$ **sppr_prop(r):** $obs\_P\Sigma(\textbf{hd}\ r)=pumping \Rightarrow obs\_P\Sigma(r(\textbf{len}\ r))=pumping$

102 $\forall\ r:R \bullet svpr(r) \wedge$
102 $\quad$ **svpr_prop(r):** $obs\_P\Sigma(\textbf{hd}\ r)=open \Rightarrow obs\_P\Sigma(r(\textbf{len}\ r))=pumping$

```
103 ∀ r:R • svvr(r) ∧
103   svvr_prop(r): obs_PΣ(hd r)=obs_PΣ(r(len r))
```

Pipeline Actions

435

- ***Simple Pump and Valve Actions***

104. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.

105. Valves may be set to be open or to be closed irrespective of the valve state.

106. In setting or resetting a pump or a valve a desirable property may be lost.

```
value
    104  pump_to_pump, pump_to_not_pump: P → N → N
    105  valve_to_open, valve_to_close: V → N → N
```

436

```
value
    104  pump_to_pump(p)(n) as n′
    104    pre p ∈ obs_Us(n)
    104    post let p′:P•obs_UI(p)=obs_UI(p′) in
    104            obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) end
    104  pump_to_not_pump(p)(n) as n′
    104    pre p ∈ obs_Us(n)
    104    post let p′:P•obs_UI(p)=obs_UI(p′) in
    104            obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) end
    105  valve_to_open(v)(n) as n′
    104    pre v ∈ obs_Us(n)
    105    post let v′:V•obs_UI(v)=obs_UI(v′) in
    104            obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) end
    105  valve_to_close(v)(n) as n′
    104    pre v ∈ obs_Us(n)
    105    post let v′:V•obs_UI(v)=obs_UI(v′) in
    104            obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) end
```

437

```
value
   else_equal: (N×N) → (U×U) → Bool
```

```
    else_equal(n,n′)(u,u′) ≡
        obs_UI(u)=obs_UI(u′)
     ∧ u ∈ obs_Us(n)∧u′ ∈ obs_Us(n′)
     ∧ omit_Σ(u)=omit_Σ(u′)
     ∧ obs_Us(n)\{u}=obs_Us(n)\{u′}
     ∧ ∀ u″:U•u″ ∈ obs_Us(n)\{u} ≡ u″ ∈ obs_Us(n′)\{u′}

    omit_Σ: U → U_no_state −−− ″magic″ function

    =: U_no_state × U_no_state → Bool
axiom
    ∀ u,u′:U•omit_Σ(u)=omit_Σ(u′) ≡ obs_UI(u)=obs_UI(u′)
```

Events

438

- ***Unit Handling Events***

107. Let n be any acyclic net.

107. If there exists p, p', v, v', pairs of distinct pumps and distinct valves of the net,

107. and if there exists a route, r, of length two or more of the net such that

108. all units, u, of the route, except its first and last unit, are pipes, then

109. if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;

110. if the route "spans" between p and v and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;

111. if the route "spans" between v and p and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and

112. if the route "spans" between v and v' and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

439

```
events:
    107  ∀ n:N • acyclic(n) ∧
    107    ∃ p,p′:P,v,v′:V • {p,p′,v,v′}⊆obs_Us(n)⇒
    107      ∧ ∃ r:R • routes(n) ∧
    108            ∀ u:U • u ∈ elems(r)\{hd r,r(len r)} ⇒ is_Π(i) ⇒
```

```
109            p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
110            p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
111            v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
112            v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

• *Foreseeable Accident Events* 440

A number of foreseeable accidents may occur.

113. A unit ceases to function, that is,
    a) a unit is clogged,
    b) a valve does not open or close,
    c) a pump does not pump or stop pumping.
114. A unit gives rise to excessive leakage.
115. A well becomes empty or a sunk becomes full.
116. A unit, or a connected net of units gets on fire.
117. Or a number of other such “accident”.

Well-formed Operational Nets 441

118. A well-formed operational net
119. is a well-formed net
    a) with at least one well, w, and at least one sink, s,
    b) and such that there is a route in the net between w and s.

```
value
    118  wf_OpN: N → Bool
    118  wf_OpN(n) ≡
    119    satisfies axiom 60 on page 131 ∧ acyclic(n): Item 65 on page 131  ∧
    119    wfN_SP(n): Item 79 on page 141 ∧
    119    satisfies flow laws, 98 on page 144 and 99 on page 144 ∧
    119a     ∃ w:W,s:S • {w,s}⊆obs_Us(n) ⇒
    119b       ∃ r:R• ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)
```

Orderly Action Sequences 442

• *Initial Operational Net*

120. Let us assume a notion of an initial operational net.

121. Its pump and valve units are in the following states

    a) all pumps are not_pumping, and

    b) all valves are closed.

value

```
120   initial_OpN: N → Bool
121   initial_OpN(n) ≡ wf_OpN(n) ∧
121a     ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
121b     ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed
```

Oil Pipeline Preparation and Engagement 443

122. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.

    a) We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.

    b) Now, an orderly action sequence for engaging route r is to “work backwards”, from s to w

    c) setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties. Finally, setting the pump adjacent to the (preceding) well starts the system. 444

value

```
122    prepare_and_engage: W × S → N $\overset{\sim}{\to}$ N
122    prepare_and_engage(w,s)(n) ≡
122a      let r:R • ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n) in
122b      action_sequence(⟨w⟩⌢r⌢⟨s⟩)(len⟨w⟩⌢r⌢⟨s⟩)(n) end
122       pre ∃ r:R • ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)

122c   action_sequence: R → Nat → N → N
122c   action_sequence(r)(i)(n) ≡
122c      if i=1 then n else
122c      case r(i) of
122c         mkV(_) → action_sequence(r)(i−1)(valve_to_open(r(i))(n)),
122c    mkP(_) → action_sequence(r)(i−1)(pump_to_pump(r(i))(n)),
122c    _ → action_sequence(r)(i−1)(n)
122c  end end
```

Emergency Actions 445

123. If a unit starts leaking excessive oil
    - a) then nearest up-stream valve(s) must be closed,
    - b) and any pumps in-between this (these) valves and the leaking unit must be set to not_pumping — following an orderly sequence.

124. If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold
    - a) then — a ha !
    - b) Left as an exercise.

■ End of Example 55

C.4 Behaviours: A Semantic Model of a Class of Mereologies 446

The model of mereology presented in Sect. C.2 (Pages 134–122) focused on the following simple entities (i) the assemblies, (ii) the units and (iii) the connectors. To assemblies and units we associate CSP processes, and to connectors we associate a CSP channels, one-by-one [56]. The connectors form the mereological attributes of the model.

C.4.1 Channels 447

The CSP channels, are each "anchored" in two parts: if a part is a unit then in "its corresponding" unit process, and if a part is an assembly then in "its corresponding" assembly process. From a system assembly we can extract all connector identifiers. They become indexes into an array of channels. Each of the connector channel identifiers is mentioned in exactly two unit or assembly processes. 448

```
value
   s:S
   kis:KI-set = xtr∀KIs(s)
type
   ChMap = AUI →m KI-set
value
   cm:ChMap = [ obs_AUI(p)↦obs_KIs(p)|p:P•p ∈ xtr_Ps(s) ]
channel
```

```
    ch[ i|i:KI•i ∈ kis ] MSG
```

C.4.2 Process Definitions 449

```
value
    system: S → Process
    system(s) ≡ assembly(s)

    assembly: a:A→in,out {ch[ cm(i) ]|i:KI•i ∈ cm(obs_AUI(a))} process
    assembly(a) ≡
        $\mathcal{M}_{\mathcal{A}}$(a)(obs_AΣ(a)) ∥
        ∥ {assembly(a′)|a′:A•a′ ∈ obs_Ps(a)} ∥
        ∥ {unit(u)|u:U•u ∈ obs_Ps(a)}
    obs_AΣ: A → AΣ

    $\mathcal{M}_{\mathcal{A}}$: a:A→AΣ→in,out {ch[ cm(i) ]|i:KI•i ∈ cm(obs_AUI(a))} process
    $\mathcal{M}_{\mathcal{A}}$(a)(a$\sigma$) ≡ $\mathcal{M}_{\mathcal{A}}$(a)(A$\mathcal{F}$(a)(a$\sigma$))

    A$\mathcal{F}$: a:A → AΣ → in,out {ch[ em(i) ]|i:KI•i ∈
    cm(obs_AUI(a))}×AΣ
```

450

```
    unit: u:U → in,out {ch[ cm(i) ]|i:KI•i ∈ cm(obs_UI(u))} process
    unit(u) ≡ $\mathcal{M}_{\mathcal{U}}$(u)(obs_UΣ(u))
    obs_UΣ: U → UΣ

    $\mathcal{M}_{\mathcal{U}}$: u:U → UΣ → in,out {ch[ cm(i) ]|i:KI•i ∈ cm(obs_UI(u))} process
    $\mathcal{M}_{\mathcal{U}}$(u)(u$\sigma$) ≡ $\mathcal{M}_{\mathcal{U}}$(u)(U$\mathcal{F}$(u)(u$\sigma$))

    U$\mathcal{F}$: U → UΣ → in,out {ch[ em(i) ]|i:KI • i ∈ cm(obs_AUI(u))}  UΣ
```

The meaning processes $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\mathcal{U}}$ are generic. Their sôle purpose is to provide a never ending recursion. “In-between” they “make use” of assembly, respectively unit specific functions here symbolised by $U\mathcal{A}$, respectively $U\mathcal{F}$.

C.4.3 Mereology, Part III 451

(See Sect. C.2.5 on page 139 for Mereology, Part II.) A little more meaning has been added to the notions of parts and connections. The within and adjacent to relations between parts (assemblies and units) reflect a phenomenological world of geometry, and the connected relation between parts (assemblies and units) reflect both physical and conceptual world understandings: physical world in that, for example, radio waves cross geometric “boundaries”, and conceptual world in that ontological classifications typically reflect lattice orderings where *overlaps* likewise cross geometric “boundaries”.

C.4.4 Discussion 452

Partial Evaluation

The assembly function "first" "functions" as a compiler. The 'compiler' translates an assembly structure into three process expressions: the $\mathcal{M}_{\mathcal{A}}(a)(a\sigma)$ invocation, the parallel composition of assembly processes, a', one for each sub-assembly of a, and the parallel composition of unit processes, one for each unit of assembly a — with these three process expressions "being put in parallel". The recursion in assembly ends when a sub-...-assembly consists of no sub-sub-...-assemblies. Then the compiling task ends and the many generated $\mathcal{M}_{\mathcal{A}}(a)(a\sigma)$ and $\mathcal{M}_{\mathcal{U}}(u)(u\sigma)$ process expressions are invoked.

Generalised Channel Processes 453

We can refine the meaning of connectors. Each connector, so far, was modelled by a CSP channel. CSP channels serve both as a synchronisation and as a communication medium. We now suggest to model it by a process. A channel process can be thought of as having four channels and a buffering process. Connector, κ:K, may connect parts π_i, π_j. The four channels could be thought of as indexed by $(\kappa, \pi_i), (\pi_i, \kappa), (\kappa, \pi_j)$ and (π_j, κ). The process buffer could, depending on parts p_i, p_j, be either queues, sets, bags, stacks, or other.

Part IV
Support Material

D Domain Actions BLANK 454

D.1 Definitions

D.2 Examples

455

D.3 Research Challenge 456

E Domain Events $\boxed{\text{BLANK}}$ 457

E.1 Definitions

E.2 Examples

458

E.3 Research Challenge

459

F Domain Behaviours $\boxed{\text{BLANK}}$

F.1 Definitions

F.2 Examples

460

F.3 Research Challenge

461

G A Specification Ontology 462

G.1 Description Ontology Versus Ontology Description

According to Wikipedia: *Ontology is the philosophical study of* (i) *the nature of being, existence or reality in general,* (ii) *as well as of the basic categories of being and their relations.*

Section 2 emphasized the need for describing domain phenomena and concepts. This section puts forward a description ontology: (i) *which "natures of being, existence or reality"* and (ii) *which "categories of being and their relations".* which we shall apply in the description of domain phenomena and concepts. 463

Yes, we do know that the term 'description ontology' can easily be confused with 'ontology description' — a term used very much in two computing related communities: AI (artificial intelligence) and WWW (World Wide Web). These communities use the term 'ontology' as we use the term 'domain' [5, 29, 33, 50, 51, 52, 53, 78].

By [domain] 'description ontology' we shall mean a set of notions that are used in describing a domain. So the ontology is one of the description language not of the domain that is being described.

G.2 Categories, Predicates and Observers for Describing Domains 464

It is not the purpose of this paper to motivate the categories, predicates and observer functions for describing phenomena and concepts. This is done elsewhere [14, 15, 19, 22, 23]. Instead we shall more-or-less postulate one approach to the analysis of domains. We do so by postulating a number of meta-categories, meta-predicates and meta-observer functions. They characterise those non-meta categories, predicates and observer functions that the domain engineer cum researcher is suggested to make use of. There may be other approaches [77, John Sowa, 1999] than the one put forward in this paper.

G.2.1 The Hypothetical Nature of Categories, Predicates and Observers 465

In the following we shall postulate some categories, that is, some meta-types:

categories
$\mathbb{ALPHA}, \mathbb{BETA}, \mathbb{GAMMA}$

What such a clause as the above means is that we postulate that there are such categories of "things" (phenomena and concepts) in the world of domains. That is, there is no proof that such "things" exists. It is just our way of modelling domains. If that way is acceptable to other domain science researchers, fine. In the end, which we shall never reach, those aspects of a, or the domain science, may "survive". If not, not ! 466

G.2.2 Predicates and Observers 467

With the categories just introduced we then go on to postulate some predicate and observer functions. For example:

predicate signatures
is_$\mathbb{ALPHA}$: “Things” $\rightarrow$ **Bool**
is_$\mathbb{BETA}$: “Things” $\rightarrow$ **Bool**
is_$\mathbb{GAMMA}$: “Things” $\rightarrow$ **Bool**
observer signatures
obs_$\mathbb{ALPHA}$: “Things” $\stackrel{\sim}{\rightarrow}$ $\mathbb{ALPHA}$
obs_$\mathbb{BETA}$: $\mathbb{ALPHA}$ $\stackrel{\sim}{\rightarrow}$ $\mathbb{BETA}$
obs_$\mathbb{GAMMA}$: $\mathbb{ALPHA}$ $\stackrel{\sim}{\rightarrow}$ $\mathbb{GAMMA}$

468 So we are “fixing” a logic !
The “Things” clause is a reference to the domain under scrutiny. Some ‘things’ in that domain are of category $\mathbb{ALPHA}$, or $\mathbb{BETA}$, or $\mathbb{GAMMA}$. Some are not. It is then postulated that from such things of category $\mathbb{ALPHA}$ one can observe things of categories $\mathbb{BETA}$ or $\mathbb{GAMMA}$. Whether this is indeed the case, i.e., that one can observe these things is a matter of conjecture, not of proof.

G.2.3 Meta-Conditions 469

Finally we may sometimes postulate the existence of a meta-axiom:

meta condition:
Predicates over $\mathbb{ALPHA}, \mathbb{BETA}$ and $\mathbb{GAMMA}$

Again, the promulgation of such logical meta-expressions are just conjectures, not the expression of “eternal” truths.

G.2.4 Discussion 470

So, all in all, we suggest four kinds of meta-notions:

- categories,
- is_Category predicates,
- obs_Property) predicates,
- obs_Category observers
- obs_Attribute observers, and
- meta-conditions (axiom-like predicates).

471 The **category [type]** A, B, ..., is_A, is_B, ... obs_A, obs_B, ... **meta-condition [axiom]** predicate notions derive from McCarthy's *analytic syntax* [63]. In that paper McCarthy also suggested a *synthetic syntax* constructor function: mk_A, ...At present, we find no need to introduce this *synthetic syntax* constructor function. A basic reason for this is that we are not constructing domain phenomena. The reason McCarthy (and computing science) needed the *synthetic syntax* constructor functions is that software is constructed.

Discussion: Thus the formal specification and the high level programming languages' use, that is, the software designers' use of **type** clauses, predicate functions and observer (in the form of selector) functions shall be seen in the context of specifications, respectively program code dealing with computable quantities and decomposing and constructing such quantities.

The proposal here, of suggesting that the domain engineer cum researcher makes us of **categories, predicates, observers** and **meta-conditions** is different. In domain descriptions an existing "universe of discourse" is being analysed. Perhaps the **categories, predicates, observers** and **meta-conditions** makes sense, perhaps the domain engineer cum researcher can use these descriptional "devices" to "compose" a consistent and relative complete "picture", i.e., description, of the domain under investigation.

Either the software designers' use of formal specification or programming language constructs is right or it is wrong, but the domain engineer cum researchers' use is just an attempt, a conjecture. If the resulting domain description is inconsistent, then it is wrong. But it can never be proven right. Right in the sense that it is **the** right description. As in physics, it is just a conjecture. There may be refutations of domain models.

G.2.5 Entities 472

What we shall describe is what we shall refer to as entities. In other words, there is a category and meta-logical predicate $\mathbb{ENTITY}$, is_$\mathbb{ENTITY}$. The is_$\mathbb{ENTITY}$ predicate applies to "whatever" in the domain, whether an entity or not, and "decides", i.e., is postulated to analyse whether that "thing" is an entity or not:

predicate signature:
is_$\mathbb{ENTITY}$: "Thing" → **Bool**
meta condition:
$\forall$ e:$\mathbb{ENTITY}$ • is_$\mathbb{ENTITY}$(e)

Discussion: When we say "things", or entities, others may say 'individuals', 'objects', or use other terms.

The meta-predicate is_$\mathbb{ENTITY}$ provides a rather "sweeping" notion, namely that someone, the domain engineer, an oracle or other, can decide whether "something" is to be described as a phenomenon or concept of the domain. 473

• • •

By introducing the predicate is_ENTITY we have put the finger on what this section is all about, namely *"what exists ?"* and *"what can be described ?"* We are postulating a description ontology. It may not be an adequate one. It may have flaws. But, for the purposes of raising some issues of epistemological and ontological nature, it is adequate.

G.2.6 Entity Categories 474

We postulate four entity categories:

category:
SIMPLE_ENTITY, ACTION, EVENT, BEHAVIOUR

Simple entities are **phenomena** or **concepts**. **Simple entity phenomena** are the things we
can point to, touch and see. They are manifest. Other phenomena, for example those we
can hear, smell, taste, or measure by physics (including chemistry) apparatus are properties
(attributes) of simple entity phenomena. **Concepts** are abstractions about phenomena
475 and/or other concepts. A subset of simple domain entities form a **state**. **Actions** are the
result of applying functions to simple domain entities and changing the **state**. **Events** are
state changes that satisfy a predicate on the 'before' and 'after states'. **Behaviours** are
476 sets of sequences (of sets of) actions and events.

category:
ENTITY = SIMPLE_ENTITY ∪ ACTION ∪ EVENT ∪ BEHAVIOUR

Discussion: The four categories of entities may overlap.
With each of the four categories there is a predicate:

predicate signature:
is_SIMPLE_ENTITY "Thing" → **Bool**
is_ACTION "Thing" → **Bool**
is_EVENT "Thing" → **Bool**
is_BEHAVIOUR "Thing" → **Bool**

Each of the above four predicates require that their argument t:"Thing" satisfies:

is_ENTITY(t)

The ∪ "union" is inclusive:

meta condition:
∀ t:``Thing''•is_ENTITY(t) ⇒
is_SIMPLE_ENTITY(t) ∨ is_ACTION(t) ∨ is_EVENT(t) ∨ is_BEHAVIOUR(t)

G.2.7 Simple Entities 477

We postulate that there are **atomic** simple entities, that there are [therefrom distinct] **composite** simple entities, and that a simple entity is indeed either atomic or composite. That atomic simple entities cannot meaningfully be described as consisting of proper other simple entities, but that composite simple entities indeed do consist of proper other simple entities. That is: 478

category:
  SIMPLE_ENTITY $=$ ATOMIC $\cup$ COMPOSITE
observer signature:
  is_ATOMIC: SIMPLE_ENTITY $\rightarrow$ **Bool**
  is_COMPOSITE: SIMPLE_ENTITY $\rightarrow$ **Bool**
meta condition:
  ATOMIC $\cap$ COMPOSITE $= \{\}$
  $\forall$ s: “Things”:SIMPLE_ENTITY •
    is_ATOMIC(s) $\equiv$ ∼is_COMPOSITE(s)

Discussion: We put in brackets, in the text paragraph before the above formulas, [therefrom distinct]. One may very well discuss this constraint — are there simple entities that are both atomic and composite ? — and that is done by Bertrand Russell in his ‘Philosophy of Logical Atomism’ [75].

G.2.8 Discrete and Continuous Entities 479

We postulate two forms of SIMPLE_ENTITIES: DISCRETE, such as a railroad net, a bank, a pipeline pump, and a securities instrument, and CONTINUOUS, such as oil and gas, coal and iron ore, and beer and wine.

category:
  SIMPLE_ENTITY $=$ DISCRETE_SIMPLE_ENTITY $\cup$ CONTINUOUS_SIMPLE_ENTITY
predicate signatures:
  is_DISCRETE_SIMPLE_ENTITY: SIMPLE_ENTITY $\rightarrow$ **Bool**
  is_CONTINUOUS_SIMPLE_ENTITY: SIMPLE_ENTITY $\rightarrow$ **Bool**
meta condition:
  [is it desirable to impose the following]
  $\forall$ s:SIMPLE_ENTITY •
    is_DISCRETE_SIMPLE_ENTITY(s) $\equiv$ ∼CONTINUOUS_SIMPLE_ENTITY(s) ?

Discussion: In the last lines above we raise the question whether it is ontologically possible or desirable to be able to have simple entities which are both discrete and continuous. Maybe we should, instead, express an axiom which dictates that every simple entity is at least of one of these two forms.

G.2.9 Attributes 480

Simple entities are characterised by their attributes: attributes have name, are of type and has some value; no two (otherwise distinct) attributes of a simple entity has the same name.

```
category:
    ATTRIBUTE, NAME, TYPE, VALUE
observer signature:
    obs_ATTRIBUTEs: SIMPLE_ENTITY → ATTRIBUTE-set
    obs_NAME: ATTRIBUTE → NAME
    obs_TYPE: ATTRIBUTE × NAME → TYPE
    obs_VALUE: ATTRIBUTE × NAME → VALUE
meta condition:
    ∀ s:SIMPLE_ENTITY •
        ∀ a,a′:ATTRIBUTE • {a,a′}⊆obs_ATTRIBUTEs(s)
         ∧ a≠a′ ⇒ obs_NAME(a)≠obs_NAME(a′)
```

481 Examples of attributes of atomic simple entities are: (i) A pipeline pump usually has the following attributes: `maximum pumping capacity, current pumping capacity, whether for oil or gas, diameter (of pipes to which the valve connects)`, etc. (ii) Attributes of a person usually includes `name, gender, birth date, central registration number, address, marital state, nationality`, etc. 482

Examples of attributes of composite simple entities are: (iii) A railway system usually has the following attributes: `name of system, name of geographic areas of location of rail nets and stations, whether a public or a private company, whether fully, partly or not electrified`, etc. (iv) Attributes of a bank usually includes: `name of bank, name of geographic areas of location of bank branch offices, whether a commercial portfolio bank or a high street, i.e., demand/deposit bank`, etc.

We do not further define what we mean by attribute names, types and values.

G.2.10 Atomic Simple Entities: Attributes 483

Atomic simple entities are characterised only by their attributes.

Discussion: We shall later cover a notion of domain actions, that is functions being applied to entities, including simple entities. We do not, as some do for programming languages, “lump” entities and functions (etc.) into what is there called ‘objects’.

G.2.11 Composite Simple Entities: Attributes, Sub-entities and Mereology

Composite simple entities are characterised by three properties: (i) their **attributes**, (ii) a proper set of one or more **sub-entities** (which are simple entities) and (iii) a **mereology** of these latter, that is, how they relate to one another, i.e., how they are composed.

Sub-entities 484

Proper sub-entities, that is simple entities properly contained, as immediate **parts** of a composite simple entity, can be observed (i.e., can be postulated to be observable):

```
observer signature:
   obs_SIMPLE_ENTITIES: COMPOSITE → SIMPLE_ENTITY-set
```

Mereology, Part IV 485

Mereology is the theory of **part-hood** relations: of the relations of part to whole and the relations of **part** to **part** within a **whole**. Suffice it to suggest some mereological structures:

- **Set Mereology:** The individual sub-entities of a composite entity are "un-ordered" like elements of a set. The obs_SIMPLE_ENTITIES function yields the set elements.

  ```
  predicate signature:
     is_SET: COMPOSITE → Bool
  ```

  486

- **Cartesian Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a Cartesian (grouping). The function obs_ARITY yields the arity, 2 or more, of the simple Cartesian entity. The function obs_CARTESIAN yields the Cartesian composite simple entity. 487

  ```
  predicate signature:
     is_CARTESIAN: COMPOSITE → Bool
  observer signatures:
     obs_ARITY: COMPOSITE ~→ Nat
        pre: obs_ARITY(s) is_CARTESIAN(s)
     obs_CARTESIAN: COMPOSITE ~→
                          SIMPLE_ENTITY × ... × SIMPLE_ENTITY
        pre obs_CARTESIAN(s): is_CARTESIAN(s)
  meta condition:
     ∀ c:SIMPLE_ENTITY•
        is_COMPOSITE(c)∧is_CARTESIAN(c) ⇒
           obs_SIMPLE_ENTITIES(c) = elements of obs_CARTESIAN(c)
         ∧ cardinality ofobs_SIMPLE_ENTITIES(c) = obs_ARITY(c)
  ```

  We just postulate the **elements of** and the **cardinality of** meta-functions. 488

- **List Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a list (i.e., a sequence). Where Cartesians are fixed arity sequences, lists are variable length sequences.

```
predicate signature:
   is_LIST: COMPOSITE $\rightarrow$ Bool
observer signatures:
   obs_LIST: COMPOSITE $\xrightarrow{\sim}$ list of SIMPLE_ENTITY
      pre is_LIST(s): is_COMPOSITE(s)
   obs_LENGTH: COMPOSITE $\xrightarrow{\sim}$ Nat
      pre is_LIST(s): is_COMPOSITE(s)
meta condition:
   $\forall$ s:SIMPLE_ENTITY•
      is_COMPOSITE(s)$\wedge$is_LIST(s) $\Rightarrow$
         obs_SIMPLE_ENTITIES(s) = elements of obs_LIST(s)
```

489 We also just postulate the **list of** and the **elements of** meta-functions.

- **Map Mereology:** The individual sub-entities of a map are "indexed" by unique definition set elements. Thus we can speak of pairings of unique map definition set element identifications and their not necessarily distinct range set elements. By a map we shall therefore understand a function, with a finite definition set, from distinct definition set elements to not necessarily distinct range elements, such that the pairs of (definition set,range) elements, which are all simple entities, can be characterised by a predicate. 490

  It is this, the finiteness of maps and the (potential, but often un-expressed) predicate, which 'distinguishes' maps from functions.

  Let us refer to the map as being of **category** MAP. Let us refer to the definition set elements of a map as being the DEFINITION_SET of the MAP. Let us refer to the range elements of such a map as being the RANGE of the MAP. No two definition set elements of a map, to repeat, are the same. Given a definition set element, s, of a map, m, one can obtain its IMAGE of the RANGE of m. 491

```
predicate signature:
   is_MAP: COMPOSITE $\rightarrow$ Bool
observer signatures:
   obs_MAP: COMPOSITE $\xrightarrow{\sim}$ MAP
      pre obs_MAP(c): is_MAP(c)
   obs_DEFINITION_SET: MAP $\rightarrow$ SIMPLE_ENTITY-set
      pre obs_MAP(c): is_MAP(c)
   obs_RANGE: MAP $\rightarrow$ SIMPLE_ENTITY-set
      pre obs_MAP(c): is_MAP(c)
   obs_IMAGE: MAP $\times$ SIMPLE_ENTITY $\xrightarrow{\sim}$ SIMPLE_ENTITY
      pre obs_IMAGE(m,d): is_MAP(m) $\wedge$ d $\in$ obs_DEFINITION_SET(m)
meta condition:
   $\forall$ m:SIMPLE_ENTITY•
```

```
    is_COMPOSITE(m)∧is_MAP(m) ⇒
        obs_SIMPLE_ENTITIES(m) =
            {(d,obs_IMAGE(c,d))|d:SIMPLE_ENTITY•d ∈ obs_DEFINITION_SET(m)}
```

492

Given that we can postulate "an existence" of the obs_DEFINITION_SET and the obs_RANGE observer functions we can likewise postulate a category

```
category
    MAP = map of SIMPLE_ENTITY  into ENTITY
observer signatures
    obs_DEF_SET: MAP → set of SIMPLE_ENTITY
    obs_RNG: MAP → set of ENTITY
meta condition
    ∀ m:MAP •
        obs_DEF_SET(m) = obs_DEFINITION_SET(m)
      ∧ obs_RNG(m) = obs_RANGE(m)
```

Here, again, the **map of ... into ...** is further unexplained.

We shall not pursue the notions of DEF_SET and RNG further. 493

- **Graph Mereology:** The individual sub-entities of a composite entity are "ordered" like elements of a graph, i.e., a net, of elements. Trees and lattices are just special cases of graphs. Any (immediate) sub-entity of a composite simple entity of GRAPH mereology may be related to any number of (not necessarily other) (immediate) sub-entities of that same composite simple entity GRAPH in a number of ways:it may immediately PRECEDE, or immediate SUCCEED or be BIDIRECTIONALLY_LINKED with these (immediate) sub-entities of that same composite simple entity. In the latter case some sub-entities PRECEDE a SIMPLE_ENTITY of the GRAPH, some sub-entities SUCCEED a SIMPLE_ENTITY of the GRAPH, some both. 494

```
predicate signature:
    is_GRAPH: COMPOSITE → Bool
observer signatures:
    obs_GRAPH: COMPOSITE $\overset{\sim}{\to}$ GRAPH
              pre obs_GRAPH(g): is_GRAPH(g)
    obs_PRECEDING_SIMPLE_ENTITIES:
          COMPOSITE × SIMPLE_ENTITY → SIMPLE_ENTITY-set
              pre obs_PRECEDING_SIMPLE_ENTITIES(c,s):
                        is_GRAPH(c) ∧ s ∈ obs_SIMPLE_ENTITIES(c)
    obs_SUCCEEDING_SIMPLE_ENTITIES:
          COMPOSITE × SIMPLE_ENTITY → SIMPLE_ENTITY-set
```

```
            pre obs_PRECEDING_SIMPLE_ENTITIES(c,s):
                    is_GRAPH(c) ∧ s ∈ obs_SIMPLE_ENTITIES(c)
    meta condition:
       ∀ c:SIMPLE_ENTITY • is_COMPOSITE(c) ∧ is_GRAPH(c)
            ⇒ let ss = SIMPLE_ENTITIES(c) in
               ∀ s′:SIMPLE_ENTITY • s′ ∈ ss
                  ⇒ obs_PRECEDING_SIMPLE_ENTITIES(c)(s′) ⊆ ss
                     ∧ obs_SUCCEEDING_SIMPLE_ENTITIES(c)(s′) ⊆ ss
            end
```

495

G.2.12 Discussion

Given a "thing", s, which satisfies is_SIMPLE_ENTITY(s), the domain engineer can now systematically analyse this "thing" using any of the predicates is_ATOMIC(s), is_COMPOSITE(s), is_SET(s), is_CARTESIAN(s), is_LIST(s), is_MAP(s), is_GRAPH(s), etcetera. and observer functions sketched above. 496

Given any SIMPLE_ENTITY the domain engineer can now analyse it to find out whether it is an ATOMIC or a COMPOSITE entity. An, in either case, the domain engineer can analyse it to find out about its ATTRIBUTES. If the SIMPLE_ENTITY is COMPOSITE then its SIMPLE_ENTITIES and their MEREOLOGY can be additionally ascertained. 497 In summery: If ATOMIC then ATTRIBUTES can be analysed. If COMPOSITE then ATTRIBUTES, SIMPLE_ENTITIES and MEREOLOGY can be analysed.

G.2.13 Actions 498

By a STATE we mean a set of one or more SIMPLE_ENTITIES. By an ACTION we shall understand the application of a FUNCTION to (a set of, including the state of) SIMPLE_ENTITIES such that a STATE change occurs. We postulate that the domain engineer can indeed decide, that is, conjecture, whether a "thing", which is an ENTITY is an ACTION.

category:
ACTION, FUNCTION, STATE
predicate signature:
is_ACTION: ENTITY → **Bool**

499

Given an ENTITY of category ACTION one can observe, i.e., conjecture the FUNCTION (being applied), the ARGUMENT CARTESIAN of SIMPLE_ENTITIES to which the FUNCTION is being applied, and the resulting change STATE change. Not all elements of the CARTESIAN ARGUMENT are SIMPLE STATE ENTITIES. 500

category:
  $\mathbb{STATE} = \mathbb{SIMPLE\_ENTITY}$
  $\mathbb{FUNCTION} = \mathbb{SIMPLE\_ENTITY} \times \mathbb{STATE} \to \mathbb{STATE}$
  $\mathbb{ARGUMENT} = \{|\mathrm{s}{:}\mathbb{SIMPLE\_ENTITY}\bullet\mathrm{is\_}\mathbb{CARTESIAN}(\mathrm{s})|\}$
observer signatures:
  obs_$\mathbb{ACTION}$: $\mathbb{ENTITY} \to \mathbb{ACTION}$
  obs_$\mathbb{FUNCTION}$: $\mathbb{ACTION} \to \mathbb{FUNCTION}$
  obs_$\mathbb{ARGUMENT}$: $\mathbb{ACTION} \to \mathbb{ARGUMENT}$
  obs_$\mathbb{INPUT_STATE}$: $\mathbb{ACTION} \to \mathbb{STATE}$
  obs_$\mathbb{RESULT_STATE}$: $\mathbb{ACTION} \to \mathbb{STATE}$

Discussion: Some pretty definite assertions were made above: We postulate that the domain engineer can indeed decide whether a "thing", which is an $\mathbb{ENTITY}$ is an $\mathbb{ACTION}$ And that one can observe the $\mathbb{FUNCTION}$, the $\mathbb{ARGUMENT}$ and the $\mathbb{RESULT}$ of an $\mathbb{ACTION}$. We do not really have to phrase it that deterministically. It is enough to say: One can speak of actions, functions, their arguments and their results. Ontologically we can do so. Whether, for any specific simple entity we can decide whether it is an actions is, in a sense, immaterial: we can always postulate that it is an action and then our analysis can be based on that hypothesis. This discussion applies `inter alia` to all of the entities being introduced here, together with their properties. 501

The domain engineer cum researcher can make such decisions as to whether an entity is a simple one, or an action, or an event or a behaviour. And from such a decision that domain engineer cum researcher can go on to make decisions as to whether a simple entity is discrete or continuous, and atomic or composite, and then onto a mereology for the composite simple entities. Similarly the domain engineer cum researcher can make decisions as to the function, arguments and results of an action. All these decisions does not necessarily represent the "truth". They hopefully are not "falsities". At best they are abstractions and, as such, they are approximations.

G.2.14 Events 502

By an $\mathbb{EVENT}$ we shall understand A pair, (σ, σ'), of $\mathbb{STATE}$s, a $\mathbb{STIMULUS}$, s, (which is like a $\mathbb{FUNCTION}$ of an $\mathbb{ACTION}$), and an $\mathbb{EVENT}$ $\mathbb{PREDICATE}$, $p : \mathcal{P}$, such that $p(\sigma, \sigma')(s)$, yields **true**.

The difference between an $\mathbb{ACTION}$ and an $\mathbb{EVENT}$ is two things: the $\mathbb{EVENT}$ $\mathbb{ACTION}$ need not originate within the analysed $\mathbb{DOMAIN}$, and the $\mathbb{EVENT}$ $\mathbb{PREDICATE}$ is trivially satisfied by most $\mathbb{ACTION}$s which originate within the analysed $\mathbb{DOMAIN}$. 503

Examples of events, that is, of predicates are: a bank goes "bust" (e.g., looses all its monies, i.e., bankruptcy), a bank account becomes negative, (unexpected) stop of gas flow and iron ore mine depleted. Respective stimuli of these events could be: (massive) loan defaults, a bank client account is overdrawn, pipeline breakage, respectively over-mining. 504

We postulate that the domain engineer from an $\mathbb{EVENT}$ can observe the $\mathbb{STIMULUS}$, the $\mathbb{BEFORE\_STATE}$, the $\mathbb{AFTER\_STATE}$ and the $\mathbb{EVENT\_PREDICATE}$. As said before: the domain engineer cum researcher can decide on these abstractions, these approximations. 505

```
category:
    STIMULUS = SIMPLE_ENTITY × STATE → STATE
    𝒫 = STATE × STATE → Bool
observer signatures:
    obs_STIMULUS: EVENT → STIMULUS
    obs_BEFORE_STATE: EVENT → STATE
    obs_AFTER_STATE: EVENT → STATE
    obs_EVENT_PREDICATE: EVENT → 𝒫
meta condition:
    ∀ e:EVENT •
         ∃ s:STIMULUS •
              INPUT_STATE(e) = BEFORE_STATE(s) ∧
              RESULT_STATE(e) = AFTER_STATE(s) ∧
              ∃ p:𝒫 •p(s)(INPUT_STATE(e),RESULT_STATE(e))
```

G.2.15 Behaviours 506

By a BEHAVIOUR we shall understand a set of sequences of ACTIONs and EVENTs such that some EVENTs in two or more such sequences have their STATEs and PREDICATEs express, for example, mutually exclusive synchronisation and communication EVENTs between these sequences which are each to be considered as simple SEQUENTIAL_BEHAVIOURs. Other forms than mutually exclusive synchronisation and communication EVENTs, that "somehow link" two or more behaviours, can be identified. 507

We may think of the mutually exclusive synchronisation and communication EVENTs as being designated simply by their PREDICATEs such as, for example, in CSP [56]:

```
type A, B, C, D, M
channel ch M
value
    f: A → out ch   C
    g: B → in ch   D
    f(a) ≡ ... point ℓ_f:ch!e ...
    g(b) ≡ ... point ℓ_g:ch? ...
```

Here the zero capacity buffer communication channel, ch, express mutual exclusivity, and the output/input clauses: ch!e and ch? express synchronisation and communication. 508

The predicate is here, in the CSP schema, "buried" in the simultaneous occurrence behaviour f having "reached point" **point** ℓ_f and behaviour g having "reached point" **point** ℓ_g. 509

We abstract from the orderly example of synchronisation and communication given above and introduce a further un-explained notion of behaviour (synchronisation and communication) BEHAVIOUR INTERACTION LABELs and allow BEHAVIOURs to now just be sets of sequences of ACTIONs and BEHAVIOUR INTERACTION LABELs. such that any one simple sequence has unique labels. 510

We can classify some BEHAVIOURs.

(i) SIMPLE SEQUENTIAL BEHAVIOURs are sequences of ACTIONs.

(ii) SIMPLE CONCURRENT BEHAVIOURs are sets of SIMPLE SEQUENTIAL BEHAVIOURs.

(iii) COMMUNICATING CONCURRENT BEHAVIOURs are sets of sequences of ACTIONs and BEHAVIOUR INTERACTION LABELs. We say that two or more such COMMUNICATING CONCURRENT BEHAVIOURs SYNCHRONISE & COMMUNICATE when all distinct BEHAVIOURs "sharing" a (same) label have all reached that label. 511

Many other composite behaviours can be observed. For our purposes it suffice with having just identified the above.

SIMPLE_ENTITIES, ACTIONs and EVENTs can be described without reference to time. BEHAVIOURs, in a sense, take place over time.[21] It will bring us into a rather long 512 discourse if we are to present some predicates, observer functions and axioms concerning behaviours — along the lines such predicates, observer functions and axioms were present, above, for SIMPLE_ENTITIES, ACTIONs and EVENTs. We refer instead to Johan van Benthems seminal work on the `The Logic of Time` [82]. In addition, more generally, we refer to A.N. Prior's [67, 68, 69, 70, 66] and McTaggart's works [64, 45, 74]. The paper by Wayne D. Blizard [26] proposes an axiom system for time-space.

G.2.16 Mereology, Part V 513

Compositionality of Entities

Simple entities — when composite — are said to exhibit a mereology. Thus composition of simple entities imply a mereology. We discussed mereologies of behaviours: simple sequential, simple concurrent, communicating concurrent, etc. Above we did not treat actions and events as potentially being composite. But we now relax that seeming constraint. There is, in principle, nothing that prevents actions and events from exhibiting mereologies. An 514 action, still instantaneous, can, for example, "fork" into a number of concurrent actions, all instantaneous, on "disjoint" parts of a state; or an instantaneous action can "dribble" (not little-by-little, but one-after-the-other. still instantaneously) into several actions as if a simple sequential behaviour, but instantaneous. Two or more events can occur simulta- 515 neously: two or more (up to four, usually) people become grandparents when a daughter of theirs give birth to their first grandchild; or an event can — again a "dribble" (not little-by-little, but instantaneously) — "rapidly" sequence through a number of instantaneous sub-events (with no intervening time intervals): A bankruptcy events immediately

[21] If it is important that ACTIONs take place over time, that is, are not instantaneous, then we can just consider ACTIONs as very simple SEQUENTIAL_BEHAVIOURs not involving EVENTs.

causes the bankruptcy of several enterprises which again causes the immediate bankruptcy
516 of several employes, etcetera.

The problems of compositionality of entities, whether simple, actions, events or behaviours, is was studied, initially, in [23, Bjørner and Eir, 2008]

G.2.17 Impossibility of Definite Mereological Analysis of Seemingly Composite Entities 517

It would be nice if there was a more-or-less obvious way of "deciphering" the mereology of an entity. In the many • (bulleted) items above (cf. Set, Cartesian, List, Map, Graph) we may have left the impression with the reader that is a more-or-less systematic way of uncovering the mereology of a composite entity. That is not the case: there is no such obvious way. It is a matter of both discovery and choice between seemingly alternative mereologies, and it is also a matter of choice of abstraction.

G.3 What Exists and What Can Be Described ? 518

519 In the previous section we have suggested a number of *categories*[22] of entities, a number of *predicate*[23] and *observer*[24] functions and a number of *meta conditions* (i.e., axioms). These concepts and their relations to one-another, suggest an ontology for describing domains. It is now very important that we understand these categories, predicates, observers and axioms properly.

G.3.1 Description Versus Specification Languages 520

Footnotes 22–24 (Page 176) summarised a number of main concepts of an ontology for describing domains. The categories and predicate and observer function signatures are not part of a formal language for descriptions. The identifiers used for these categories are intended to denote the real thing, classes of entities of a domain. In a philosophical discourse about describability of domains one refers to the real things. That alone prevents us from devising a formal specification language for giving (syntax and) semantics to a specification, in that language, of what these (Footnote 22–24) identifiers mean.

G.3.2 Formal Specification of Specific Domains 521

Once we have decided to describe a specific domain then we can avail ourselves of using one or more of a set of formal specification languages. But such a formal specification

[22] Some categories: ENTITY, SIMPLE_ENTITY, ACTION, EVENT, BEHAVIOUR, ATOMIC, COMPOSITE, DISCRETE, CONTINUOUS, ATTRIBUTE, NAME, TYPE, VALUE, SET, CARTESIAN, LIST, MAP, GRAPH, FUNCTION, STATE, ARGUMENT, STIMULUS, EVENT_PREDICATE, BEFORE_STATE, AFTER_STATE, SEQUENTIAL_BEHAVIOUR, BEHAVIOUR_INTERACTION_LABEL, SIMPLE_SEQUENTIAL_BEHAVIOUR, SIMPLE_CONCURRENT_BEHAVIOUR, COMMUNICATING_CONCURRENT_BEHAVIOUR, etc.

[23] Some predicates: is_ENTITY, is_SIMPLE_ENTITY, is_ACTION, is_EVENT, is_BEHAVIOUR, is_ATOMIC, is_COMPOSITE, is_DISCRETE_SIMPLE_ENTITY, is_CONTINUOUS_SIMPLE_ENTITY, is_SET, is_CARTESIAN, is_LIST, is_MAP, is_GRAPH, etc.

[24] Some observers: obs_SIMPLE_ENTITY, obs_ACTION, obs_EVENT, obs_BEHAVIOUR, obs_ATTRIBUTE, obs_NAME, obs_TYPE, obs_VALUE, obs_SET, obs_CARTESIAN, obs_ARITY, obs_LIST, obs_LENGTH, obs_DEFINITION_SET, obs_RANGE, obs_IMAGE, obs_GRAPH, obs_PRECEDING_SIMPLE_ENTITIES, obs_SUCCEEDING_SIMPLE_ENTITIES, obs_MEREOLOGY, obs_INPUT_STATE, obs_ARGUMENT, obs_RESULT_STATE, obs_STIMULUS, obs_EVENT_PREDICATE, obs_BEFORE_STATE, obs_AFTER_STATE, etc.

does not give meaning to identifiers of the categories and predicate and observer functions; they give meaning to very specific subsets of such categories and predicate and observer functions. And the domain specification now ascribes, not the real thing, but usually some form of mathematical structures as models of the specified domain.

G.3.3 Formal Domain Specification Languages 522

There are, today, 2009, a large number of formal specification languages. Some or textual, some are diagrammatic. The textual specification languages are like mathematical expressions, that is: linear text, often couched in an abstract "programming language" notation. The diagrammatic specification languages provide for the specifier to draw two-dimensional figures composed from primitives. Both forms of specification languages have precise mathematical meanings, but the linear textual ones additionally provide for proof rules. 523

Examples of textual, formal specification languages are

- `Alloy [59]:` model-oriented,
- `B, Event-B [1]:` model-oriented,
- `CafeOBJ [47]:` property-oriented (algebraic),
- `CASL [34]:` property-oriented (algebraic),
- `DC (Duration Calculus) [85]:` temporal logic,
- `RAISE, RSL [49]:` property and model-oriented,
- `TLA+ [61]:` temporal logic and sets,
- `VDM, VDM-SL [46]:` model-oriented and
- `Z [84]:` model-oriented.

`DC` and `TLA+` are often used in connection with either a model-oriented specification languages or just plain old discrete mathematics notation ! 524

But the model-oriented specification languages mentioned above do not succinctly express concurrency. The diagrammatic, formal specification languages, listed below, all do that:

- `Petri Nets [73]`,
- `Message Sequence Charts (MSC) [57]`,
- `Live Sequence Charts (LSC) [55]` and
- `Statecharts [54]`.

G.3.4 Discussion: “Take-it-or-leave-it !” 525

With the formal specification languages, not just those listed above, but with any conceivable formal specification language, the issue is: you can basically only describe using that language what it was originally intended to specify, and that, usually, was to specify software ! If, in the real domain you find phenomena or concepts, which it is somewhat clumsy and certainly not very abstract or, for you, outright impossible, to describe, then, well, then you cannot formalise them !

526

527

H Business Processes BLANK 528

H.1 Definitions

H.2 Business Process Focus 529

H.2.1 Entity-oriented Business Processes 530

H.2.4 Behaviour-oriented Business Processes 533

H.3 Discussion 534

H.4 Research Challenge 535

I Domain Intrinsics $\boxed{\text{BLANK}}$ 536

I.1 Delineation

I.2 Examples 537

I.3 Research Challenges

538

J Domain Support Technologies 539

J.1 Definition

By a support technology of a domain we shall understand either of a set of (one or more) alternative **entities, functions, events** and **behaviours** which "implement" an intrinsic phenomenon or concept. Thus for some one or more intrinsic phenomena or concepts there might be a technology which supports those phenomena or concepts.

J.2 Examples 540

Example 56 – Railway Switches (I):

We give a rough sketch description of possible rail unit switch technologies.

(i) In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.

(ii) With the advent of reasonably reliable mechanics, pulleys and levers[25] and steel wires, switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

(iii) This partial mechanical technology then emerged into electromechanics, and cabin tower staff was "reduced" to pushing buttons.

(iv) Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another). ■ End of Example 56

541

It must be stressed that Example 56 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electromechanics and the human operator interface (buttons, lights, sounds, etc.).

An aspect of supporting technology includes recording the state-behaviour in response
542 to external stimuli. We give an example.

Example 57 – Railway Switches (II):

Figure 13 indicates a way of formalising this aspect of a supporting technology. Figure 13 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd.

543

544 Another example shows another aspect of support technology: Namely that the technology must guarantee certain of its own behaviours, so that software designed to interface with
545 this technology, together with the technology, meets dependability requirements.

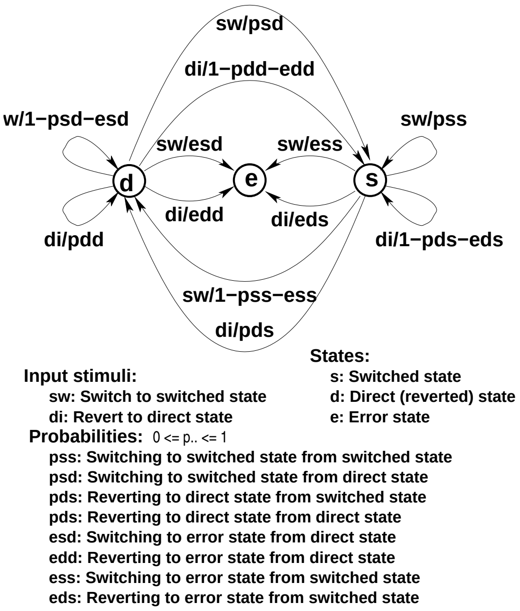

sw/psd
di/1−pdd−edd
w/1−psd−esd
sw/pss
sw/esd
sw/ess
d
e
s
di/edd
di/eds
di/pdd
di/1−pds−eds
sw/1−pss−ess
di/pds
States:
s: Switched state
d: Direct (reverted) state
e: Error state
Input stimuli:
sw: Switch to switched state
di: Revert to direct state
Probabilities: 0 <= p.. <= 1
pss: Switching to switched state from switched state
psd: Switching to switched state from direct state
pds: Reverting to direct state from switched state
pds: Reverting to direct state from direct state
esd: Switching to error state from direct state
edd: Reverting to error state from direct state
ess: Switching to error state from switched state
eds: Reverting to error state from switched state

Figure 13: Probabilistic state switching

Example 58 – Sampling Behaviour of Support Technologies:

Let us consider **i**ntrinsic **A**ir **T**raffic as a continuous function (→) from **T**ime to **F**light **L**ocations:

type
    T, F, L
    iAT = T → (F $\xrightarrow[m]{}$ L)

But what is observed, by some support technology, is not a continuous function, but a discrete sampling (a map $\xrightarrow[m]{}$):

    sAT = T $\xrightarrow[m]{}$ (F $\xrightarrow[m]{}$ L)

There is a support technology, say in the form of **radar** which “observes” the intrinsic traffic and delivers the sampled traffic:

value
    radar: iAT → sAT

But even the radar technology is not perfect. Its positioning of flights follows some probabilistic or statistical pattern:

```
type
    $P = \{|r{:}\mathbf{Real} \bullet 0 \le r \le 1|\}$
    $ssAT = P \underset{m}{\rightarrow} sAT\text{-}\mathbf{infset}$
value
    $radar'\colon iAT \overset{\sim}{\rightarrow} ssAT$
```

The radar technology will, with some probability produce either of a set of samplings, and with some other probability some other set of samplings, etc.[26] ∎ End of Example 58

J.3 Support Technology Quality Control, a Sketch 546

How can we express that a given technology delivers a reasonable support ? One approach is to postulate intrinsic and technology states (or observed behaviours), Θ_i, Θ_s, a support technology τ and a "**close**ness" predicate:

```
type
    $\Theta\_i, \Theta\_s$
value
    $\tau\colon \Theta\_i \rightarrow P \underset{m}{\rightarrow} \Theta\_s\text{-}\mathbf{infset}$
    close: $\Theta\_i \times \Theta\_s \rightarrow \mathbf{Bool}$
```

and then require that an experiment can be performed which validates the support technology.

547

The experiment is expressed by the following axiom:

```
value
    p_threshhold:P
axiom
    $\forall\ \theta\_i{:}\Theta\_i \bullet$
        let $p\theta\_ss = \tau(\theta\_i)$ in
        $\forall\ p{:}P \bullet p > p\_threshhold \Rightarrow$
            $\theta\_s{:}\Theta\_s \bullet \theta\_s \in p\theta\_ss(p) \Rightarrow close(\theta\_i,\theta\_s)$ end
```

J.4 Research Challenges BLANK 548

K Domain Rules and Regulations 549

K.1 Definitions

By a **rule** of an enterprise (an institution) we understand a **syntactic** piece of text whose **meaning** apply in any pair of actual present and potential next **states** of the enterprise, and then evaluates to either **true** or **false**: **the rule has been obeyed, or the rule has been (or will be, or might be) broken.**

By a **regulation** of an enterprise (an institution) we understand a **syntactic** piece of text whose **meaning**, for example, apply in **states** of the enterprise where a rule has been broken, and when applied in such states will **change the state**, **that is, "remedy" the "breaking of a rule".**

K.2 Abstraction of Rules and Regulations 550

Stimuli are introduced in order to capture the possibility of rule-breaking next states. **Rul**es **Reg**ulations Θ To each of the three syntactic notions: **Sti**muli, **Rul**es and **Reg**ulations there is a **meaning**, i.e., a semantic function from syntax to semantics. The meaning, **STI**, of **Sti**muli, are state transitions, that is, a stimulus provokes a state change. The meaning, **RUL**, of a **Rul**e, is a predicate over a before (stimulus) state and an after (stimulus) state. The meaning, **REG**, of a **Reg**ulation, is another transition, intended to replace stimula transitions whose **Rul**e predicate does not hold, that is, the regulation transition shall lead to an after state for which the rule now holds.

type
  $\mathrm{Sti}, \mathrm{Rul}, \mathrm{Reg}, \Theta$
  $\mathrm{RulReg} = \mathrm{Rul} \times \mathrm{Reg}$
  $\mathrm{STI} = \Theta \to \Theta$
  $\mathrm{RUL} = (\Theta \times \Theta) \to \mathbf{Bool}$
  $\mathrm{REG} = \Theta \to \Theta$
value
  $\mathrm{meaning}\colon \mathrm{Sti} \to \mathrm{STI}$
  $\mathrm{meaning}\colon \mathrm{Rul} \to \mathrm{RUL}$
  $\mathrm{meaning}\colon \mathrm{Reg} \to \mathrm{REG}$
  $\mathrm{valid}\colon \mathrm{Sti} \times \mathrm{Rul} \to \Theta \to \mathbf{Bool}$
  $\mathrm{valid}(\mathrm{sti},\mathrm{rul})\theta \equiv (\mathrm{meaning}(\mathrm{rul}))(\theta,\mathrm{meaning}(\mathrm{sti})\theta)$
axiom
  $\forall\ \mathrm{sti}{:}\mathrm{Sti},(\mathrm{rul},\mathrm{reg}){:}\mathrm{RulReg},\theta{:}\Theta \bullet \sim\mathrm{valid}(\mathrm{sti},\mathrm{rul})\theta \Rightarrow \mathrm{meaning}(\mathrm{rul})(\theta,\mathrm{meaning}(\mathrm{reg})\theta)$

K.2.1 Quality Control of Rules and Regulations 551

The axiom above presents us with a guideline for checking the suitability of (pairs of) rules and regulations in the context of stimuli: for every proposed pair of rules and regulations and for every conceivable stimulus check whether the stimulus might cause a breaking of the rule and, if so, whether the regulation will restore the system to an acceptable state.

K.2.2 Research Challenges 552

The above sketched a quality control procedure for ‘stimuli, rules and regulations’. It left out the equally important ‘monitoring’ aspects. Here is a research challenge: Develop experimentally two or three distinct models of domains involving distinct sets of rules and regulations. Then propose and study concrete implementations of procedures for quality monitoring and control of ‘stimuli, rules and regulations’.

L Domain Scripts, Licenses and Contracts 553

L.1 Domain Scripts

By a **domain script** we shall understand a structured text which can be interpreted as a set of rules ("in disguise").

Example 59 – **Timetables** We shall view timetables as scripts.

In this example (that is, Pages 192–198) we shall first narrate and formalise the **syntax**, including the well-formedness of timetable scripts, then we consider the **pragmatics** of timetable scripts, including the bus routes prescribed by these journey descriptions and timetables marked with the status of its currently active routes, and finally we consider the **semantics** of timetable, that is, the traffic they denote.

In Example. ?? on contracts for bus traffic, we shall assume the timetable scripts of this section. 554

We all have some image of how a timetable may manifest itself. Figure 14 shows some such images.

BUS TIME
EL PRÁCTICO
Desde el centro de Iguazú a Eterno Reverdecer
Desde Eterno Reverdecer al centro de Iguazú
BUSSRUTER
532 Otta - Høvringen - Smuksjøseter
532 Smuksjøseter - Høvringen - Otta

Figure 14: Some bus timetables: Italy, India and Norway

555

What we shall capture is, of course, an abstraction of "such timetables". We claim that the enumerated narrative which now follows and its accompanying formalisation represents an adequate description. Adequate in the sense that the reader "gets the idea", that is, is shown how to narrate and formalise when faced with an actual task of describing a concept of timetables.

In the following we distinguish between bus lines and bus rides. A bus line description is basically a sequence of two or more bus stop descriptions. A bus ride is basically a sequence of two or more time designators.[27] A bus line description may cover several bus rides. The former have unique identifications and so has the latter. The times of the latter are the approximate times at which the bus of that bus line and bus identification is supposed to be at respective stops. You may think of the bus line identification to express something like "The Flying Scotsman", and the bus ride identification something like "The 4.50 From Paddington".

[27] We do not distinguish between a time and a time description. That is, when we say April 22, 2010, 16: 0 8 we mean it either as a description of the time at which this text that you are now reading was LaTeX compiled, and as "that time !".

The Syntax of Timetable Scripts

125. Time is a concept covered earlier. Bus lines and bus rides have unique names (across any set of time tables). Hub and link identifiers, HI, LI, were treated from the very beginning.

126. A TimeTable associates to Bus Line Identifiers a set of Journies.

127. Journies are designated by a pair of a BusRoute and a set of BusRides.

128. A BusRoute is a triple of the Bus Stop of origin, a list of zero, one or more intermediate Bus Stops and a destination Bus Stop.

129. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.

130. A Bus Schedule a triple of the initial departure Time, a list of zero, one or more intermediate bus stop Times and a destination arrival Time.

131. A Bus Stop (i.e., its position) is a Fraction of the distance along a link (identified by a Link Identifier) from an identified hub to an identified hub.

132. A Fraction is a **Real** properly between 0 and 1.

133. The Journies must be well_formed in the context of some net. 556

type
125. T, BLId, BId
126. TT = BLId $\overrightarrow{m}$ Journies
127. Journies$'$ = BusRoute $\times$ BusRides
128. BusRoute = BusStop $\times$ BusStop* $\times$ BusStop
129. BusRides = BId $\overrightarrow{m}$ BusSched
130. BusSched = T $\times$ T* $\times$ T
131. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
132. Frac = {|r:**Real**•0<r<1|}
133. Journies = {|j:Journies$'$•$\exists$ n:N • wf_Journies(j)(n)|}

The free *n* in $\exists$ n:N • wf_Journies(j)(n) is the net given in the license. 557

Well-formedness of Journies

134. A set of journies is well-formed

135. if the bus stops are all different[28],

136. if a defined notion of a bus line is embedded in some line of the net, and

[28] This restriction is, strictly speaking, not a necessary domain property. But it simplifies our subsequent formulations.

137. if all defined bus trips (see below) of a bus line are commensurable.

```
value
134. wf_Journies: Journies → N → Bool
134. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
135.   diff_bus_stops(bs1,bsl,bsn) ∧
136.   is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
137.   commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
```

558

138. The bus stops of a journey are all different

139. if the number of elements in the list of these equals the length of the list.

```
value
138. diff_bus_stops: BusStop × BusStop* × BusStop → Bool
138. diff_bus_stops(bs1,bsl,bsn) ≡
139.   card elems ⟨bs1⟩^bsl^⟨bsn⟩ = len ⟨bs1⟩^bsl^⟨bsn⟩
```

559

We shall refer to the (concatenated) list (⟨bs1⟩^bsl^⟨bsn⟩ = **len** ⟨bs1⟩^bsl^⟨bsn⟩) of all bus stops as the bus line.

140. To explain that a bus line is embedded in a line of the net

141. let us introduce the notion of all lines of the net, lns,

142. and the notion of projecting the bus line on link sector descriptors.

143. For a bus line to be embedded in a net then means that there exists a line, ln, in the net, such that a compressed version of the projected bus line is amongst the set of projections of that line on link sector descriptors.

560

```
value
140. is_net_embedded_bus_line: BusStop* → N → Bool
140. is_net_embedded_bus_line(bsl)(hs,ls)
141.   let lns = lines(hs,ls),
142.       cbln = compress(proj_on_links(bsl)(elems bsl)) in
143.   ∃ ln:Line • ln ∈ lns ∧ cbln ∈ projs_on_links(ln) end
```

561

144. Projecting a list (*) of BusStop descriptors (mkBS(hi,li,f,hi′)) onto a list of Sector Descriptors ((hi,li,hi′))

145. we recursively unravel the list from the front:

146. if there is no front, that is, if the whole list is empty, then we get the empty list of sector descriptors,

147. else we obtain a first sector descriptor followed by those of the remaining bus stop descriptors.

```
value
144. proj_on_links: BusStop* → SectDescr*
144. proj_on_links(bsl) ≡
145.   case bsl of
146.     ⟨⟩ → ⟨⟩,
147.     ⟨mkBS(hi,li,f,hi′)⟩⌢bsl′ → ⟨(hi,li,hi′)⟩⌢proj_on_links(bsl′)
147.   end
```

562

148. By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.

149. The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.

150. We express the function recursively.

151. If the argument sector descriptor list an empty result sector descriptor list is yielded;

152. else

153. if the front argument sector descriptor has not yet been inserted in the result sector descriptor list it is inserted else an empty list is “inserted”

154. in front of the compression of the rest of the argument sector descriptor list.

563

```
148. compress: SectDescr* → SectDescr-set → SectDescr*
149. compress(sdl)(sds) ≡
150.  case sdl of
151.    ⟨⟩ → ⟨⟩,
152.    ⟨sd⟩⌢sdl′ →
153.       (if sd ∈ sds then ⟨sd⟩ else ⟨⟩ end)
154.       ⌢compress(sdl′)(sds\{sd}) end
```

In the last recursion iteration (line 154.) the continuation argument sds$\backslash\{$sd$\}$ can be shown to be empty: $\{\}$. 564

155. We recapitulate the definition of lines as sequences of sector descriptions.

156. Projections of a line generate a set of lists of sector descriptors.

157. Each list in such a set is some arbitrary, but ordered selection of sector descriptions. The arbitrariness is expressed by the "ranged" selection of arbitrary subsets isx of indices, isx⊆**inds** ln, into the line ln. The "ordered-ness" is expressed by making that arbitrary subset isx into an ordered list isl, isl=sort(isx).

```
type
155. Line′ = (HI×LI×HI)* axiom ... type Line = ...
value
156. projs_on_links: Line → Line′-set
156. projs_on_links(ln) ≡
157.   {⟨isl(i)|i:⟨1..len isl⟩⟩|isx:Nat-set•isx⊆inds ln∧isl=sort(isx)}
```

565

158. sorting a set of natural numbers into an ordered list, isl, of these is expressed by a post-condition relation between the argument, isx, and the result, isl.

159. The result list of (arbitrary) indices must contain all the members of the argument set;

160. and "earlier"elements of the list must precede, in value, those of "later" elements of the list.

```
value
158. sort: Nat-set → Nat*
158. sort(isx) as isl
159.   post card isx = lsn isl ∧ isx = elems isl ∧
160.       ∀ i:Nat • {i,i+1}⊆inds isl ⇒ isl(i)<isl(i+1)
```

566

161. The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:

162. All the intermediate bus stop times must equal in number that of the bus stop list.

163. We then express, by case distinction, the reality (i.e., existence) and timeliness of the bus stop descriptors and their corresponding time descriptors – and as follows.

164. If the list of intermediate bus stops is empty, then there is only the bus stops of origin and destination, and they must be exist and must fit time-wise. 567

165. If the list of intermediate bus stops is just a singleton list, then the bus stop of origin and the singleton intermediate bus stop must exist and must fit time-wise. And likewise for the bus stop of destination and the the singleton intermediate bus stop.

166. If the list is more than a singleton list, then the first bus stop of this list must exist and must fit time-wise with the bus stop of origin.

167. As for Item 166 but now with respect to last, resp. destination bus stop.

168. And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops

169. they must exist and fit time-wise.

568

```
value
161. commensurable_bus_trips: Journies → N → Bool
161. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
162.   ∀ (t1,til,tn):BusSched•(t1,til,tn)∈ rng js∧len til=len bsl∧
163.     case len til of
164.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
165.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧fit((til(1),t2),(bsl(1),bsn))(hs,ls),
166.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
167.            real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
168.            ∀ i:Nat•{i,i+1}⊆inds til ⇒
169.              real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

569

170. A pair of (adjacent) bus stops exists and a pair of times, that is the time interval between them, fit with the bus stops if the following conditions hold:

171. All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).

172. There exists links, l, l′, for the identified bus stop links, li, li′,

173. such that these links connect the identified bus stop hubs.

174. Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops

175. The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.

176. So we leave it as an exercise to the reader to change/augment the description, in order to be able to ascertain a plausible measure of distance.

177. The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.

178. So we leave also this as an exercise to the reader to complete.

570

```
170. real_and_fit: (T×T)×(BusStop×BusStop) → N → Bool
170. real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
171.   {hi,hi′,hi″,hi‴}⊆his(hs)∧
172.   ∃ l,l′:L•{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
173.     obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
174.   afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

175. distance: BusStop × BusStop → N → Distance
176. distance(bs1,bs2)(n) ≡ ... [ left as an exercise ! ] ...

177. afit: TI → Distance → Bool
178.   [ time interval fits distance between bus stops ]
```

L.2 Domain Licenses and Contracts 571

By a **domain license** we shall understand a right or permission granted in accordance with law by a competent authority to engage in some business or occupation, to do some act, or to engage in some transaction which but for such license would be unlawful Merriam Webster On-line [79]. 572

By a **domain contract** we shall understand very much the same thing as a license: a binding agreement between two or more persons or parties — one which is legally enforceable. 573

The concepts of licenses and licensing express relations between *actors* (licensors (the authority) and licensees), *simple entities* (artistic works, hospital patients, public administration and citizen documents) and *operations* (on simple entities), and as performed by actors. By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations: which operations on which entities the licensee is allowed (is licensed, is permitted) to perform. As such a license denotes a possibly infinite set of allowable behaviours. 574

We shall consider four kinds of entities: (i) digital recordings of artistic and intellectual nature: music, movies, readings ("audio books"), and the like, (ii) patients in a hospital: as represented also by their patient medical records, (iii) documents related to public government: citizen petitions, law drafts, laws, administrative forms, letters between state and local government adminsitrators and between these and citizens, court verdicts, etc., and (iv) bus timetables, as part of contracts for a company to provide bus servises. 575

The *permissions* and *obligations* issues are: (i) for the owner (agent) of some intellectual property to be paid (i.e., an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works; (ii) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient; (iii) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents; (iv) for citizens to enjoy timely and reliable bus services and the local government to secure adequate price-performance standards. 576

Example 60 – A Health Care License Language

Citizens go to hospitals in order to be treated for some calamity (disease or other), and by doing so these citizens become patients. At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution. This request is directed at medical staff, that is, the patient authorises medical staff to perform a set of actions upon the patient. One could claim, as we shall, that the patient issues a license. 577

Patients and Patient Medical Records So patients and their attendant patient medical records (PMRs) are the main entities, the "works" of this domain. We shall treat them synonymously: PMRs as surrogates for patients. Typical actions on patients — and hence on PMRs — involve admitting patients, interviewing patients, analysing patients, diagnosing patients, planning treatment for patients, actually treating patients, and, under normal circumstance, to finally release patients. 578

Medical Staff Medical staff may request ('refer' to) other medical staff to perform some of these actions. One can conceive of describing action sequences (and 'referrals') in the form of hospitalisation (not treatment) plans. We shall call such scripts for licenses. 579

Professional Health Care The issue is now, given that we record these licenses, their being issued and being honoured, whether the handling of patients at hospitals follow, or does not follow properly issued licenses.

We refer to the abstract syntax formalised below (that is, formulas 1.–5.). The work on the specific form of the syntax has been facilitated by the work reported in [4].[29] 580

A Notion of License Execution State In the context of the Artistic License Language licensees could basically perform licensed actions in any sequence and as often as they so desired. There were, of course, some obvious constraints. Operations on local works could not be done before these had been created — say by copying. Editing could only be done on local works and hence required a prior action of, for example, copying a licensed work. In the context of hospital health care most of the actions can only be performed if the patient has reached a suitable state in the hospitalisation. We refer to Fig. 15 on the next page for an idealised hospitalisation plan.

581

582

We therefore suggest to join to the licensed commands an indicator which prescribe the (set of) state(s) of the hospitalisation plan in which the command action may be performed.

[29] As this work, [4], has yet to be completed the syntax and annotations given here may change.

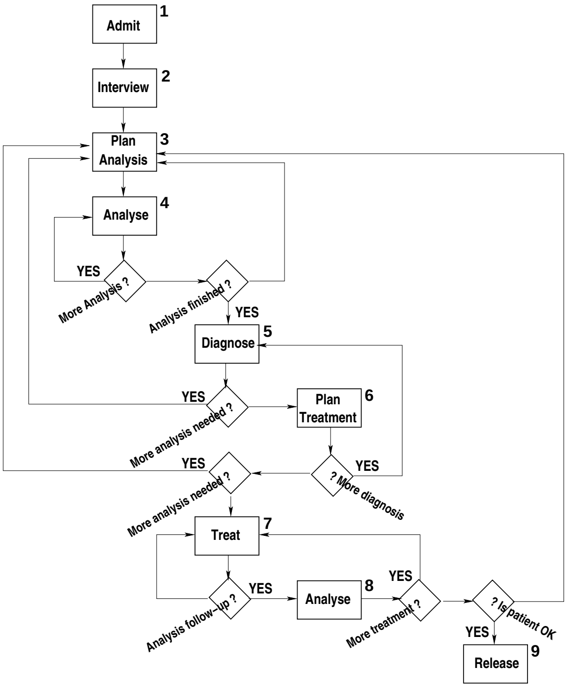


Figure 15: An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

Two or more medical staff may now be licensed to perform different (or even same !) actions in same or different states. If licensed to perform same action(s) in same state(s) — well that may be "bad license programming" if and only if it is bad medical practice ! One cannot design a language and prevent it being misused! 583

The License Language The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform.

type
0. Ln, Mn, Pn
1. License = Ln $\times$ Lic
2. Lic == mkLic(staff1:Mn,mandate:ML,pat:Pn)
3. ML == mkML(staff2:Mn,to_perform_acts:CoL-**set**)
4 CoL = Cmd | ML | Alt
5. Cmd == mkCmd(σs:Σ-**set**,stmt:Stmt)
6 Alt == mkAlt(cmds:Cmd-**set**)
7. Stmt = **admit** | **interview** | **plan-analysis** | **do-analysis**
| **diagnose** | **plan-treatment** | **treat** | **transfer** | **release**

584

The above syntax is correct RSL. But it is decorated! The subtypes {|**boldface keyword**|} are inserted for readability.

(0.) Licenses, medical staff and patients have names.

(1.) Licenses further consist of license bodies (Lic).

(2.) A license body names the licensee (Mn), the patient (Pn), and,

(3.) through the "mandated" licence part (ML), it names the licensor (Mn) and which set of commands (C) or (o) implicit licenses (L, for CoL) the licensor is mandated to issue.

(4.) An explicit command or licensing (CoL) is either a command (Cmd), or a sub-license (ML) or an alternative.

585

(5.) A command (Cmd) is a state-labelled statement.

(3.) A sub-license just states the command set that the sub-license licenses. As for the Artistic License Language the licensee chooses an appropriate subset of commands. The context "inherits" the name of the patient. But the sub-licensee is explicitly mandated in the license!

(6.) An alternative is also just a set of commands. The meaning is that either the licensee choose to perform the designated actions or, as for ML, but now freely choosing the sub-licensee, the licensee (now new licensor) chooses to confer actions to other staff.

586

(7.) A statement is either an admit, an interview, a plan analysis, an analysis, a diagnose, a plan treatment, a treatment, a transfer, or a release directive Information given in the patient medical report for the designated state inform medical staff as to the details of analysis, what to base a diagnosis on, of treatment, etc.

587

8. Action = Ln × Act
9. Act = Stmt | SubLic
10. SubLic = mkSubLic(sublicensee:Ln,license:ML)

(8.) Each action actually attempted by a medical staff refers to the license, and hence the patient name.

(9.) Actions are either of an admit, an interview, a plan analysis, an analysis, a diagnose, a plan treatment, a treatment, a transfer, or a release actions. 588Each individual action is only allowed in a state σ if the action directive appears in the named license and the patient (medical record) designates state σ.

(10.) Or an action can be a sub-licensing action. Either the sub-licensing action that the licensee is attempting is explicitly mandated by the license (4. ML), or is an alternative one thus implicitly mandated (6.). The full sub-license, as defined in (1.–3.) is compiled from contextual information.

589

Example 61 – A Public Administration License Language

The Three Branches of Government By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)[30], understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public government. Typically national parliament and local (province and city) councils are part of law-making government, law-enforcing government is called the executive (the administration), and law-interpreting government is called the judiciary [system] (including lawyers etc.). 590

Documents A crucial means of expressing public administration is through *documents.*[31] We shall therefore provide a brief domain analysis of a concept of documents. (This document domain description also applies to patient medical records and, by some "light" interpretation, also to artistic works — insofar as they also are documents.)

591

Documents are *created*, *edited* and *read*; and documents can be *copied*, *distributed*, the subject of *calculations* (interpretations) and be *shared* and *shredded* . 592

Document Attributes With documents one can associate, as attributes of documents, the *actors* who created, edited, read, copied, distributed (and to whom distributed),shared, performed calculations and shredded documents.

With these operations on documents, and hence as attributes of documents one can, again conceptually, associate the *location* and *time* of these operations. 593

Actor Attributes and Licenses With actors (whether agents of public government or citizens) one can associate the *authority* (i.e., the *rights*) these actors have with respect to performing actions on documents. We now intend to express these *authorisations as licenses*. 594

Document Tracing An issue of public government is whether citizens and agents of public government act in accordance with the laws — with actions and laws reflected in documents such that the action documents enables a trace from the actions to the laws "governing" these actions.

We shall therefore assume that every document can be traced back to its law-origin as well as to all the documents any one document-creation or -editing was based on. 595

A Document License Language The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform.

type
0. Ln, An, Cfn
1. L == Grant | Extend | Restrict | Withdraw
2. Grant == mkG(license:Ln,licensor:An,granted_ops:Op-**set**,licensee:An)

[30] *De l'esprit des lois* (*The Spirit of the Laws*), published 1748

[31] Documents are, for the case of public government to be the "equivalent" of artistic works.

```
3. Extend   ==  mkE(licensor:An,licensee:An,license:Ln,with_ops:Op-set)
4. Restrict == mkR(licensor:An,licensee:An,license:Ln,to_ops:Op-set)
5. Withdraw == mkW(licensor:An,licensee:An,license:Ln)
6. Op       == Crea|Edit|Read|Copy|Licn|Shar|Rvok|Rlea|Rtur|Calc|Shrd
```

596

```
type
7.   Dn, DCn, UDI
8.   Crea  == mkCr(dn:Dn,doc_class:DCn,based_on:UDI-set)
9.   Edit  == mkEd(doc:UDI,based_on:UDI-set)
10.  Read  == mkRd(doc:UDI)
11.  Copy  == mkCp(doc:UDI)
12a. Licn  == mkLi(kind:LiTy)
12b. LiTy  == grant | extend | restrict | withdraw
13.  Shar  == mkSh(doc:UDI,with:An-set)
14.  Rvok  == mkRv(doc:UDI,from:An-set)
15.  Rlea  == mkRl(dn:Dn)
16.  Rtur  == mkRt(dn:Dn)
17.  Calc  == mkCa(fcts:CFn-set,docs:UDI-set)
18.  Shrd  == mkSh(doc:UDI)
```

597
(0.) The are names of licenses (Ln), actors (An), documents (UDI), document classes (DCn) and calculation functions (Cfn).

(1.) There are four kinds of licenses: granting, extending, restricting and withdrawing.

(2.) Actors (licensors) grant licenses to other actors (licensees). An actor is constrained to always grant distinctly named licenses. No two actors grant identically named licenses.[32] A set of operations on (named) documents are granted. 598

(3.–5.) Actors who have issued named licenses may extend, restrict or withdraw the license rights (wrt. operations, or fully).

(6.) There are nine kinds of operation authorisations. Some of the next explications also explain parts of some of the corresponding actions (see (16.–24.).

(7.) There are names of documents (Dn), names of classes of documents (DCn), and there are unique document identifiers (UDI). 599

(8.) **Creation** results in an initially void document which is

not necessarily uniquely named (dn:Dn) (but that name is uniquely associated with the unique document identifier created when the document is created[33]) typed by a document class name (dcn:DCn) and possibly based on one or more identified documents (over which the licensee (at least) has reading rights). We can presently omit consideration of the document class concept. "based on" means that the initially void document contains references to those

[32] This constraint can be enforced by letting the actor name be part of the license name.

[33] — hence there is an assumption here that the create operation is invoked by the licensee exactly (or at most) once.

(zero, one or more) documents.[34] The "based on" documents are moved from licensor to licensee. 600

(9.) **Editing** a document may be based on "inspiration" from, that is, with reference to a number of other documents (over which the licensee (at least) has reading rights). What this "be based on" means is simply that the edited document contains those references. (They can therefore be traced.) The "based on" documents are moved from licensor to licensee if not already so moved as the result of the specification of other authorised actions. 601

(10.) **Reading** a document only changes its "having been read" status (etc.) — as per [10]. The read document, if not the result of a copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. 602

(11.) **Copying** a document increases the document population by exactly one document. All previously existing documents remain unchanged except that the document which served as a master for the copy has been so marked. The copied document is like the master document except that the copied document is marked to be a copy (etc.) — as per [10]. The master document, if not the result of a create or copy, is moved from licensor to licensee if not already so moved as the result of the specification of other authorised actions. 603

(12a.) A licensee can **sub-license** (sL) certain operations to be performed by other actors.

(12b.) The granting, extending, restricting or withdrawing permissions, cannot name a license (the user has to do that), do not need to refer to the licensor (the licensee issuing the sub-license), and leaves it open to the licensor to freely choose a licensee. One could, instead, for example, constrain the licensor to choose from a certain class of actors. The licensor (the licensee issuing the sub-license) must choose a unique license name. 604

(13.) A document can be **shared** between two or more actors. One of these is the licensee, the others are implicitly given read authorisations. (One could think of extending, instead the licensing actions with a **shared** attribute.) The shared document, if not the result of a create and edit or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. Sharing a document does not move nor copy it. 605

(14.) Sharing documents can be **revoked**. That is, the reading rights are removed.

(15.) The **release** operation: if a licensor has authorised a licensee to create a document (and that document, when created got the unique document identifier udi:UDI) then that licensee can **release** the created, and possibly edited document (by that identification) to the licensor, say, for comments. The licensor thus obtains the master copy. 606

(16.) The **return** operation: if a licensor has authorised a licensee to create a document (and that document, when created got the unique document identifier udi:UDI) then that licensee can **return** the created, and possibly edited document (by that identification) to the licensor — "for good"! The licensee relinquishes all control over that document. 607

(17.) Two or more documents can be subjected to any one of a set of permitted **calculation** functions. These documents, if not the result of a creates and edits or copies, are moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. Observe that there can be many calculation permissions, over overlapping documents and functions.

[34] They can therefore be traced (etc.) — as per [10].

(18.) A document can be **shredded**. It seems pointless to shred a document if that was the only right granted wrt. document. 608

```
17. Action = Ln × Clause
18. Clause =  Cre | Edt | Rea | Cop | Lic | Sha | Rvk | Rel | Ret | Cal | Shr
19. Cre == mkCre(dcn:DCn,based_on_docs:UID-set)
20. Edt == mkEdt(uid:UID,based_on_docs:UID-set)
21. Rea == mkRea(uid:UID)
22. Cop == mkCop(uid:UID)
23. Lic == mkLic(license:L)
24. Sha == mkSha(uid:UID,with:An-set)
25. Rvk == mkRvk(uid:UID,from:An-set)
25. Rev == mkRev(uid:UID,from:An-set)
26. Rel == mkRel(dn:Dn,uid:UID)
27. Ret == mkRet(dn:Dn,uid:UID)
28. Cal == mkCal(fct:Cfn,over_docs:UID-set)
29. Shr == mkShr(uid:UID)
```

609 A clause elaborates to a state change and usually some value. The value yielded by elaboration of the above create, copy, and calculation clauses are **unique document identifiers**. These are chosen by the "system". 610

(17.) Actions are **tagged** by the name of the license with respect to which their authorisation and document names has to be checked. No action can be performed by a licensee unless it is so authorised by the named license, both as concerns the operation (create, edit, read, copy, license, share, revoke, calculate and shred) and the documents actually named in the action. They must have been mentioned in the license, or, created or copies of downloaded (and possibly edited) documents or copies of these — in which cases operations are inherited. 611

(19.) A licensee may **create** documents if so licensed — and obtains all operation authorisations to this document.

(20.) A licensee may **edit** "downloaded" (edited and/or copied) or created documents.

(21.) A licensee may **read** "downloaded" (edited and/or copied) or created and edited documents.

(22.) A licensee may (conditionally) **copy** "downloaded" (edited and/or copied) or created and edited documents. The licensee decides which name to give the new document, i.e., the copy. All rights of the master are inherited to the copy. 612

(23.) A licensee may **issue licenses** of the kind permitted. The licensee decides whether to do so or not. The licensee decides to whom, over which, if any, documents, and for which operations. The licensee looks after a proper ordering of licensing commands: first grant, then sequences of zero, one or more either extensions or restrictions, and finally, perhaps, a withdrawal. 613

(24.) A "downloaded" (possibly edited or copied) document may (conditionally) be **shared** with one or more other actors. Sharing, in a digital world, for example, means that any edits done after the opening of the sharing session, can be read by all so-granted other actors. 614

(25.) Sharing may (conditionally) be **revoked**, partially or fully, that is, wrt. original "sharers".

(26.) A document may be **released**. It means that the licensor who originally requested a document (named dn:Dn) to be created now is being able to see the results — and is expected to comment on this document and eventually to re-license the licensee to further work. 615

(27.) A document may be **returned**. It means that the licensor who originally requested a document (named dn:Dn) to be created is now given back the full control over this document. The licensee will no longer operate on it. 616

(28.) A license may (conditionally) apply any of a licensed set of **calculation functions** to "downloaded" (edited, copied, etc.) documents, or can (unconditionally) apply any of a licensed set of calculation functions to created (etc.) documents. The result of a calculation is a document. The licensee obtains all operation authorisations to this document (— as for created documents).

(29.) A license may (conditionally) **shred** a "downloaded" (etc.) document.

617

Example 62 – A Bus Services Contract Language

In a number of steps ('A Synopsis', 'A Pragmatics and Semantics Analysis', and 'Contracted Operations, An Overview') we arrive at a sound basis from which to formulate the narrative. We shall, however, forego such a detailed narrative. Instead we leave that detailed narrative to the reader. (The detailed narrative can be "derived" from the formalisation.)

618 **A Synopsis** :

Contracts obligate transport companies to deliver bus traffic according to a timetable. The timetable is part of the contract. A contractor may sub-contract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable. Contractors are either public transport authorities or contracted transport companies. Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit[35]. The cancellation rights are spelled out in the contract[36]. A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor[37]. Etcetera.

619 **A Pragmatics and Semantics Analysis** :

The "works" of the bus transport contracts are two: the timetables and, implicitly, the designated (and obligated) bus traffic. A bus timetable appears to define one or more bus lines, with each bus line giving rise to one or more bus rides. We assume a timetable description along the lines of Sect. 59. Nothing is (otherwise) said about regularity of bus rides. It appears that bus ride cancellations must be reported back to the contractor. And we assume

[35] We do not treat this aspect further in this book.

[36] See Footnote 35.

[37] See Footnote 35.

that cancellations by a sub-contractor is further reported back also to the sub-contractor's contractor. Hence eventually that the public transport authority is notified. 620

Nothing is said, in the contracts, such as we shall model them, about passenger fees for bus rides nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor. So we shall not bother, in this example, about transport costs nor transport subsidies. But will leave that necessary aspect as an exercise.

The opposite of cancellations appears to be 'insertion' of extra bus rides, that is, bus rides not listed in the time table, but, perhaps, mandated by special events[38] We assume that such insertions must also be reported back to the contractor. 621

We assume concepts of acceptable and unacceptable bus ride delays. Details of delay acceptability may be given in contracts, but we ignore further descriptions of delay acceptability. but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors.

We finally assume that sub-contractors cannot (otherwise) change timetables. (A timetable change can only occur after, or at, the expiration of a license.) Thus we find that contracts have definite period of validity. (Expired contracts may be replaced by new contracts, possibly with new timetables.) 622

Contracted Operations, An Overview So these are the operations that are allowed by a contractor according to a contract: (i) *start:* to perform, i.e., to start, a bus ride (obligated); (ii) *cancel:* to cancel a bus ride (allowed, with restrictions); (iii) *insert:* to insert a bus ride; and (iv) *subcontract:* to sub-contract part or all of a contract. 623

Syntax We treat separately, the syntax of contracts (for a schematised example see Page 207) and the syntax of the actions implied by contracts (for schematised examples see Page 208).

Contracts

An example contract can be 'schematised':

cid: **contractor** cor **contracts sub-contractor** cee

to perform operations

{"start","cancel","insert","subcontract"}

with respect to timetable tt.

624

We assume a context (a global state) in which all contract actions (including contracting) takes place and in which the implicit net is defined. 625

[38] Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

179. contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.

180. A contract has a unique identification, names the contractor and the sub-contractor (and we assume the contractor and sub-contractor names to be distinct). A contract also specifies a contract body.

181. A contract body stipulates a timetable and the set of operations that are mandated or allowed by the contractor.

182. An Operation is either a "`start`" (i.e., start a bus ride), a bus ride "`cancel`"lation, a bus ride "`insert`", or a "`subcontract`"ing operation.

626

type
179. CId, CNm
180. Contract = CId × CNm × CNm × Body
181. Body = Op-set × TT
182. Op == ″`start`″ | ″`cancel`″ | ″`insert`″ | ″`subcontract`″

An abstract example contract:

(cid,cnm_i,cnm_j,({″`start`″,″`cancel`″,″`insert`″,″`sublicense`″},tt))

627

Actions

Concrete example actions can be schematised:

(a) cid: **conduct bus ride** (blid,bid) **to start at time** t

(b) cid: **cancel bus ride** (blid,bid) **at time** t

(c) cid: **insert bus ride like** (blid,bid) **at time** t

The schematised license (Page 34) shown earlier is almost like an action; here is the action form:

(d) cid: **sub-contractor** cnm′ **is granted a contract** cid′

to perform operations {"conduct","cancel","insert",sublicense"}

with respect to timetable tt′.

628 All actions are being performed by a sub-contractor in a context which defines that sub-contractor cnm, the relevant net, say n, the base contract, referred here to by cid (from which this is a sublicense), and a timetable tt of which tt′ is a subset. contract name cnm′ is new and is to be unique. The subcontracting action can (thus) be simply transformed into a contract as shown on Page 34. 629

```
type
   Action = CNm × CId × (SubCon | SmpAct) × Time
   SmpAct = Start | Cancel | Insert
   Conduct == mkSta(s_blid:BLId,s_bid:BId)
   Cancel == mkCan(s_blid:BLId,s_bid:BId)
   Insert = mkIns(s_blid:BLId,s_bid:BId)
   SubCon == mkCon(s_cid:CId,s_cnm:CNm,s_body:(s_ops:Op-set,s_tt:TT))

examples:
   (a) (cnm,cid,mkSta(blid,id),t)
   (b) (cnm,cid,mkCan(blid,id),t)
   (c) (cnm,cid,mkIns(blid,id),t)
   (d) (cnm,cid,mkCon(cid′,({″conduct″,″cancel″,″insert″,″sublicense″},tt′),t))

where: cid′ = generate_CId(cid,cnm,t)      See Item/Line 185
```

630 We observe that the essential information given in the start, cancel and insert action prescriptions is the same; and that the RSL record-constructors (mkSta, mkCan, mkIns) make them distinct. 631

Uniqueness and Traceability of Contract Identifications

183. There is a “root” contract name, rcid.

184. There is a “root” contractor name, rcnm.

```
value
183  rcid:CId
184  rcnm:CNm
```

632 All other contract names are derived from the root name. Any contractor can at most generate one contract name per time unit. Any, but the root, sub-contractor obtains contracts from other sub-contractors, i.e., the contractor. Eventually all sub-contractors, hence contract identifications can be referred back to the root contractor. 633

185. Such a contract name generator is a function which given a contract identifier, a subcontractor name and the time at which the new contract identifier is generated, yields the unique new contract identifier.

186. From any but the root contract identifier one can observe the contract identifier, the sub-contractor name and the time that “went into” its creation.

```
value
185  gen_CId: CId × CNm × Time → CId
186  obs_CId: CId ~→ CIdL [ pre obs_CId(cid):cid≠rcid ]
186  obs_CNm: CId ~→ CNm [ pre obs_CNm(cid):cid≠rcid ]
186  obs_Time: CId ~→ Time [ pre obs_Time(cid):cid≠rcid ]
```

634

187. All contract names are unique.

```
axiom
187  ∀ cid,cid′:CId•cid≠cid′⇒
187    obs_CId(cid)≠obs_CId(cid′) ∨ obs_CNm(cid)≠obs_CNm(cid′)
187    ∨ obs_LicNm(cid)=obs_CId(cid′)∧obs_CNm(cid)=obs_CNm(cid′)
187      ⇒ obs_Time(cid)≠obs_Time(cid′)
```

635

188. Thus a contract name defines a trace of license name, sub-contractor name and time triple, “all the way back” to “creation”.

```
type
   CIdCNmTTrace = TraceTriple*
   TraceTriple == mkTrTr(CId,CNm,s_t:Time)
value
188  contract_trace: CId → LCIdCNmTTrace
188  contract_trace(cid) ≡
188    case cid of
188      rcid → ⟨⟩,
188      _ → contract_trace(obs_LicNm(cid))^⟨obs_TraceTriple(cid)⟩
188    end

188  obs_TraceTriple: CId → TraceTriple
188  obs_TraceTriple(cid) ≡
188    mkTrTr(obs_CId(cid),obs_CNm(cid),obs_Time(cid))
```

636

The trace is generated in the chronological order: most recent contract name generation times last.

Well, there is a theorem to be proven once we have outlined the full formal model of this contract language: namely that time entries in contract name traces increase with increasing indices.

```
theorem
   ∀ licn:LicNm •
      ∀ trace:LicNmLeeNmTimeTrace • trace ∈ license_trace(licn) ⇒
            ∀ i:Nat • {i,i+1}⊆inds trace ⇒ s_t(trace(i))<s_t(trace(i+1))
```

637

Execution State

Local and Global States

Each sub-contractor has an own local state and has access to a global state. All sub-contractors access the same global state. The global state is the bus traffic on the net. There is, in addition, a notion of running-state. It is a meta-state notion. The running state "is made up" from the fact that there are n sub-contractors, each communicating, as contractors, over channels with other sub-contractors. The global state is distinct from sub-contractor to sub-contractor – no sharing of local states between sub-contractors. We now examine, in some detail, what the states consist of. 638

Global State

The net is part of the global state (and of bus traffics). We consider just the bus traffic.

189. Bus traffic is a modelled as a discrete function from densely positioned time points to a pair of the (possibly dynamically changing) net and the position of busses. Bus positions map bus numbers to the physical entity of busses and their position.

190. A bus is positioned either

191. at a hub (coming from some link heading for some link), or

192. on a link, some fraction of the distance from a hub towards a hub, or

193. at a bus stop, some fraction of the distance from a hub towards a hub.

639

type
131. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

189. BusTraffic = T $\underset{m}{\rightarrow}$ (N × (BusNo $\underset{m}{\rightarrow}$ (Bus × BPos)))
190. BPos = atHub | onLnk | atBS
191. atHub == mkAtHub(s_fl:LI,s_hi:HI,s_tl:LI)
192. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
193. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
Frac = {|f:**Real**•0<f<1|}

We shall consider BusTraffic (with its Net) to reflect the global state. 640

Local Sub-contractor Contract States: Semantic Types

A sub-contractor state contains, as a state component, the zero, one or more contracts that the sub-contractor has received and that the sub-contractor has sublicensed.

type

  Body = Op-**set** × TT
  LicΣ = RcvLicΣ×SubLicΣ×LorBusΣ
  RcvLicΣ = LorNm $\xrightarrow[m]{}$(LicNm $\xrightarrow[m]{}$(Body×TT))
  SubLicΣ = LeeNm $\xrightarrow[m]{}$(LicNm $\xrightarrow[m]{}$Body)
  LorBusΣ ... [see ″`Local sub-contractor Bus States: Semantic Types`″ next] ...

(Recall that LorNm and LeeNm are the same.)

In RecvLics we have that LorNm is the name of the contractor by whom the contract has been granted, LicNm is the name of the contract assigned by the contractor to that license, Body is the body of that license, and TT is that part of the timetable of the Body which has not (yet) been sublicensed.

In DespLics we have that LeeNm is the name of the sub-contractor to whom the contract has been despatched, the first (left-to-right) LicNm is the name of the contract on which that sublicense is based , the second (left-to-right) LicNm is the name of the sublicense, and License is the contract named by the second LicNm. 641

Local Sub-contractor Bus States: Semantic Types

The sub-contractor state further contains a bus status state component which records which buses are free, FreeBusΣ, that is, available for dispatch, and where "garaged", which are in active use, ActvBusΣ, and on which bus ride, and a bus history for that bus ride, and histories of all past bus rides, BusHistΣ. A trace of a bus ride is a list of zero, one or more pairs of times and bus stops. A bus history, BusHistory, associates a bus trace to a quadruple of bus line identifiers, bus ride identifiers, contract names and sub-contractor name.[39] 642

type

  BusNo
  BusΣ = FreeBusesΣ × ActvBusesΣ × BusHistsΣ
  FreeBusesΣ = BusStop $\xrightarrow[m]{}$ BusNo-**set**
  ActvBusesΣ = BusNo $\xrightarrow[m]{}$ BusInfo
  BusInfo = BLId×BId×LicNm×LeeNm×BusTrace
  BusHistsΣ = Bno $\xrightarrow[m]{}$ BusInfo*
  BusTrace = (Time×BusStop)*
  LorBusΣ = LeeNm $\xrightarrow[m]{}$ (LicNm $\xrightarrow[m]{}$ ((BLId×BId) $\xrightarrow[m]{}$(BNo×BusTrace)))

A bus is identified by its unique number (i.e., registration) plate (BusNo). We could model a bus by further attributes: its capacity, etc., for for the sake of modelling contracts this is enough. The two components are modified whenever a bus is commissioned into action or returned from duty, that is, twice per bus ride. 643

[39] In this way one can, from the bus history component ascertain for any bus which for whom (sub-contractor), with respect to which license, it carried out a further bus line and bus ride identified tour and its trace.

Local Sub-contractor Bus States: Update Functions

```
value
   update_BusΣ: Bno×(T×BusStop) → ActBusΣ → ActBusΣ
   update_BusΣ(bno,(t,bs))(actσ) ≡
      let (blid,bid,licn,leen,trace) = actσ(bno) in
      actσ†[ bno↦(licn,leen,blid,bid,trace⌢⟨(t,bs)⟩) ] end
      pre bno ∈ dom actσ

   update_FreeΣ_ActΣ:
      BNo×BusStop→BusΣ→BusΣ
   update_FreeΣ_ActΣ(bno,bs)(freeσ,actvσ) ≡
      let (_,_,_,_,trace) = actσ(b) in
      let freeσ′ = freeσ†[ bs ↦ (freeσ(bs))∪{b} ] in
      (freeσ′,actσ\{b}) end end
      pre bno ∉ freeσ(bs) ∧ bno ∈ dom actσ
```

644

```
   update_LorBusΣ:
       LorNm×LicNm×lee:LeeNm×(BLId×BId)×(BNo×Trace)
          →LorBusΣ→out {l_to_l[ leen,lorn ]|lorn:LorNm•lorn ∈ leenms\{leen}}  LorΣ
   update_LorBusΣ(lorn,licn,leen,(blid,bid),(bno,tr))(lbσ) ≡
      l_to_l[ leenm,lornm ]!Licensor_BusHistΣMsg(bno,blid,bid,libn,leen,tr) ;
      lbσ†[ leen↦(lbσ(leen))†[ licn↦((lbσ(leen))(licn))†[ (blid,bid)↦(bno,trace) ] ] ]
      pre leen ∈ dom lbσ ∧ licn ∈ dom (lbσ(leen))

   update_ActΣ_FreeΣ:
      LeeNm×LicNm×BusStop×(BLId×BId)→BusΣ→BusΣ×BNo
   update_ActΣ_FreeΣ(leen,licn,bs,(blid,bid))(freeσ,actvσ) ≡
      let bno:Bno • bno ∈ freeσ(bs) in
      ((freeσ\{bno},actvσ ∪ [ bno↦(blid,bid,licnm,leenm,⟨⟩) ]),bno) end
      pre bs ∈ dom freeσ ∧ bno ∈ freeσ(bs) ∧ bno ∉ dom actvσ ∧ [ bs exists ... ]
```

645

Constant State Values

There are a number of constant values, of various types, which characterise the “business of contract holders”. We define some of these now.

194. For simplicity we assume a constant net — constant, that is, only with respect to the set of identifiers links and hubs. These links and hubs obviously change state over time.

195. We also assume a constant set, leens, of sub-contractors. In reality sub-contractors, that is, transport companies, come and go, are established and go out of business. But assuming constancy does not materially invalidate our model. Its emphasis is on contracts and their implied actions — and these are unchanged wrt. constancy or variability of contract holders. 646

196. There is an initial bus traffic, tr.

197. There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.

198. To maintain the bus traffic "spelled out", in total, by timetable tt one needs a number of buses.

199. The various bus companies (that is, sub-contractors) each have a number of buses. Each bus, independent of ownership, has a unique (car number plate) bus number (BusNo).

     These buses have distinct bus (number [registration] plate) numbers.

200. We leave it to the reader to define a function which ascertain the minimum number of buses needed to implement traffic tr.

647

value
194. net : N,
195. leens : LeeNm-**set**,
196. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
197. t_0 : T • $t_0 \geq$ **min dom** tr,
198. min_no_of_buses : **Nat** • necessary_no_of_buses(itt),
199. busnos : BusNo-**set** • **card** busnos $\geq$ min_no_of_buses
200. necessary_no_of_buses: TT → **Nat**

648

201. To "bootstrap" the whole contract system we need a distinguished contractor, named init_leen, whose only license originates with a "ghost" contractor, named root_leen (o, for outside [the system]).

202. The initial, i.e., the distinguished, contract has a name, root_licn.

203. The initial contract can only perform the `"sublicense"` operation.

204. The initial contract has a timetable, tt.

205. The initial contract can thus be made up from the above.

649

value
201. root_leen,init_ln : LeeNm • root_leen $\notin$ leens $\wedge$ initi_leen $\in$ leens,
202. root_licn : LicNm
203. iops : Op-**set** = {″`sublicense`″},
204. itt : TT,
205. init_lic:License = (root_licn,root_leen,(iops,itt),init_leen)

650

Initial Sub-contractor Contract States

type
  InitLicΣs = LeeNm $\underset{m}{\rightarrow}$ LicΣ
value
  ilσ:LicΣ=([init_leen $\mapsto$ [root_leen $\mapsto$ [iln $\mapsto$ init_lic]]]
    $\cup$ [leen $\mapsto$ [ ] | leen:LeeNm • leen $\in$ leenms$\backslash${init_leen}],[ ],[ ])

651

Initial Sub-contractor Bus States

206. Initially each sub-contractor possesses a number of buses.

207. No two sub-contractors share buses.

208. We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.

209. We do not prescribe a “satisfiable and practical” such initial assignment (ibσs).

210. But we can constrain ibσs.

211. The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.

212. Active bus states must be empty.

213. No two free bus states must share buses.

214. All bus histories are void.

652

type
206. AllBuses′ = LeeNm $\underset{m}{\rightarrow}$ BusNo-**set**
207. AllBuses = {|ab:AllBuses′•$\forall$ {bs,bs′}$\subseteq$**rng** ab$\wedge$bns$\neq$bns′$\Rightarrow$bns $\cap$ bns′={}|}
208. InitBusΣs = LeeNm $\underset{m}{\rightarrow}$ BusΣ
value

```
207. allbuses:Allbuses • dom allbuses = leenms ∪ {root_leen} ∧ ∪ rng allbuses = busnos

208. ibσs:InitBusΣs
209. wf_InitBusΣs: InitBusΣs → Bool
210. wf_InitBusΣs(iσs) ≡
211.   dom iσs = leenms ∧
212.   ∀ (_,abσ,_):BusΣ•(_,abσ,_) ∈ rng iσs ⇒ abσ=[ ] ∧
213.   ∀ (fbiσ,abiσ),(fbjσ,abjσ):BusΣ •
213.     {(fbiσ,abiσ),(fbjσ,abjσ)}⊆rng iσs
213.       ⇒ (fbiσ,actiσ)≠(fbjσ,actjσ)
213.         ⇒ rng fbiσ ∩ rng fbjσ = {}
214.           ∧ actiσ=[ ]=actjσ
```

653 **Communication Channels** :

The running state is a meta notion. It reflects the channels over which contracts are issued; messages about committed, cancelled and inserted bus rides are communicated, and fund transfers take place. 654

Sub-Contractor↔Sub-Contractor Channels

Consider each sub-contractor (same as contractor) to be modelled as a behaviour. Each sub-contractor (licensor) behaviour has a unique name, the LeeNm. Each sub-contractor can potentially communicate with every other sub-contractor. We model each such communication potential by a channel. For n sub-contractors there are thus $n \times (n-1)$ channels.

```
channel { l_to_l[ fi,ti ] | fi:LeeNm,ti:LeeNm • {fi,ti}⊆leens ∧ fi≠ti } LLMSG
type LLMSG = ...
```

We explain the declaration: **channel** { l_to_l[fi,ti] | fi:LeeNm, ti:LeeNm • fi≠ti } LLMSG. It prescribes $n \times (n-1)$ channels (where n is the cardinality of the sub-contractor name sets). Each channel is prescribed to be capable of communicating messages of type MSG. The square brackets [...] defines l_to_l (sub-contractor-to-sub-contractor) as an array.

We shall later detail the BusRideNote, CancelNote, InsertNote and FundXfer message types.

655

Sub-Contractor↔Bus Channels

Each sub-contractor has a set of buses. That set may vary. So we allow for any sub-contractor to potentially communicate with any bus. In reality only the buses allocated and scheduled by a sub-contractor can be "reached" by that sub-contractor.

```
channel { l_to_b[ l,b ] | l:LeeNm,b:BNo • l ∈ leens ∧ b ∈ busnos } LBMSG
type LBMSG = ...
```

656

Sub-Contractor↔Time Channels

Whenever a sub-contractor wishes to perform a contract operation that sub-contractor needs know the time. There is just one, the global time, modelled as one behaviour: time_clock.

channel { l_to_t[l] | l:LeeNm • l ∈ leens } LTMSG
type LTMSG = ...

657

Bus↔Traffic Channels

Each bus is able, at any (known) time to ascertain where in the traffic it is. We model bus behaviours as processes, one for each bus. And we model global bus traffic as a single, separate behaviour.

channel { b_to_tr[b] | b:BusNo • b ∈ busnos } LTrMSG
type BTrMSG == reqBusAndPos(s_bno:BNo,s_t:Time) | (Bus×BusPos)

658

Buses↔Time Channel

Each bus needs to know what time it is.

channel { b_to_t[b] | b:BNo • b ∈ busnos } BTMSG
type BTMSG ...

659

Run-time Environment :
So we shall be modelling the transport contract domain as follows: As for behaviours we have this to say. There will be n sub-contractors. One sub-contractor will be initialised to one given license. You may think of this sub-contractor being the transport authority. Each sub-contractor is modelled, in RSL, as a CSP-like process. With each sub-contractor, l_i, there will be a number, b_i, of buses. That number may vary from sub-contractor to sub-contractor. There will be b_i channels of communication between a sub-contractor and that sub-contractor's buses, for each sub-contractor. There is one global process, the traffic. There is one channel of communication between a sub-contractor and the traffic. Thus there are n such channels. 660

As for operations, including behaviour interactions we assume the following. All operations of all processes are to be thought of as instantaneous, that is, taking nil time ! Most such operations are the result of channel communications either just one-way notifications, or inquiry requests. Both the former (the one-way notifications) and the latter (inquiry requests) must not be indefinitely barred from receipt, otherwise holding up the notifier. The latter (inquiry requests) should lead to rather immediate responses, thus must not lead to dead-locks.

The System Behaviour : 661

The system behaviour starts by establishing a number of licenseholder and bus_ride behaviours and the single time_clock and bus_traffic behaviours 662

```
value
   system: Unit → Unit
   system() ≡
      licenseholder(init_leen)(ilσ(init_leen),ibσ(init_leen))
      ∥ (∥ {licenseholder(leen)(ilσ(leen),ibσ(leen))
             | leen:LeeNm•leen ∈ leens\{init_leen}})
      ∥ (∥ {bus_ride(b,leen)(root_lorn,"nil")
             | leen:LeeNm,b:BusNo •leen ∈ dom allbuses ∧ b ∈ allbuses(leen)})
      ∥ time_clock(t₀) ∥ bus_traffic(tr)
```

663 The initial licenseholder behaviour states are individually initialised with basically empty license states and by means of the global state entity bus states. The initial bus behaviours need no initial state other than their bus registration number, a "nil" route prescription, and their allocation to contract holders as noted in their bus states.

664 Only a designated licenseholder behaviour is initialised to a single, received license.

Semantic Elaboration Functions

The Licenseholder Behaviour

215. The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.

216. It internally non-deterministically (⌈⌉) alternates between

    a) performing the licensed operations (on the net and with buses),

    b) receiving information about the whereabouts of these buses, and informing contractors of its (and its subsub-contractors') handling of the contracts (i.e., the bus traffic), and

    c) negotiating new, or renewing old contracts.

```
215. licenseholder: LeeNm → (LicΣ×BusΣ) → Unit
216. licenseholder(leen)(licσ,busσ) ≡
216.    licenseholder(leen)((lic_ops⌈⌉bus_mon⌈⌉neg_licenses)(leen)(licσ,busσ))
```

665

The Bus Behaviour

217. Buses ply the network following a timed bus route description.

A timed bus route description is a list of timed bus stop visits.

218. A timed bus stop visit is a pair: a time and a bus stop.

219. Given a bus route and a bus schedule one can construct a timed bus route description.

    a) The first result element is the first bus stop and origin departure time.

    b) Intermediate result elements are pairs of respective intermediate schedule elements and intermediate bus route elements.

    c) The last result element is the last bus stop and final destination arrival time.

220. Bus behaviours start with a “nil” bus route description.

666

```
type
217.  TBR = TBSV*
218.  TBSV = Time × BusStop
value
219.  conTBR: BusRoute × BusSched → TBR
219.  conTBR((dt,til,at),(bs1,bsl,bsn)) ≡
219a)    ⟨(dt,bs1)⟩
219b)    ⌢ ⟨(til[ i ],bsl[ i ])|i:Nat•i:⟨1..len til⟩⟩
219c)    ⌢ ⟨(at,bsn)⟩
           pre: len til = len bsl
type
220.  BRD == "nil" | TBR
```

667

221. The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,

222. The bus repeatedly observes the time, t, and its position, po, in the traffic.

223. There are now four case distinctions to be made.

224. If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.

225. If the bus is at the destination of its journey then it so informs its owner (i.e., the sub-contractor) and resumes being idle.

226. If the bus is ‘en route’, at a bus stop, then it so informs its owner and continues the journey.

227. In all other cases the bus continues its journey

668

```
value
221. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
221.     in,out l_to_b[leen,bno], in,out b_to_tr[bno], in b_to_t[bno] Unit
221. bus_ride(leen,bno)(licn,brd) ≡
222.   let t = b_to_t[bno]? in
222.   let (bus,pos) = (b_to_tr[bno]!reqBusAndPos(bno,t) ; b_to_tr[bno]?) in
223.   case (brd,pos) of
224.     (″nil″,mkAtBS(_,_,_,_)) →
224.          let (licn,brd′) = (l_to_b[leen,bno]!reqBusRid(pos);l_to_b[leen,bno]?) in
224.          bus_ride(leen,bno)(licn,brd′) end
225.     (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
225s         l_to_b[l,b]!BusΣMsg(t,pos);
225         l_to_b[l,b]!BusHistΣMsg(licn,bno);
225         l_to_b[l,b]!FreeΣ_ActΣMsg(licn,bno) ;
225          bus_ride(leen,bno)(ilicn,″nil″),
226.     (⟨(t,pos),(t′,bs′)⟩⌢brd′,mkAtBS(_,_,_,_)) →
226s         l_to_b[l,b]!BusΣMsg(t,pos) ;
226          bus_ride(licn,bno)(⟨(t′,bs′)⟩⌢brd′),
227.     _ → bus_ride(leen,bno)(licn,brd) end end end
```

669 In formula line 222 of **bus_ride** we obtained the **bus**. But we did not use "that" bus ! We we may wish to record, somehow, number of passengers alighting and boarding at bus stops, bus fees paid, one way or another, etc. The **bus**, which is a time-dependent entity, gives us that information. Thus we can revise formula lines 225s and 226s:

```
Simple:   225s  l_to_b[l,b]!BusΣMsg(pos);
Revised:  225r  l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));

Simple:   226s  l_to_b[l,b]!BusΣMsg(pos);
Revised:  226r  l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));
```

```
type
   Bus_Info = Passengers × Passengers × Cash × ...
value
   bus_info: Bus → Bus_Info
   bus_info(bus) ≡ (obs_alighted(bus),obs_boarded(bus),obs_till(bus),...)
```

It is time to discuss our description (here we choose the **bus_ride** behaviour) in the light of our claim of modeling "the domain". These are our comments:

- First one should recognise, i.e., be reminded, that the narrative and formal descriptions are always abstractions. That is, they leave out few or many things. We, you and I, shall never be able to describe everything there is to describe about even the simplest entity, operation, event or behaviour.

•

•

•

670

The Global Time Behaviour

228. The time_clock is a never ending behaviour — started at some time t_0.

229. The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.

230. At any moment the time_clock behaviour may not be inquired.

231. After a skip of the clock or an inquiry the time_clock behaviour continues, non-deterministically either maintaining the time or advancing the clock!

671

```
value
228. time_clock: T →
228.     in,out {l_to_t[ leen ] | leen:LeeNm • leen ∈ leenms}
228.     in,out {b_to_t[ bno ] | bno:BusNo • bno ∈ busnos}  Unit
228. time_clock:(t) ≡
230.   (skip ⌈⌉
229.   (⌷{l_to_t[ leen ]? ; l_to_t[ leen ]!t | leen:LeeNm•leen ∈ leens})
229.   ⌈⌉ (⌷{b_to_t[ bno ]? ; b_to_t[ bno ]!t | bno:BusNo•bno ∈ busnos})) ;
231.   (time_clock:(t) ⌈⌉ time_clock(t+δ_t))
```

672

The Bus Traffic Behaviour

232. There is a single bus_traffic behaviour. It is, “mysteriously”, given a constant argument, “the” traffic, tr.

233. At any moment it is ready to inform of the position, bps(b), of a bus, b, assumed to be in the traffic at time t.

234. The request for a bus position comes from some bus.

235. The bus positions are part of the traffic at time t.

236. The bus_traffic behaviour, after informing of a bus position reverts to “itself”.

673

```
value
232. bus_traffic: TR → in,out {b_to_tr[ bno ]|bno:BusNo•bno ∈ busnos} Unit
232. bus_traffic(tr) ≡
234.   ⌷ { let reqBusAndPos(bno,time) = b_to_tr[ b ]? in assert b=bno
233.        if time ∉ dom tr then chaos else
235.        let (_,bps) = tr(t) in
233.        if bno ∉ dom tr(t) then chaos else
233.        b_to_tr[ bno ]!bps(bno) end end end end | b:BusNo•b ∈ busnos} ;
236.   bus_traffic(tr)
```

674

License Operations

237. The lic_ops function models the contract holder choosing between and performing licensed operations.

     We remind the reader of the four actions that licensed operations may give rise to; cf. the abstract syntax of actions, Page 34.

238. To perform any licensed operation the sub-contractor needs to know the time and

239. must choose amongst the four kinds of operations that are licensed. The choice function, which we do not define, makes a basically non-deterministic choice among licensed alternatives. The choice yields the contract number of a received contract and, based on its set of licensed operations, it yields either a simple action or a sub-contracting action.

240. Thus there is a case distinction amongst four alternatives.

241. This case distinction is expressed in the four lines identified by: 241.

242. All the auxiliary functions, besides the action arguments, require the same state arguments.

675

```
value
237. lic_ops: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
237. lic_ops(leen)(licσ,busσ) ≡
238.   let t = (time_channel(leen)!req_Time;time_channel(leen)?) in
239.   let (licn,act) = choice(licσ)(busσ)(t) in
240.   (case act of
241.      mkCon(blid,bid)  → cndct(licn,leenm,t,act),
241.      mkCan(blid,bid)   → cancl(licn,leenm,t,act),
241.      mkIns(blid,bid)   → insrt(licn,leenm,t,act),
241.      mkLic(leenm′,bo) → sublic(licn,leenm,t,act) end)(licσ,busσ) end end

        cndct,cancl,insert: SmpAct→(LicΣ×BusΣ)→(LicΣ×BusΣ)
        sublic: SubLic→(LicΣ×BusΣ)→(LicΣ×BusΣ)
```

676

Bus Monitoring

Like for the **bus_ride** behaviour we decompose the **bus_mon**itoring behaviour into two behaviours. The **local_bus_mon**itoring behaviour monitors the buses that are commissioned by the sub-contractor. The **licensor_bus_mon**itoring behaviour monitors the buses that are commissioned by sub-contractors sub-contractd by the contractor.

```
value
   bus_mon: l:LeeNm → (LicΣ×BusΣ)
               → in {l_to_b[ l,b ]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
   bus_mon(l)(licσ,busσ) ≡
       local_bus_mon(l)(licσ,busσ) ⌈⌉ licensor_bus_mon(l)(licσ,busσ)
```

677

243. The **local_bus_mon**itoring function models all the interaction between a contract holder and its despatched buses.

244. We show only the communications from buses to contract holders.

245.

246.

247.

248.

249.

250.

251.

252.

253.

678

```
243. local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
244.         → in {l_to_b[ leen,b ]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
243. local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
245.    let (bno,msg) = ⌈⌉{(b,l_to_b[ l,b ]?)|b:BNo•b ∈ allbuses(leen)} in
249.    let (blid,bid,licn,lorn,trace) = abσ(bno) in
246.    case msg of
247.       BusΣMsg(t,bs) →
```

```
251.        let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
251.        (licσ,(fbσ,abσ′,histσ)) end,
253.      BusHistΣMsg(licn,bno) →
253.        let lbσ′ =
253.           update_LorBusΣ(obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))(lbσ)  in
253.        l_to_l[ leen,obs_LorNm(licn) ]!Licensor_BusHistΣMsg(licn,leen,bno,blid,bid,tr);
253.        ((rlσ,slσ,lbσ′),busσ) end
252.      FreeΣ_ActΣMsg(licn,bno) →
253.        let (fbσ′,abσ′) = update_FreeΣ_ActΣ(bno,bs)(fbσ,abσ) in
253.        (licσ,(fbσ′,abσ′)) end
253.    end end end
```

679

254.

255.

256.

257.

258.

680

```
254. licensor_bus_mon: lorn:LorNm → (LicΣ×BusΣ)
254.        → in {l_to_l[ lorn,leen ]|leen:LeeNm•leen ∈ leenms\{lorn}} (LicΣ×BusΣ)
254. licensor_bus_mon(lorn)(licσ,busσ) ≡
254.    let (rlσ,slσ,lbhσ) = licσ in
254.    let (leen,Licensor_BusHistΣMsg(licn,leen″,bno,blid,bid,tr))
                                   = ⌷{(leen′,l_to_l[ lorn,leen′ ]?)|leen′:LeeNm•leen′ ∈ leenms\{lorn}} in
254.    let lbhσ′ =
254.        update_BusHistΣ(obs_LorNm(licn),licn,leen″,(blid,bid),(bno,trace))(lbhσ) in
254.    l_to_l[ leenm,obs_LorNm(licnm) ]!Licensor_BusHistΣMsg(b,blid,bid,lin,lee,tr);
254.    ((rlσ,slσ,lbhσ′),busσ)
254.    end end end
```

681

License Negotiation

259.

260.

261.

262.

263.

264.

265.

266.

267.

268.

269.

270.

682

259.
260.
261.
262.
263.
264.
265.
266.
267.
268.
269.

683

The Conduct Bus Ride Action

271. The conduct bus ride action prescribed by (ln,mkCon(bli,bi,t′) takes place in a context and shall have the following effect:

   a) The action is performed by contractor li and at time t. This is known from the context.

   b) First it is checked that the timetable in the contract named ln does indeed provide a journey, j, indexed by bli and (then) bi, and that that journey starts (approximately) at time t′ which is the same as or later than t.

   c) Being so the action results in the contractor, whose name is "embedded" in ln, receiving notification of the bus ride commitment. 684

d) Then a bus, selected from a pool of available buses at the bust stop of origin of journey j, is given j as its journey script, whereupon that bus, as a behaviour separate from that of sub-contractor li, commences its ride.

e) The bus is to report back to sub-contractor li the times at which it stops at en route bus stops as well as the number (and kind) of passengers alighting and boarding the bus at these stops.

f) Finally the bus reaches its destination, as prescribed in j, and this is reported back to sub-contractor li.

g) Finally sub-contractor li, upon receiving this 'end-of-journey' notification, records the bus as no longer in actions but available at the destination bus stop.

685

271.
271a)
271b)
271c)
271d)
271e)
271f)
271g)

686

The Cancel Bus Ride Action

272. The cancel bus ride action prescribed by (ln,mkCan(bli,bi,t′)) takes place in a context and shall have the following effect:

a) The action is performed by contractor li and at time t. This is known from the context.

b) First a check like that prescribed in Item 271b) is performed.

c) If the check is OK, then the action results in the contractor, whose name is "embedded" in ln, receiving notification of the bus ride cancellation.

That's all !

687

272.
272a)
272b)
272c)

688

The Insert Bus Ride Action

273. The insert bus ride action prescribed by (ln,mkIns(bli,bi,t′) takes place in a context and shall have the following effect:

   a) The action is performed by contractor li and at time t. This is known from the context.

   b) First a check like that prescribed in Item 271b) is performed.

   c) If the check is OK, then the action results in the contractor, whose name is "embedded" in ln, receiving notification of the new bus ride commitment.

   d) The rest of the effect is like that prescribed in Items 271d)–271g).

689

273.
273a)
273b)
273c)
273d)

690

The Contracting Action

274. The subcontracting action prescribed by (ln,mkLic(li′,(pe′,ops′,tt′))) takes place in a context and shall have the following effect:

   a) The action is performed by contractor li and at time t. This is known from the context.

   b) First it is checked that timetable tt is a subset of the timetable contained in, and that the operations ops are a subset of those granted by, the contract named ln.

   c) Being so the action gives rise to a contract of the form (ln′,li,(pe′,ops′,tt′),li′). ln′ is a unique new contract name computed on the basis of ln, li, and t. li′ is a sub-contractor name chosen by contractor li. tt′ is a timetable chosen by contractor li. ops′ is a set of operations likewise chosen by contractor li.

   d) This contract is communicated by contractor li to sub-contractor li′.

   e) The receipt of that contract is recorded in the license state.

   f) The fact that the contractor has sublicensed part (or all) of its obligation to conduct bus rides is recorded in the modified component of its received contracts.

691

274.
274a)
274b)
274c)
274d)
274e)
274f)

692

L.3 Principles $\boxed{\text{BLANK}}$ 693

L.4 Discussion $\boxed{\text{BLANK}}$ 694

L.5 Research Challenges $\boxed{\text{BLANK}}$ 695

M Domain Management and Organisation 696

M.1 Definition

By the **management** of an enterprise (an institution) we shall understand a (possibly **stratified**, see ‘organisation’ next) set of enterprise staff (behaviours, processes) **authorised** to perform certain **functions** not allowed performed by other enterprise staff (behaviours, processes) and where such functions involve monitoring and controlling other enterprise staff (behaviours, processes). By **organisation** of an enterprise (an institution) we shall understand the **stratification** (partitioning) of enterprise staff (behaviours, processes) with each **partition** endowed with a set of **authorised functions** and with **communication interfaces** defined between partitions, i.e., between behaviours (processes).

M.2 An Abstraction of Management Functions 697

Let **E** designate some enterprise state concept, and let **stra_mgt, tact_mgt, oper_mgt, wrkr** and **merge** designate strategic management, tactical management, operational management and worker actions on states such that these actions are “somehow aware” of the state targets of respective management groups and or workers. Let **p** be a predicate which determines whether a given target state has been reached, and let **merge** harmonise different state targets into an agreeable one. Then the following behaviour reflects some aspects of management.

```
type
   E
value
   stra_mgt, tact_mgt, oper_mgt, wrkr, merge: E×E×E×E → E
   p: E* → Bool
   mgt: E → E
   mgt(e) ≡
      let e′ = stra_mgt(e,e″,e‴,e⁗),
          e″ = tact_mgt(e,e″,e‴,e⁗),
          e‴ = oper_mgt(e,e″,e‴,e⁗),
          e⁗ = wrkr(e,e″,e‴,e⁗) in
      if p(e,e″,e‴,e⁗)
          then skip
          else mgt(merge(e,e″,e‴,e⁗))
      end end
```

The recursive set of $e^{\prime\cdot\cdot\prime} = f(e, e'', e''', e'''')$ equations are “solved” by iterative communication between the management groups and the workers. The arrangement of these equations reflect the organisation and the various functions, **stra_mgt, tact_mgt, oper_mgt** and **wrkr** reflect the management. The frequency of communication between the management groups and the workers help determine a quality of the result.

The above is just a very crude, and only an illustrative model of management and organisation.

We could also have given a generic model, as the above, of management and organisation but now in terms of, say, CSP processes. Individual managers are processes and so are workers. The enterprise state, $e : E$, is maintained by one or more processes, separate from manager and worker processes. Etcetera.

M.3 Research Challenges 698

We made no explicit references to such "business school of administration" "BA101" topics as 'strategic' and 'tactical' management. Contemplate the types of entities and signatures of functions related to executive, strategic, tactical and operational management and organisation matters given on Page 35. Come up with better or other proposals, and/or attempt clear, but not necessarily computable predicates which (help) determine whether an operation (above they are alluded to as 'stra' and 'tact') is one of strategic or of tactical concern.

N Domain Human Behaviour BLANK 699

N.1 Definitions

N.2 A Formal Characterisation of Human Behaviour 700

N.3 Examples

701

N.4 Research Challenge

702

O Business Process Re-Engineering (BPR) $\boxed{\text{BLANK}}$ 703

O.1 Definitions

O.2 Facet-Orientations 704

O.2.1 Intrinsics BPR 705

O.2.2 Support Technology BPR 706

O.2.3 Rules and Regulations BPR 707

O.2.4 Script, Contract or License BPR 708

O.2.5 Management and Organisation BPR 709

O.2.6 Human Behaviour BPR 710

O.3 Research Challenge

711

P Domain Requirements Projection BLANK 712

P.1 Definitions

P.2 Examples

713

P.3 Research Challenge 714

Q Domain Requirements Instantiation BLANK 715

Q.1 Definitions

Q.2 Examples 716

Q.3 Research Challenge 717

R Domain Requirements Determination BLANK 718

R.1 Definitions

R.2 Examples

719

R.3 Research Challenge

720

S Domain Requirements Extension BLANK 721

S.1 Definitions

S.2 Examples 722

S.3 Research Challenge 723

T Domain Requirements Fitting BLANK 724

T.1 Definitions

T.2 Examples

725

T.3 Research Challenge 726

U Interface Requirements $\boxed{\text{BLANK}}$ 727

U.1 Definitions

U.2 Shared Phenomena and Concepts 728

U.3 Ontology-Oriented Interfaces 729

U.3.1 Shared Entities 730

U.3.2 Shared Actions 731

U.3.3 Shared Events 732

U.3.4 Shared Behaviours 733

U.4 Examples

734

U.5 Research Challenge 735

Part V
Three Example Domain Descriptions

V The Tokyo Stock Exchange

V.1 Introduction

This chapter shall try describe: narrate and formalise some facets of the (now "old"[40]) stock trading system of the TSE: Tokyo Stock Exchange (especially the 'matching' aspects).

V.2 The Problem

The reason that I try tackle a description (albeit of the "old" system) is that Prof. Tetsuo Tamai published a delightful paper [81, IEEE Computer Journal, June 2009 (vol. 42 no. 6) pp. 58-65)], Social Impact of Information Systems, in which a rather sad story is unfolded: a human error key input: an offer for selling stocks, although "ridiculous" in its input data (*"sell 610 thousand stocks, each at one (1) Japanese Yen"*, whereas one stock at 610,000 JPY was meant), and although several immediate — within seconds — attempts to cancel this "order", could not be cancelled ! This lead to a loss for the selling broker at around 42 Billion Yen, at today's exchange rate, 26 Jan. 2010, 469 million US $s ![41] Prof. Tetsuo Tamai's paper gives a, to me, chilling account of what I judge as an extremely sloppy and irresponsible design process by TSE and Fujitsu. It also leaves, I think, a strong impression of arrogance on the part of TSE. This arrogance, I claim, is still there in the documents listed in Footnote 40.

So the problem is a threefold one of

- **Proper Requirements:** How does one (in this case a stock exchange) prescribe (to the software developer) what is required by an appropriate hardware/software system for, as in this case, stock handling: acceptance of buy bids and sell offers, the possible withdrawal (or cancellation) of such submitted offers, and their matching (i.e., the actual trade whereby buy bids are marched in an appropriate, clear and transparent manner).
- **Correctness of Implementation:** How does one make sure that the software/hardware system meets customers' expectations.

[40] We write "old" since, as of January 4, 2010, that 'old' stock trading system has been replaced by the so-called arrowhead system. We refer to the following documents:

- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet.html`
- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet-e.pdf`
- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet1e.pdf`
- `http://www.tse.or.jp/english/rules/equities/arrowhead/pamphlet2e.html`

[41] So far three years of law court case hearing etc., has, on Dec. 4, 2009, resulted in complainant being awarded 10.7 billion Yen in damages. See `http://www.ft.com/cms/s/0/e9d89050-e0d7-11de-9f58--00144feab49a.html`.

- **Proper Explanation to Lay Users:** How does one explain, to the individual and institutional customers of the stock exchange, those offering stocks for sale of bids for buying stocks – how does one explain – in a clear and transparent manner the applicable rules governing stock handling.[42]

I shall only try contribute, in this document, to a solution to the first of these sub-problems.

V.3 A Domain Description

V.3.1 Market and Limit Offers and Bids

1. A market sell offer or buy bid specifies
   a) the unique identification of the stock,
   b) the number of stocks to be sold or bought, and
   c) the unique name of the seller.
2. A limit sell offer or buy bid specifies the same information as a market sell offer or buy bid (i.e., Items 1a–1c), and
   d) the price at which the identified stock is to be sold or bought.
3. A trade order is either a (mkMkt marked) market order or (mkLim marked) a limit order.
4. A trading command is either a sell order or a buy bid.
5. The sell orders are made unique by the mkSell "make" function.
6. The buy orders are made unique by the mkBuy "make" function.

type
```
 1  Market = Stock_id × Number_of_Stocks × Name_of_Customer
  1a    Stock_id
 1b    Number_of_Stocks = {|n•n:Nat∧n≥1|}
  1c    Name_of_Customer
  2  Limit = Market × Price
 2d    Price = {|n•n:Nat∧n≥1|}
  3  Trade == mkMkt(m:Market) | mkLim(l:Limit)
  4  Trading_Command = Sell_Order | Buy_Bid
  5  Sell_Order == mkSell(t:Trade)
  6  Buy_Bid == mkBuy(t:Trade)
```

[42] The rules as explained in the Footnote 40 on the facing page listed documents are far from clear and transparent: they are full of references to fast computers, overlapping processing, etc., etc.: matters with which these buying and selling customers should not be concerned — so, at least, thinks this author !

V.3.2 Order Books

7. We introduce a concept of linear, discrete time.
8. For each stock the stock exchange keeps an order book.
9. An order book for stock $s_{id} : SI$ keeps track of limit buy bids and limit sell offers (for the *id*entified *s*tock, s_{id}), as well as the market buy bids and sell offers; that is, for each price
   - d) the number of stocks, designated by unique order number, offered for sale at that price, that is, limit sell orders, and
   - e) the number of stocks, by unique order number, bid for buying at that price, that is, limit buy bid orders;
   - f) if an offer is a market sell offer, then the number of stocks to be sold is recorded, and if an offer is a market buy bid (also an offer), then the number of stocks to be bought is recorded,
10. Over time the stock exchange displays a series of full order books.
11. A trade unit is a pair of a unique order number and an amount (a number larger than 0) of stocks.
12. An amount designates a number of one or more stocks.

type

7 T, On [Time, Order number]

8 All_Stocks_Order_Book = Stock_Id $\underset{m}{\rightarrow}$ Stock_Order_Book

9 Stock_Order_Book = (Price $\underset{m}{\rightarrow}$ Orders) × Market_Offers

9 Orders:: so:Sell_Orders × bo:Buy_Bids

9d Sell_Orders = On $\underset{m}{\rightarrow}$ Amount

9e Buy_Bids = On $\underset{m}{\rightarrow}$ Amount

9f Market_Offers :: mkSell(n:**Nat**) × mkBuy(n:**Nat**)

10 TSE = T $\underset{m}{\rightarrow}$ All_Stocks_Order_Book

11 TU = On × Amount

12 Amount = {|n•**Nat**∧n≥1|}

V.3.3 Aggregate Offers

13. We introduce the concepts of aggregate sell and buy orders for a given stock at a given price (and at a given time).
14. The aggregate sell orders for a given stock at a given price is
   - g) the stocks being market sell offered and

   h) the number of stocks being limit offered for sale at that price or lower

15. The aggregate buy bids for a given stock at a given price is

   i) including the stocks being market bid offered and

   j) the number of stocks being limit bid for buying at that price or higher

```
value
  14  aggr_sell: All_Stocks_Order_Book × Stock_Id × Price → Nat
  14  aggr_sell(asob,sid,p) ≡
  14    let ((sos,_),(mkSell(ns),_)) = asob(sid) in
  14g     ns +
  14h     all_sell_summation(sos,p) end
  15  aggr_buy: All_Stocks_Order_Book × Stock_Id × Price → Nat
  15  aggr_buy(asob,sid,p) ≡
  15    let ((_,bbs),(_,mkBuy(nb))) = asob(sid) in
  15i    nb +
  15j    nb + all_buy_summation(bbs,p) end

  all_sell_summation: Sell_Orders × Price → Nat
  all_sell_summation(sos,p) ≡
     let ps = {p′|p′:Prices • p′ ∈ dom sos ∧ p′ ≥ p} in accumulate(sos,ps)(0) end

  all_buy_summation: Buy_Bids × Price → Nat
  all_buy_summation(bbs,p) ≡
     let ps = {p′|p′:Prices • p′ ∈ dom bos ∧ p′ ≤ p} in accumulate(bbs,ps)(0) end
```

The auxiliary accumulate function is shared between the all_sell_summation and the all_buy_summation functions. It sums the amounts of limit stocks in the price range of the accumulate function argument ps. The auxiliary sum function sums the amounts of limit stocks — "pealing off" the their unique order numbers.

```
value
  accumulate: (Price →m Orders) × Price-set → Nat → Nat
  accumulate(pos,ps)(n) ≡
     case ps of {} → n, {p}∪ ps′ → accumulate(pos,ps′)(n+sum(pos(p)){dom pos(p)}) end

  sum: (Sell_Orders|Buy_Bids) → On-set → Nat
  sum(ords)(ns) ≡
     case ns of {} → 0, {n}∪ ns′ → ords(n)+sum(ords)(ns′) end
```

To handle the sub_limit_sells and sub_limit_buys indicated by Item 17c on the next page of the Itayose "algorithm" we need the corresponding sub_sell_summation and sub_buy_summation functions:

```
value
   sub_sell_summation: Stock_Order_Book × Price → Nat
   sub_sell_summation(((sos,_),(ns,_)),p) ≡ ns +
      let ps = {p′|p′:Prices • p′ ∈ dom sos ∧ p′ > p} in accumulate(sos,ps)(0) end

   sub_buy_summation: Stock_Order_Book × Price → Nat
   sub_buy_summation(((_,bbs),(_,nb)),p) ≡ nb +
      let ps = {p′|p′:Prices • p′ ∈ dom bos ∧ p′ < p} in accumulate(bbs,ps)(0) end
```

V.3.4 The TSE Itayose "Algorithm"

16. The TSE practices the so-called Itayose "algorithm" to decide on opening and closing prices[43]. That is, the Itayose "algorithm" determines a single so-called 'execution' price, one that matches sell and buy orders[44]:

17. The "matching sell and buy orders" rules:

    a) *All market orders must be 'executed'*[45].

    b) *All limit orders to sell/buy at prices higher/lower*[46] *than the 'execution price'*[47] *must be executed.*

    c) *The following amount of limit orders to sell or buy at the execution prices must be executed: the entire amount of either all sell or all buy orders, and at least one 'trading unit'*[48] *from 'the opposite side of the order book'*[49].

- The 28 January 2010 version had lines
  - $17c'_{\exists}$ name some_priced_buys, should have been, as now, some_priced_sells and
  - $17c''_{\forall}$ name all_priced_buys, should have been, as now, all_priced_sells.
- My current understanding of *and assumptions* about the TSE is
  - that each buy bid or sell order concerns a number, n, of one or more of the same kind of stocks (i.e. sid).
  - that each buy bid or sell order when being accepted by the TSE is assigned a unique order number *on*, and

[43] [81, pp 59, col. 1, lines 4-3 from bottom, cf. Page 271]

[44] [81, pp 59, col. 2, lines 1–3 and Items 1.–3. after yellow, four line 'insert', cf. Page 271] These items 1.–3. are reproduced as "our" Items 17a–17c.

[45] To execute an order: ?????

[46] Yes, it should be: "higher/lower"

[47] Execution price: ?????

[48] Trading unit: ?????

[49] The opposite side of the order book: ?????

  - that this is reflected in some Sell_Orders or Market_Bids entry being augmented.[50]

- For current (Monday 22 Feb., 2010) lack of a better abstraction[51], I have structured the Itayose "Algorithm" as follows:
  - $(17')$ either a match can be made based on
    - all buys and
    - some sells,
  - $(17'_{\vee})$ or
  - $(17'')$ a match can be made based on
    - aome buys and
    - all sells.

value

17 match: All_Stocks_Order_Book $\times$ Stock_Id $\rightarrow$ Price-**set**
17 match(asob,sid) **as** ps
17 **pre**: sid $\in$ **dom** asob
17 **post**: $\forall$ p':Price $\bullet$ $p' \in$ ps $\Rightarrow$
$17'$ all_buys_some_sells(p',ason,sid,ps) $\vee$
$17'_{\vee}$ $\vee$
$17''$ some_buys_all_sells(p',ason,sid,ps)

- $(17')$ The all_buys_some_sells part of the above disjunction "calculates" as follows:
  - The all_buys... part includes
    - all the market_buys
    - all the buys properly below the stated price, and
    - all the buys at that price.
  - The ...some_sells part includes
    - all the market_sells
    - all the sells properly below the stated price, and
    - some of the buys at that price.

$17'$ all_buys_some_sells(p',ason,sid,ps) $\equiv$
$17'$ $\exists$ os:On-**set** $\bullet$
$17a'$ all_market_buys(asob(sid))
$17b'$ + all_sub_limit_buys(asob(sid))(p')

[50] The present, 22.2.2010, model "lumps" all market orders. This simplification must be corrected, as for the Sell_Orders and Market_Bids, the Market_Offers must be modelled as are Orders.

[51] One that I am presently contemplating is based on another set of **pre/post** conditions.

```
17c′      + all_priced_buys(asob(sid))(p′)
17a′   =   all_market_sells(asob(sid))
17b′      + all_sub_limit_sells(asob(sid))(p′)
17c′_∃    + some_priced_sells(asob(sid))(p′)(os)
```

- (17″) As for the above, only "versed".

```
17″ some_buys_all_sells(p′,ason,sid,ps) ≡
17″   ∃ os:On-set •
17a″        all_market_buys(asob(sid))
17b″      + all_sub_limit_buys(asob(sid))(p′)
17c″      + some_priced_buys(asob(sid))(p′)(os)
17a″   =   all_market_sells(asob(sid))
17b″      + all_sub_limit_sells(asob(sid))(p′)
17c″_∀    + all_priced_sells(asob(sid))(p′) ∨
```

The match function calculates a set of prices for each of which a match can be made. The set may be empty: there is no price which satisfies the match rules (cf. Items 17a–17c below). The set may be a singleton set: there is a unique price which satisfies match rules Items 17a–17c. The set may contain more than one price: there is not a unique price which satisfies match rules Items 17a–17c. The single (′) and the double (″) quoted (17a–17c) group of lines, in the match formulas above, correspond to the Itayose "algorithm"s Item 17c *'opposite sides of the order book'* description. The existential quantification of a set of order numbers of lines 17′ and 17″ correspond to that "algorithms" (still Item 17c) point of *at least one 'trading unit'*. It may be that the **post** condition predicate is only fullfilled for all trading units – so be it.

```
value
   all_market_buys: Stock_Order_Book → Amount
   all_market_buys((_,(_,mkBuys(nb))),p) ≡ nb

   all_market_sells: Stock_Order_Book → Amount
   all_market_sells((_,(mkSells(ns),_)),p) ≡ ns

   all_sub_limit_buys: Stock_Order_Book → Price → Amount
   all_sub_limit_buys(((,bbs),_))(p) ≡ sub_buy_summation(bbs,p)

   all_sub_limit_sells: Stock_Order_Book → Price → Amount
   all_sub_limit_sells((sos,_))(p) ≡ sub_sell_summation(sos,p)

   all_priced_buys: Stock_Order_Book → Price → Amount
   all_priced_buys((_,bbs),_)(p) ≡ sum(bbs(p))
```

```
all_priced_sells: Stock_Order_Book → Price → Amount
all_priced_sells((sos,_),_)(p) ≡ sum(sos(p))

some_priced_buys: Stock_Order_Book → Price → On-set → Amount
some_priced_buys((_,bbs),_)(p)(os) ≡
   let tbs = bbs(p) in if {}≠os∧os⊆dom tbs then sum(tbs)(os) else 0 end end

some_priced_sells: Stock_Order_Book → Price → On-set → Amount
some_priced_sells((sos,_),_)(p)(os) ≡
   let tss = sos(p) in if {}≠os∧os⊆dom tss then sum(tss)(os) else 0 end end
```

The formalisation of the Itayise “algorithm”, as well as that “algorithm” [itself], does not guarantee a match where a match “ought” be possible. The “stumbling block” seems to be the Itayose “algorithm”s Item 17c. There it says: *‘at least one trading unit’*. We suggest that a match could be made in which some of the stocks of a candidate trading unit be matched with the remaining stocks also being traded, but now with the stock exchange being the buyer and with the stock exchange immediately “turning around” and posting those remaining stocks as a TSE marked trading unit for sale.

18. It seems to me that the Tetsuo Tamai paper does not really handle
    a) the issue of order numbers,
    b) therefore also not the issue of the number of stocks to be sold or bought per order number.
19. Therefore the Tetsuo Tamai paper does not really handle
    a) the situation where a match “only matches” part of a buy or a sell order.

Much more to come: essentially I have only modelled column 2, rightmost column, Page 59 of [81, Tetsuo Tamai, “TSE”]. Next to be modelled is column 1, leftmost column, Page 60 of [81]. See these same page numbers of the present document !

V.3.5 Match Executions

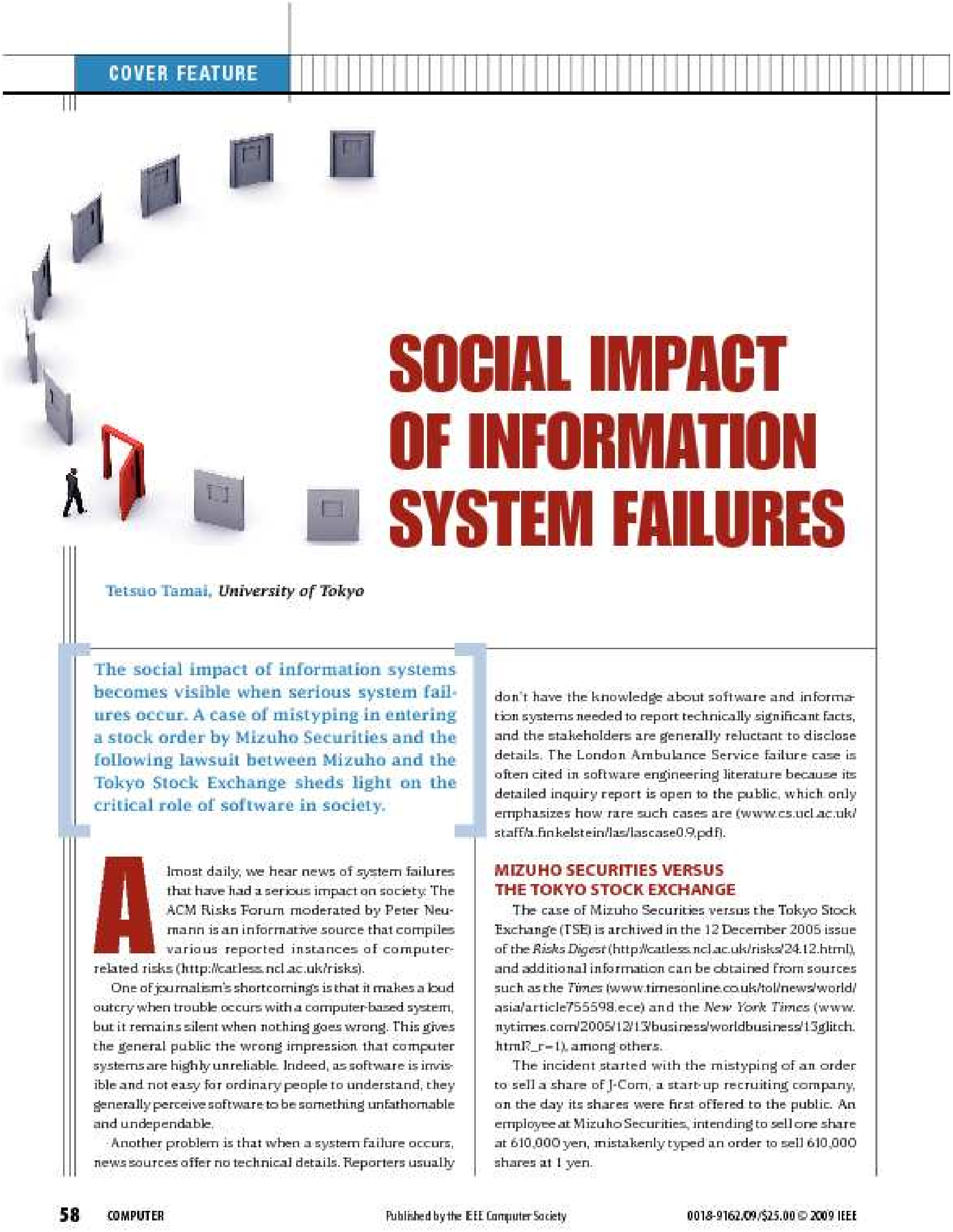
`to come`

V.3.6 Order Handling

`to come`

V.4 Tetsuo Tamai's Paper

For private, limited circulation only, I take the liberty of enclosing Tetsuo Tamai's IEEE Computer Journal paper.



COVER FEATURE

SOCIAL IMPACT OF INFORMATION SYSTEM FAILURES

Tetsuo Tamai, *University of Tokyo*

The social impact of information systems becomes visible when serious system failures occur. A case of mistyping in entering a stock order by Mizuho Securities and the following lawsuit between Mizuho and the Tokyo Stock Exchange sheds light on the critical role of software in society.

Almost daily, we hear news of system failures that have had a serious impact on society. The ACM Risks Forum moderated by Peter Neumann is an informative source that compiles various reported instances of computer-related risks (http://catless.ncl.ac.uk/risks).

One of journalism's shortcomings is that it makes a loud outcry when trouble occurs with a computer-based system, but it remains silent when nothing goes wrong. This gives the general public the wrong impression that computer systems are highly unreliable. Indeed, as software is invisible and not easy for ordinary people to understand, they generally perceive software to be something unfathomable and undependable.

Another problem is that when a system failure occurs, news sources offer no technical details. Reporters usually don't have the knowledge about software and information systems needed to report technically significant facts, and the stakeholders are generally reluctant to disclose details. The London Ambulance Service failure case is often cited in software engineering literature because its detailed inquiry report is open to the public, which only emphasizes how rare such cases are (www.cs.ucl.ac.uk/staff/a.finkelstein/las/lascase0.9.pdf).

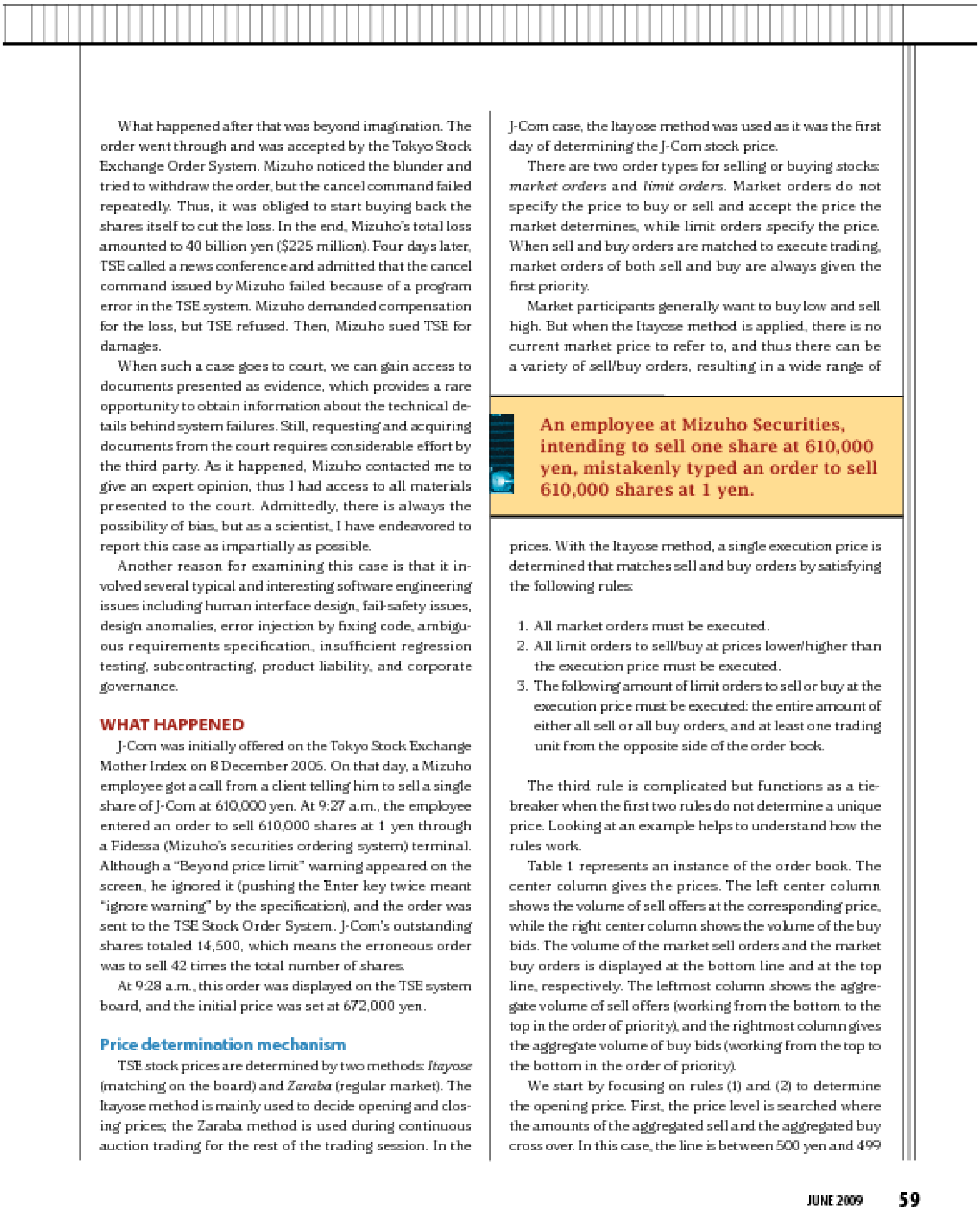
MIZUHO SECURITIES VERSUS THE TOKYO STOCK EXCHANGE

The case of Mizuho Securities versus the Tokyo Stock Exchange (TSE) is archived in the 12 December 2005 issue of the *Risks Digest* (http://catless.ncl.ac.uk/risks/24.12.html), and additional information can be obtained from sources such as the *Times* (www.timesonline.co.uk/tol/news/world/asia/article755598.ece) and the *New York Times* (www.nytimes.com/2005/12/13/business/worldbusiness/13glitch.html?_r=1), among others.

The incident started with the mistyping of an order to sell a share of J-Com, a start-up recruiting company, on the day its shares were first offered to the public. An employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly typed an order to sell 610,000 shares at 1 yen.

Published by the IEEE Computer Society

0018-9162/09/$25.00 © 2009 IEEE



What happened after that was beyond imagination. The order went through and was accepted by the Tokyo Stock Exchange Order System. Mizuho noticed the blunder and tried to withdraw the order, but the cancel command failed repeatedly. Thus, it was obliged to start buying back the shares itself to cut the loss. In the end, Mizuho's total loss amounted to 40 billion yen ($225 million). Four days later, TSE called a news conference and admitted that the cancel command issued by Mizuho failed because of a program error in the TSE system. Mizuho demanded compensation for the loss, but TSE refused. Then, Mizuho sued TSE for damages.

When such a case goes to court, we can gain access to documents presented as evidence, which provides a rare opportunity to obtain information about the technical details behind system failures. Still, requesting and acquiring documents from the court requires considerable effort by the third party. As it happened, Mizuho contacted me to give an expert opinion, thus I had access to all materials presented to the court. Admittedly, there is always the possibility of bias, but as a scientist, I have endeavored to report this case as impartially as possible.

Another reason for examining this case is that it involved several typical and interesting software engineering issues including human interface design, fail-safety issues, design anomalies, error injection by fixing code, ambiguous requirements specification, insufficient regression testing, subcontracting, product liability, and corporate governance.

WHAT HAPPENED

J-Com was initially offered on the Tokyo Stock Exchange Mother Index on 8 December 2005. On that day, a Mizuho employee got a call from a client telling him to sell a single share of J-Com at 610,000 yen. At 9:27 a.m., the employee entered an order to sell 610,000 shares at 1 yen through a Fidessa (Mizuho's securities ordering system) terminal. Although a "Beyond price limit" warning appeared on the screen, he ignored it (pushing the Enter key twice meant "ignore warning" by the specification), and the order was sent to the TSE Stock Order System. J-Com's outstanding shares totaled 14,500, which means the erroneous order was to sell 42 times the total number of shares.

At 9:28 a.m., this order was displayed on the TSE system board, and the initial price was set at 672,000 yen.

Price determination mechanism

TSE stock prices are determined by two methods: *Itayose* (matching on the board) and *Zaraba* (regular market). The Itayose method is mainly used to decide opening and closing prices; the Zaraba method is used during continuous auction trading for the rest of the trading session. In the J-Com case, the Itayose method was used as it was the first day of determining the J-Com stock price.

There are two order types for selling or buying stocks: *market orders* and *limit orders*. Market orders do not specify the price to buy or sell and accept the price the market determines, while limit orders specify the price. When sell and buy orders are matched to execute trading, market orders of both sell and buy are always given the first priority.

Market participants generally want to buy low and sell high. But when the Itayose method is applied, there is no current market price to refer to, and thus there can be a variety of sell/buy orders, resulting in a wide range of prices. With the Itayose method, a single execution price is determined that matches sell and buy orders by satisfying the following rules:

An employee at Mizuho Securities, intending to sell one share at 610,000 yen, mistakenly typed an order to sell 610,000 shares at 1 yen.

1. All market orders must be executed.
2. All limit orders to sell/buy at prices lower/higher than the execution price must be executed.
3. The following amount of limit orders to sell or buy at the execution price must be executed: the entire amount of either all sell or all buy orders, and at least one trading unit from the opposite side of the order book.

The third rule is complicated but functions as a tie-breaker when the first two rules do not determine a unique price. Looking at an example helps to understand how the rules work.

Table 1 represents an instance of the order book. The center column gives the prices. The left center column shows the volume of sell offers at the corresponding price, while the right center column shows the volume of the buy bids. The volume of the market sell orders and the market buy orders is displayed at the bottom line and at the top line, respectively. The leftmost column shows the aggregate volume of sell offers (working from the bottom to the top in the order of priority), and the rightmost column gives the aggregate volume of buy bids (working from the top to the bottom in the order of priority).

We start by focusing on rules (1) and (2) to determine the opening price. First, the price level is searched where the amounts of the aggregated sell and the aggregated buy cross over. In this case, the line is between 500 yen and 499

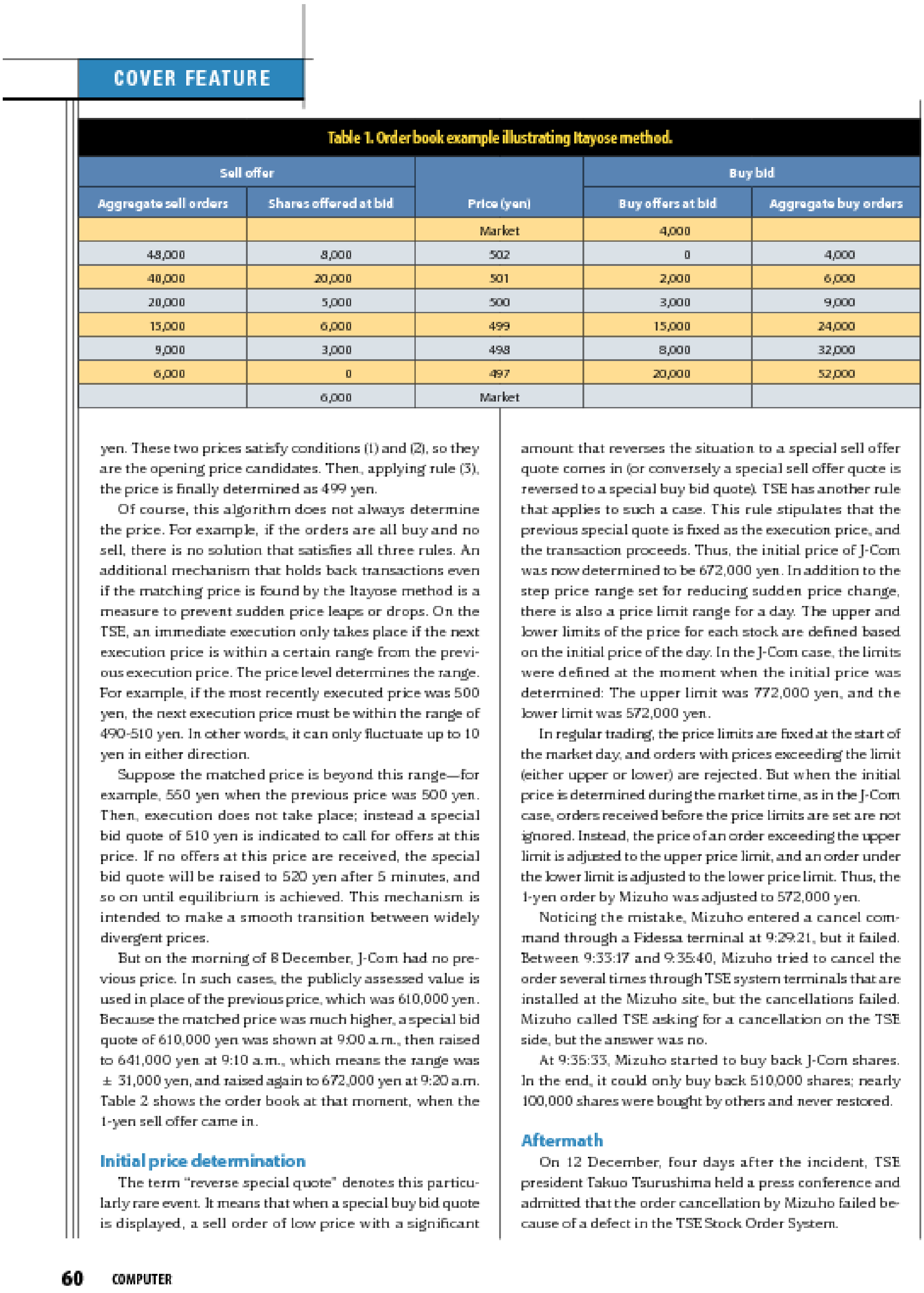


Table 1. Order book example illustrating Itayose method.

Sell offer			Buy bid	
Aggregate sell orders	Shares offered at bid	Price (yen)	Buy offers at bid	Aggregate buy orders
		Market	4,000	
48,000	8,000	502	0	4,000
40,000	20,000	501	2,000	6,000
20,000	5,000	500	3,000	9,000
15,000	6,000	499	15,000	24,000
9,000	3,000	498	8,000	32,000
6,000	0	497	20,000	52,000
	6,000	Market		

yen. These two prices satisfy conditions (1) and (2), so they are the opening price candidates. Then, applying rule (3), the price is finally determined as 499 yen.

Of course, this algorithm does not always determine the price. For example, if the orders are all buy and no sell, there is no solution that satisfies all three rules. An additional mechanism that holds back transactions even if the matching price is found by the Itayose method is a measure to prevent sudden price leaps or drops. On the TSE, an immediate execution only takes place if the next execution price is within a certain range from the previous execution price. The price level determines the range. For example, if the most recently executed price was 500 yen, the next execution price must be within the range of 490-510 yen. In other words, it can only fluctuate up to 10 yen in either direction.

Suppose the matched price is beyond this range—for example, 550 yen when the previous price was 500 yen. Then, execution does not take place; instead a special bid quote of 510 yen is indicated to call for offers at this price. If no offers at this price are received, the special bid quote will be raised to 520 yen after 5 minutes, and so on until equilibrium is achieved. This mechanism is intended to make a smooth transition between widely divergent prices.

But on the morning of 8 December, J-Com had no previous price. In such cases, the publicly assessed value is used in place of the previous price, which was 610,000 yen. Because the matched price was much higher, a special bid quote of 610,000 yen was shown at 9:00 a.m., then raised to 641,000 yen at 9:10 a.m., which means the range was ± 31,000 yen, and raised again to 672,000 yen at 9:20 a.m. Table 2 shows the order book at that moment, when the 1-yen sell offer came in.

Initial price determination

The term "reverse special quote" denotes this particularly rare event. It means that when a special buy bid quote is displayed, a sell order of low price with a significant amount that reverses the situation to a special sell offer quote comes in (or conversely a special sell offer quote is reversed to a special buy bid quote). TSE has another rule that applies to such a case. This rule stipulates that the previous special quote is fixed as the execution price, and the transaction proceeds. Thus, the initial price of J-Com was now determined to be 672,000 yen. In addition to the step price range set for reducing sudden price change, there is also a price limit range for a day. The upper and lower limits of the price for each stock are defined based on the initial price of the day. In the J-Com case, the limits were defined at the moment when the initial price was determined: The upper limit was 772,000 yen, and the lower limit was 572,000 yen.

In regular trading, the price limits are fixed at the start of the market day, and orders with prices exceeding the limit (either upper or lower) are rejected. But when the initial price is determined during the market time, as in the J-Com case, orders received before the price limits are set are not ignored. Instead, the price of an order exceeding the upper limit is adjusted to the upper price limit, and an order under the lower limit is adjusted to the lower price limit. Thus, the 1-yen order by Mizuho was adjusted to 572,000 yen.

Noticing the mistake, Mizuho entered a cancel command through a Fidessa terminal at 9:29:21, but it failed. Between 9:33:17 and 9:35:40, Mizuho tried to cancel the order several times through TSE system terminals that are installed at the Mizuho site, but the cancellations failed. Mizuho called TSE asking for a cancellation on the TSE side, but the answer was no.

At 9:35:33, Mizuho started to buy back J-Com shares. In the end, it could only buy back 510,000 shares; nearly 100,000 shares were bought by others and never restored.

Aftermath

On 12 December, four days after the incident, TSE president Takuo Tsurushima held a press conference and admitted that the order cancellation by Mizuho failed because of a defect in the TSE Stock Order System.

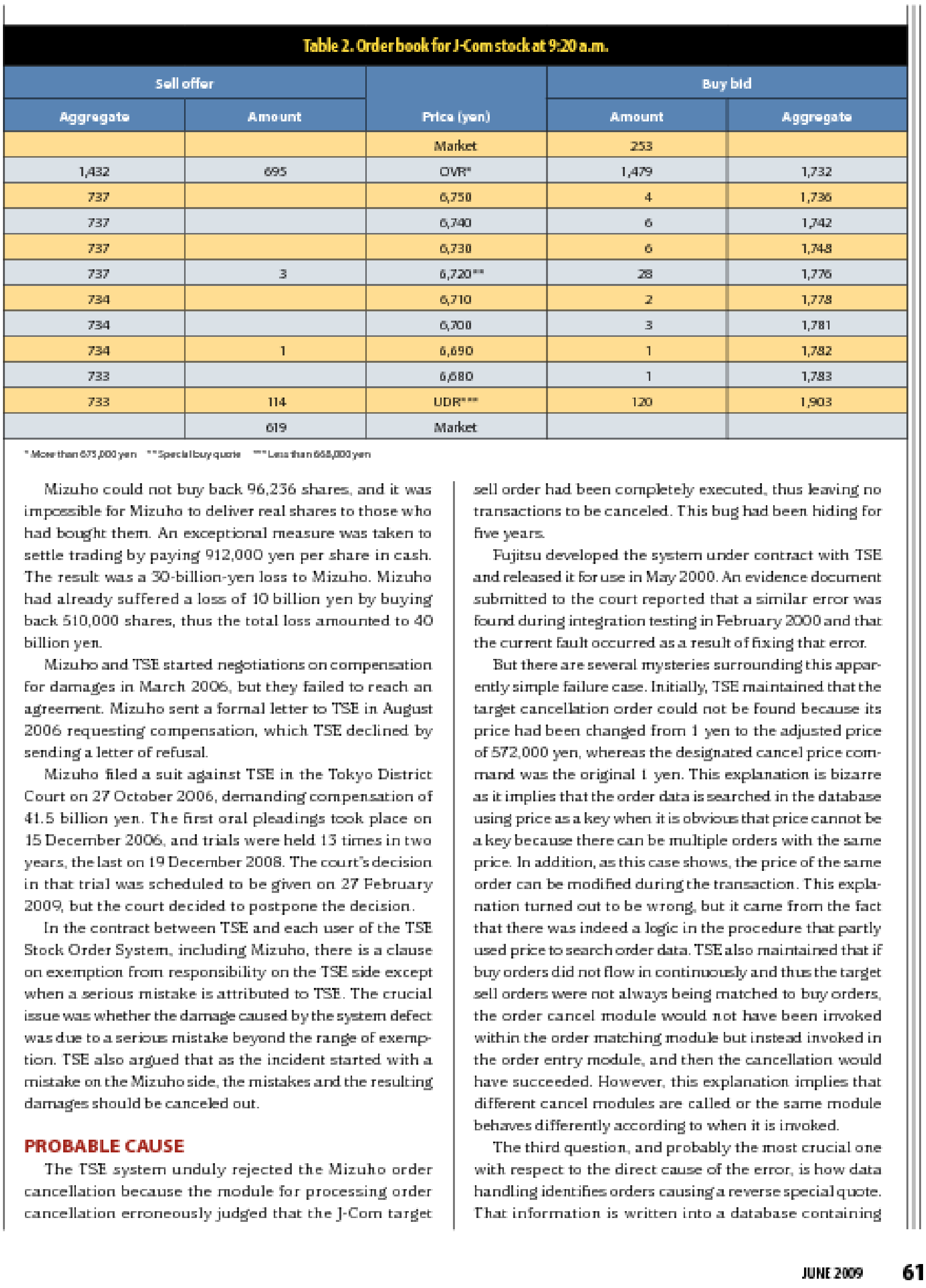


Table 2. Order book for J-Com stock at 9:20 a.m.

Sell offer			Buy bid	
Aggregate	Amount	Price (yen)	Amount	Aggregate
		Market	253	
1,432	695	OVR*	1,479	1,732
737		6,750	4	1,736
737		6,740	6	1,742
737		6,730	6	1,748
737	3	6,720**	28	1,776
734		6,710	2	1,778
734		6,700	3	1,781
734	1	6,690	1	1,782
733		6,680	1	1,783
733	114	UDR***	120	1,903
	619	Market		

* More than 673,000 yen ** Special buy quote *** Less than 668,000 yen

Mizuho could not buy back 96,236 shares, and it was impossible for Mizuho to deliver real shares to those who had bought them. An exceptional measure was taken to settle trading by paying 912,000 yen per share in cash. The result was a 30-billion-yen loss to Mizuho. Mizuho had already suffered a loss of 10 billion yen by buying back 510,000 shares, thus the total loss amounted to 40 billion yen.

Mizuho and TSE started negotiations on compensation for damages in March 2006, but they failed to reach an agreement. Mizuho sent a formal letter to TSE in August 2006 requesting compensation, which TSE declined by sending a letter of refusal.

Mizuho filed a suit against TSE in the Tokyo District Court on 27 October 2006, demanding compensation of 41.5 billion yen. The first oral pleadings took place on 15 December 2006, and trials were held 13 times in two years, the last on 19 December 2008. The court's decision in that trial was scheduled to be given on 27 February 2009, but the court decided to postpone the decision.

In the contract between TSE and each user of the TSE Stock Order System, including Mizuho, there is a clause on exemption from responsibility on the TSE side except when a serious mistake is attributed to TSE. The crucial issue was whether the damage caused by the system defect was due to a serious mistake beyond the range of exemption. TSE also argued that as the incident started with a mistake on the Mizuho side, the mistakes and the resulting damages should be canceled out.

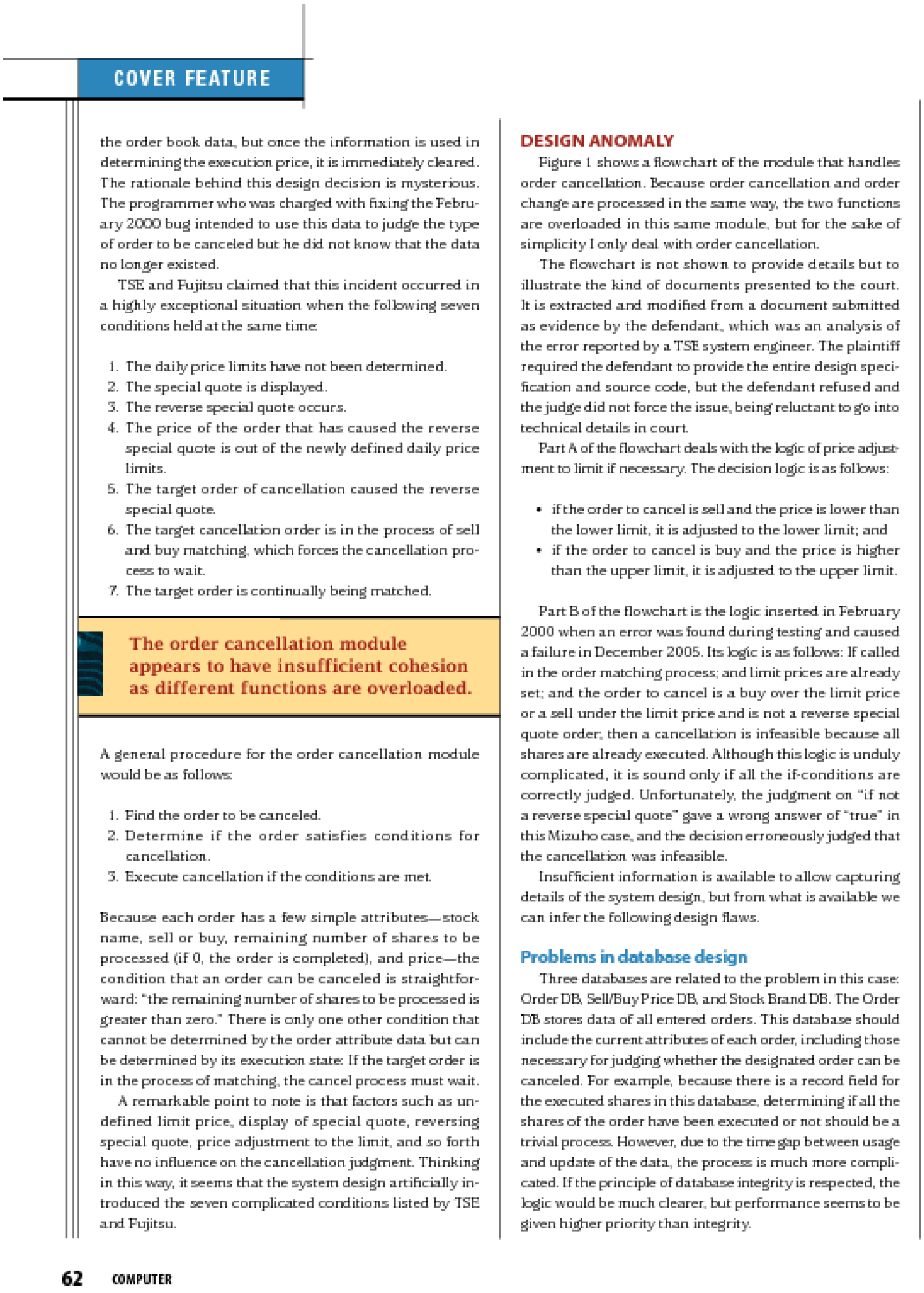
PROBABLE CAUSE

The TSE system unduly rejected the Mizuho order cancellation because the module for processing order cancellation erroneously judged that the J-Com target sell order had been completely executed, thus leaving no transactions to be canceled. This bug had been hiding for five years.

Fujitsu developed the system under contract with TSE and released it for use in May 2000. An evidence document submitted to the court reported that a similar error was found during integration testing in February 2000 and that the current fault occurred as a result of fixing that error.

But there are several mysteries surrounding this apparently simple failure case. Initially, TSE maintained that the target cancellation order could not be found because its price had been changed from 1 yen to the adjusted price of 572,000 yen, whereas the designated cancel price command was the original 1 yen. This explanation is bizarre as it implies that the order data is searched in the database using price as a key when it is obvious that price cannot be a key because there can be multiple orders with the same price. In addition, as this case shows, the price of the same order can be modified during the transaction. This explanation turned out to be wrong, but it came from the fact that there was indeed a logic in the procedure that partly used price to search order data. TSE also maintained that if buy orders did not flow in continuously and thus the target sell orders were not always being matched to buy orders, the order cancel module would not have been invoked within the order matching module but instead invoked in the order entry module, and then the cancellation would have succeeded. However, this explanation implies that different cancel modules are called or the same module behaves differently according to when it is invoked.

The third question, and probably the most crucial one with respect to the direct cause of the error, is how data handling identifies orders causing a reverse special quote. That information is written into a database containing



the order book data, but once the information is used in determining the execution price, it is immediately cleared. The rationale behind this design decision is mysterious. The programmer who was charged with fixing the February 2000 bug intended to use this data to judge the type of order to be canceled but he did not know that the data no longer existed.

TSE and Fujitsu claimed that this incident occurred in a highly exceptional situation when the following seven conditions held at the same time:

1. The daily price limits have not been determined.
2. The special quote is displayed.
3. The reverse special quote occurs.
4. The price of the order that has caused the reverse special quote is out of the newly defined daily price limits.
5. The target order of cancellation caused the reverse special quote.
6. The target cancellation order is in the process of sell and buy matching, which forces the cancellation process to wait.
7. The target order is continually being matched.

The order cancellation module appears to have insufficient cohesion as different functions are overloaded.

A general procedure for the order cancellation module would be as follows:

1. Find the order to be canceled.
2. Determine if the order satisfies conditions for cancellation.
3. Execute cancellation if the conditions are met.

Because each order has a few simple attributes—stock name, sell or buy, remaining number of shares to be processed (if 0, the order is completed), and price—the condition that an order can be canceled is straightforward: "the remaining number of shares to be processed is greater than zero." There is only one other condition that cannot be determined by the order attribute data but can be determined by its execution state: If the target order is in the process of matching, the cancel process must wait.

A remarkable point to note is that factors such as undefined limit price, display of special quote, reversing special quote, price adjustment to the limit, and so forth have no influence on the cancellation judgment. Thinking in this way, it seems that the system design artificially introduced the seven complicated conditions listed by TSE and Fujitsu.

DESIGN ANOMALY

Figure 1 shows a flowchart of the module that handles order cancellation. Because order cancellation and order change are processed in the same way, the two functions are overloaded in this same module, but for the sake of simplicity I only deal with order cancellation.

The flowchart is not shown to provide details but to illustrate the kind of documents presented to the court. It is extracted and modified from a document submitted as evidence by the defendant, which was an analysis of the error reported by a TSE system engineer. The plaintiff required the defendant to provide the entire design specification and source code, but the defendant refused and the judge did not force the issue, being reluctant to go into technical details in court.

Part A of the flowchart deals with the logic of price adjustment to limit if necessary. The decision logic is as follows:

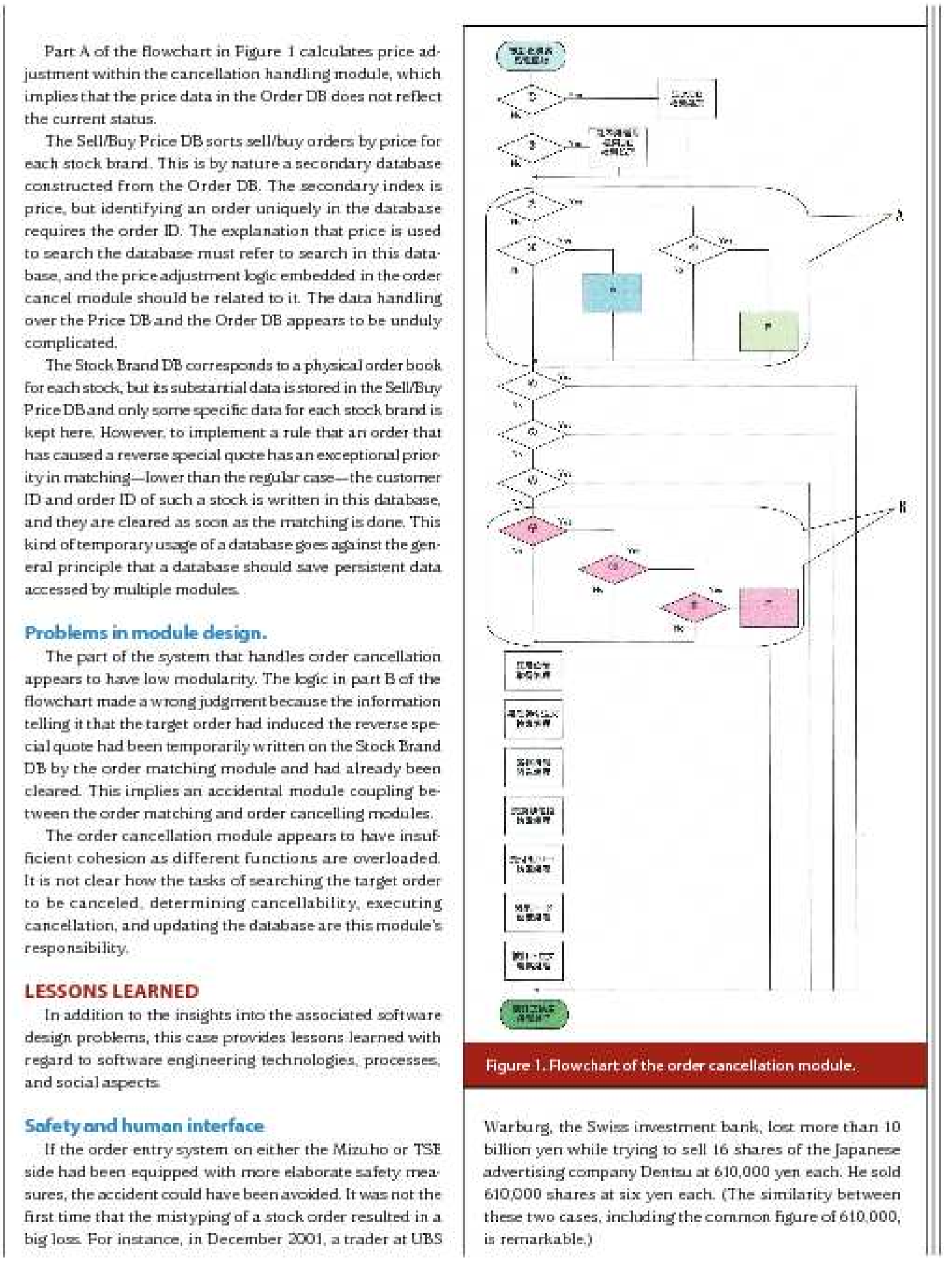
- if the order to cancel is sell and the price is lower than the lower limit, it is adjusted to the lower limit; and
- if the order to cancel is buy and the price is higher than the upper limit, it is adjusted to the upper limit.

Part B of the flowchart is the logic inserted in February 2000 when an error was found during testing and caused a failure in December 2005. Its logic is as follows: If called in the order matching process; and limit prices are already set; and the order to cancel is a buy over the limit price or a sell under the limit price and is not a reverse special quote order; then a cancellation is infeasible because all shares are already executed. Although this logic is unduly complicated, it is sound only if all the if-conditions are correctly judged. Unfortunately, the judgment on "if not a reverse special quote" gave a wrong answer of "true" in this Mizuho case, and the decision erroneously judged that the cancellation was infeasible.

Insufficient information is available to allow capturing details of the system design, but from what is available we can infer the following design flaws.

Problems in database design

Three databases are related to the problem in this case: Order DB, Sell/Buy Price DB, and Stock Brand DB. The Order DB stores data of all entered orders. This database should include the current attributes of each order, including those necessary for judging whether the designated order can be canceled. For example, because there is a record field for the executed shares in this database, determining if all the shares of the order have been executed or not should be a trivial process. However, due to the time gap between usage and update of the data, the process is much more complicated. If the principle of database integrity is respected, the logic would be much clearer, but performance seems to be given higher priority than integrity.



Part A of the flowchart in Figure 1 calculates price adjustment within the cancellation handling module, which implies that the price data in the Order DB does not reflect the current status.

The Sell/Buy Price DB sorts sell/buy orders by price for each stock brand. This is by nature a secondary database constructed from the Order DB. The secondary index is price, but identifying an order uniquely in the database requires the order ID. The explanation that price is used to search the database must refer to search in this database, and the price adjustment logic embedded in the order cancel module should be related to it. The data handling over the Price DB and the Order DB appears to be unduly complicated.

The Stock Brand DB corresponds to a physical order book for each stock, but its substantial data is stored in the Sell/Buy Price DB and only some specific data for each stock brand is kept here. However, to implement a rule that an order that has caused a reverse special quote has an exceptional priority in matching—lower than the regular case—the customer ID and order ID of such a stock is written in this database, and they are cleared as soon as the matching is done. This kind of temporary usage of a database goes against the general principle that a database should save persistent data accessed by multiple modules.

Problems in module design.

The part of the system that handles order cancellation appears to have low modularity. The logic in part B of the flowchart made a wrong judgment because the information telling it that the target order had induced the reverse special quote had been temporarily written on the Stock Brand DB by the order matching module and had already been cleared. This implies an accidental module coupling between the order matching and order cancelling modules.

The order cancellation module appears to have insufficient cohesion as different functions are overloaded. It is not clear how the tasks of searching the target order to be canceled, determining cancellability, executing cancellation, and updating the database are this module's responsibility.

LESSONS LEARNED

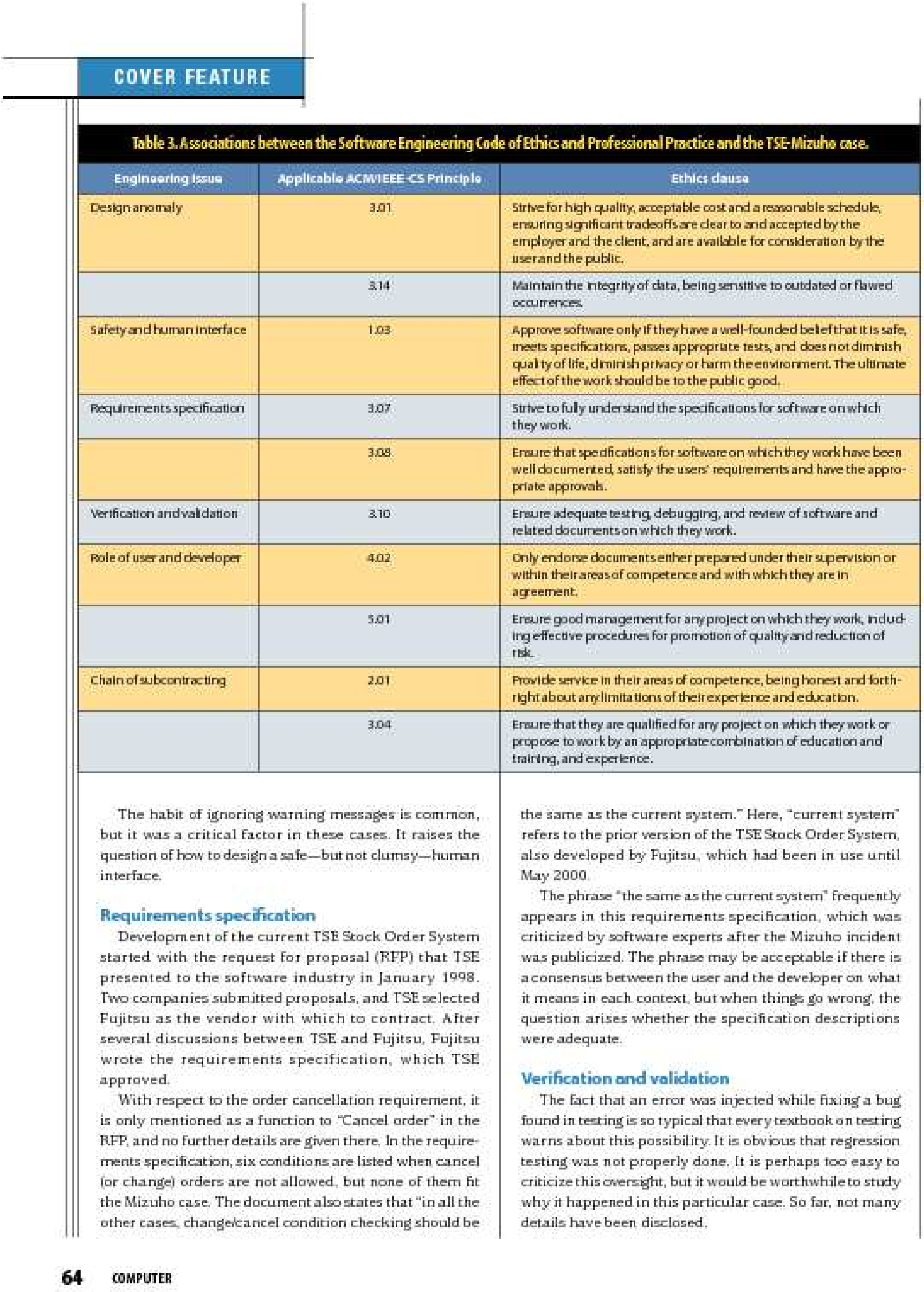
In addition to the insights into the associated software design problems, this case provides lessons learned with regard to software engineering technologies, processes, and social aspects.

Safety and human interface

If the order entry system on either the Mizuho or TSE side had been equipped with more elaborate safety measures, the accident could have been avoided. It was not the first time that the mistyping of a stock order resulted in a big loss. For instance, in December 2001, a trader at UBS Warburg, the Swiss investment bank, lost more than 10 billion yen while trying to sell 16 shares of the Japanese advertising company Dentsu at 610,000 yen each. He sold 610,000 shares at six yen each. (The similarity between these two cases, including the common figure of 610,000, is remarkable.)

Figure 1. Flowchart of the order cancellation module.

JUNE 2009 63



COVER FEATURE

Table 3. Associations between the Software Engineering Code of Ethics and Professional Practice and the TSE-Mizuho case.

Engineering issue	Applicable ACM/IEEE-CS Principle	Ethics clause
Design anomaly	3.01	Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.
	3.14	Maintain the integrity of data, being sensitive to outdated or flawed occurrences.
Safety and human interface	1.03	Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.
Requirements specification	3.07	Strive to fully understand the specifications for software on which they work.
	3.08	Ensure that specifications for software on which they work have been well documented, satisfy the users' requirements and have the appropriate approvals.
Verification and validation	3.10	Ensure adequate testing, debugging, and review of software and related documents on which they work.
Role of user and developer	4.02	Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.
	5.01	Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.
Chain of subcontracting	2.01	Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.
	3.04	Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.

The habit of ignoring warning messages is common, but it was a critical factor in these cases. It raises the question of how to design a safe—but not clumsy—human interface.

Requirements specification

Development of the current TSE Stock Order System started with the request for proposal (RFP) that TSE presented to the software industry in January 1998. Two companies submitted proposals, and TSE selected Fujitsu as the vendor with which to contract. After several discussions between TSE and Fujitsu, Fujitsu wrote the requirements specification, which TSE approved.

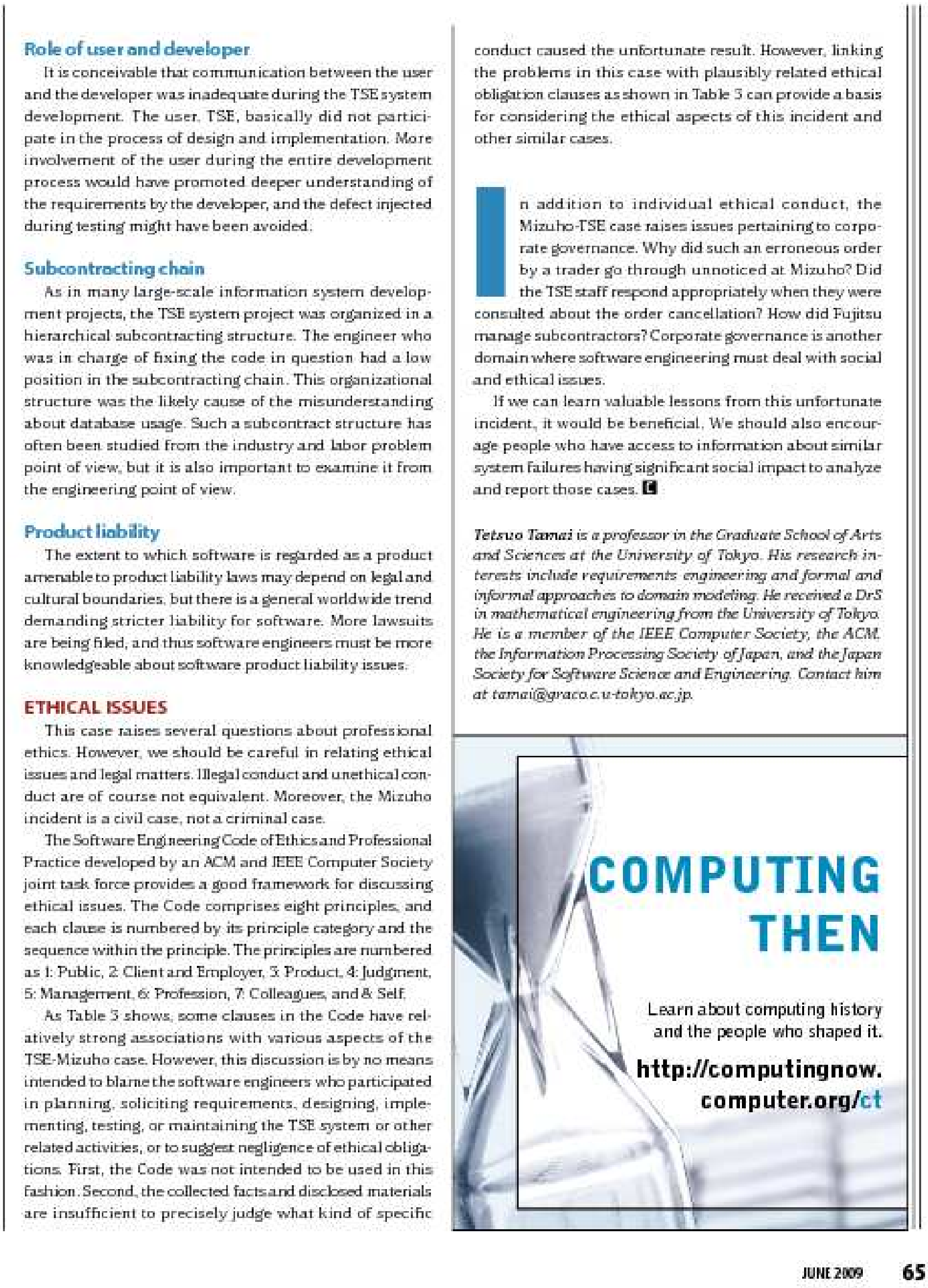
With respect to the order cancellation requirement, it is only mentioned as a function to "Cancel order" in the RFP, and no further details are given there. In the requirements specification, six conditions are listed when cancel (or change) orders are not allowed, but none of them fit the Mizuho case. The document also states that "in all the other cases, change/cancel condition checking should be the same as the current system." Here, "current system" refers to the prior version of the TSE Stock Order System, also developed by Fujitsu, which had been in use until May 2000.

The phrase "the same as the current system" frequently appears in this requirements specification, which was criticized by software experts after the Mizuho incident was publicized. The phrase may be acceptable if there is a consensus between the user and the developer on what it means in each context, but when things go wrong, the question arises whether the specification descriptions were adequate.

Verification and validation

The fact that an error was injected while fixing a bug found in testing is so typical that every textbook on testing warns about this possibility. It is obvious that regression testing was not properly done. It is perhaps too easy to criticize this oversight, but it would be worthwhile to study why it happened in this particular case. So far, not many details have been disclosed.

64 COMPUTER



Role of user and developer

It is conceivable that communication between the user and the developer was inadequate during the TSE system development. The user, TSE, basically did not participate in the process of design and implementation. More involvement of the user during the entire development process would have promoted deeper understanding of the requirements by the developer, and the defect injected during testing might have been avoided.

Subcontracting chain

As in many large-scale information system development projects, the TSE system project was organized in a hierarchical subcontracting structure. The engineer who was in charge of fixing the code in question had a low position in the subcontracting chain. This organizational structure was the likely cause of the misunderstanding about database usage. Such a subcontract structure has often been studied from the industry and labor problem point of view, but it is also important to examine it from the engineering point of view.

Product liability

The extent to which software is regarded as a product amenable to product liability laws may depend on legal and cultural boundaries, but there is a general worldwide trend demanding stricter liability for software. More lawsuits are being filed, and thus software engineers must be more knowledgeable about software product liability issues.

ETHICAL ISSUES

This case raises several questions about professional ethics. However, we should be careful in relating ethical issues and legal matters. Illegal conduct and unethical conduct are of course not equivalent. Moreover, the Mizuho incident is a civil case, not a criminal case.

The Software Engineering Code of Ethics and Professional Practice developed by an ACM and IEEE Computer Society joint task force provides a good framework for discussing ethical issues. The Code comprises eight principles, and each clause is numbered by its principle category and the sequence within the principle. The principles are numbered as 1: Public, 2: Client and Employer, 3: Product, 4: Judgment, 5: Management, 6: Profession, 7: Colleagues, and 8: Self.

As Table 3 shows, some clauses in the Code have relatively strong associations with various aspects of the TSE-Mizuho case. However, this discussion is by no means intended to blame the software engineers who participated in planning, soliciting requirements, designing, implementing, testing, or maintaining the TSE system or other related activities, or to suggest negligence of ethical obligations. First, the Code was not intended to be used in this fashion. Second, the collected facts and disclosed materials are insufficient to precisely judge what kind of specific conduct caused the unfortunate result. However, linking the problems in this case with plausibly related ethical obligation clauses as shown in Table 3 can provide a basis for considering the ethical aspects of this incident and other similar cases.

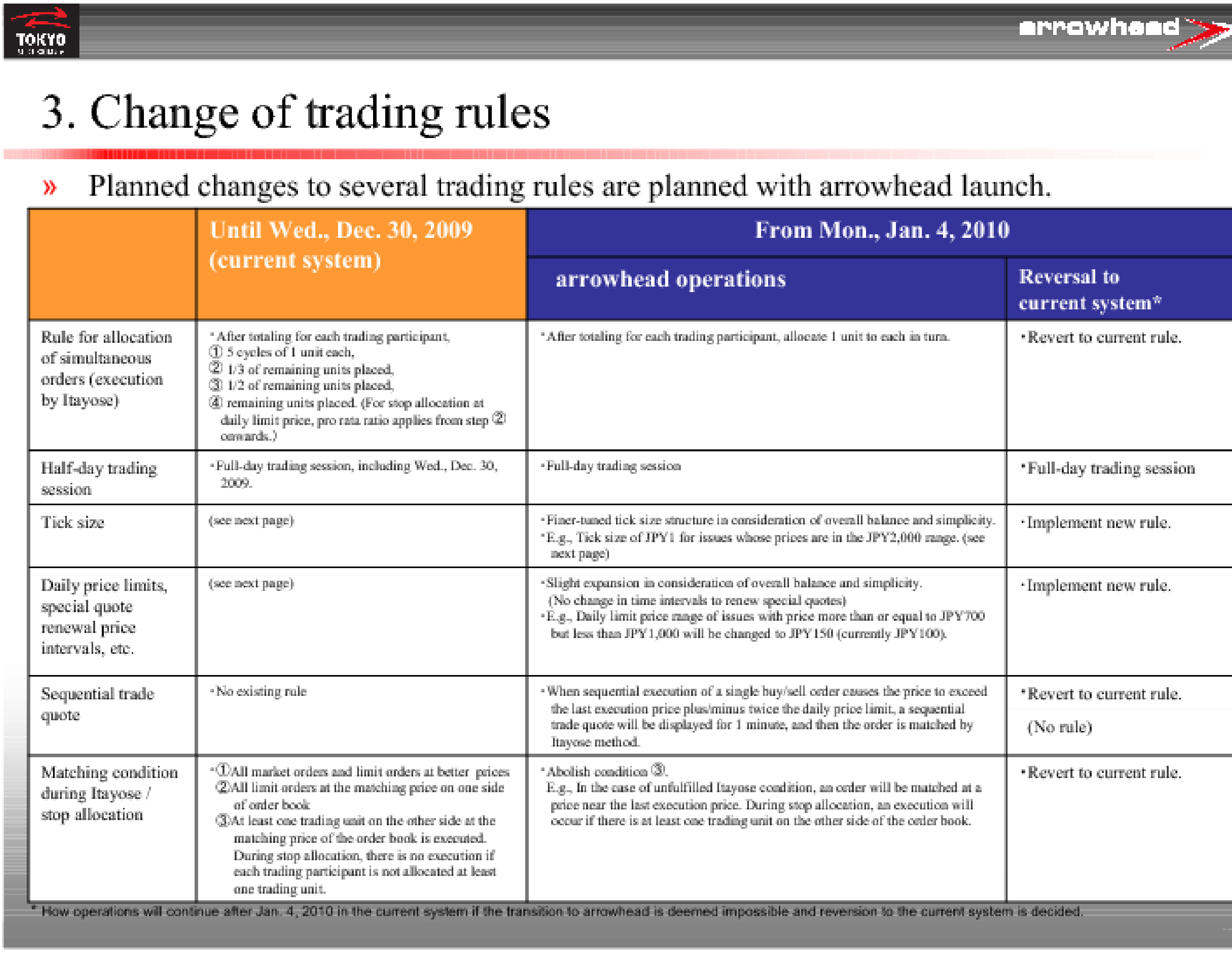
In addition to individual ethical conduct, the Mizuho-TSE case raises issues pertaining to corporate governance. Why did such an erroneous order by a trader go through unnoticed at Mizuho? Did the TSE staff respond appropriately when they were consulted about the order cancellation? How did Fujitsu manage subcontractors? Corporate governance is another domain where software engineering must deal with social and ethical issues.

If we can learn valuable lessons from this unfortunate incident, it would be beneficial. We should also encourage people who have access to information about similar system failures having significant social impact to analyze and report those cases. ■

Tetsuo Tamai is a professor in the Graduate School of Arts and Sciences at the University of Tokyo. His research interests include requirements engineering and formal and informal approaches to domain modeling. He received a DrS in mathematical engineering from the University of Tokyo. He is a member of the IEEE Computer Society, the ACM, the Information Processing Society of Japan, and the Japan Society for Software Science and Engineering. Contact him at tamai@graco.c.u-tokyo.ac.jp.

V.5 The New Tokyo Stock Exchange System

V.5.1 Change of trading rules



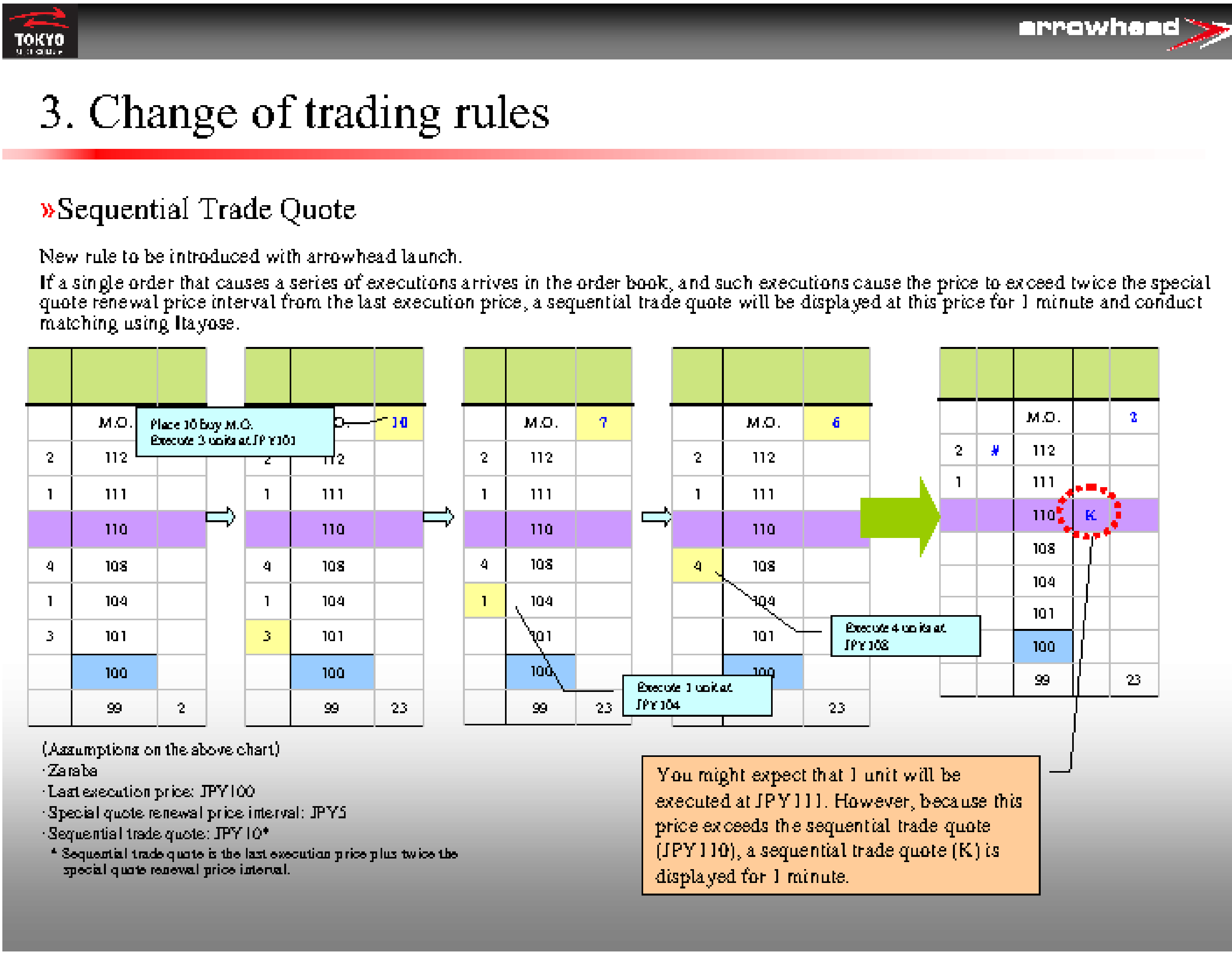
TOKYO
arrowhead

3. Change of trading rules

» Planned changes to several trading rules are planned with arrowhead launch.

	Until Wed., Dec. 30, 2009 (current system)	From Mon., Jan. 4, 2010	
		arrowhead operations	Reversal to current system*
Rule for allocation of simultaneous orders (execution by Itayose)	・After totaling for each trading participant, ① 5 cycles of 1 unit each, ② 1/3 of remaining units placed, ③ 1/2 of remaining units placed, ④ remaining units placed. (For stop allocation at daily limit price, pro rata ratio applies from step ② onwards.)	・After totaling for each trading participant, allocate 1 unit to each in turn.	・Revert to current rule.
Half-day trading session	・Full-day trading session, including Wed., Dec. 30, 2009.	・Full-day trading session	・Full-day trading session
Tick size	(see next page)	・Finer-tuned tick size structure in consideration of overall balance and simplicity. ・E.g., Tick size of JPY1 for issues whose prices are in the JPY2,000 range. (see next page)	・Implement new rule.
Daily price limits, special quote renewal price intervals, etc.	(see next page)	・Slight expansion in consideration of overall balance and simplicity. (No change in time intervals to renew special quotes) ・E.g., Daily limit price range of issues with price more than or equal to JPY700 but less than JPY1,000 will be changed to JPY150 (currently JPY100).	・Implement new rule.
Sequential trade quote	・No existing rule	・When sequential execution of a single buy/sell order causes the price to exceed the last execution price plus/minus twice the daily price limit, a sequential trade quote will be displayed for 1 minute, and then the order is matched by Itayose method.	・Revert to current rule. (No rule)
Matching condition during Itayose / stop allocation	・①All market orders and limit orders at better prices ②All limit orders at the matching price on one side of order book ③At least one trading unit on the other side at the matching price of the order book is executed. During stop allocation, there is no execution if each trading participant is not allocated at least one trading unit.	・Abolish condition ③. E.g., In the case of unfulfilled Itayose condition, an order will be matched at a price near the last execution price. During stop allocation, an execution will occur if there is at least one trading unit on the other side of the order book.	・Revert to current rule.

* How operations will continue after Jan. 4, 2010 in the current system if the transition to arrowhead is deemed impossible and reversion to the current system is decided.



3. Change of trading rules

»Sequential Trade Quote

New rule to be introduced with arrowhead launch.

If a single order that causes a series of executions arrives in the order book, and such executions cause the price to exceed twice the special quote renewal price interval from the last execution price, a sequential trade quote will be displayed at this price for 1 minute and conduct matching using Itayose.

Place 10 Buy M.O.
Execute 3 units at JPY101

	M.O.	
2	112	
1	111	
	110	
4	108	
1	104	
3	101	
	100	
	99	2

	O.	10
2	112	
1	111	
	110	
4	108	
1	104	
3	101	
	100	
	99	23

	M.O.	7
2	112	
1	111	
	110	
4	108	
1	104	
	101	
	100	
	99	23

Execute 1 unit at JPY104

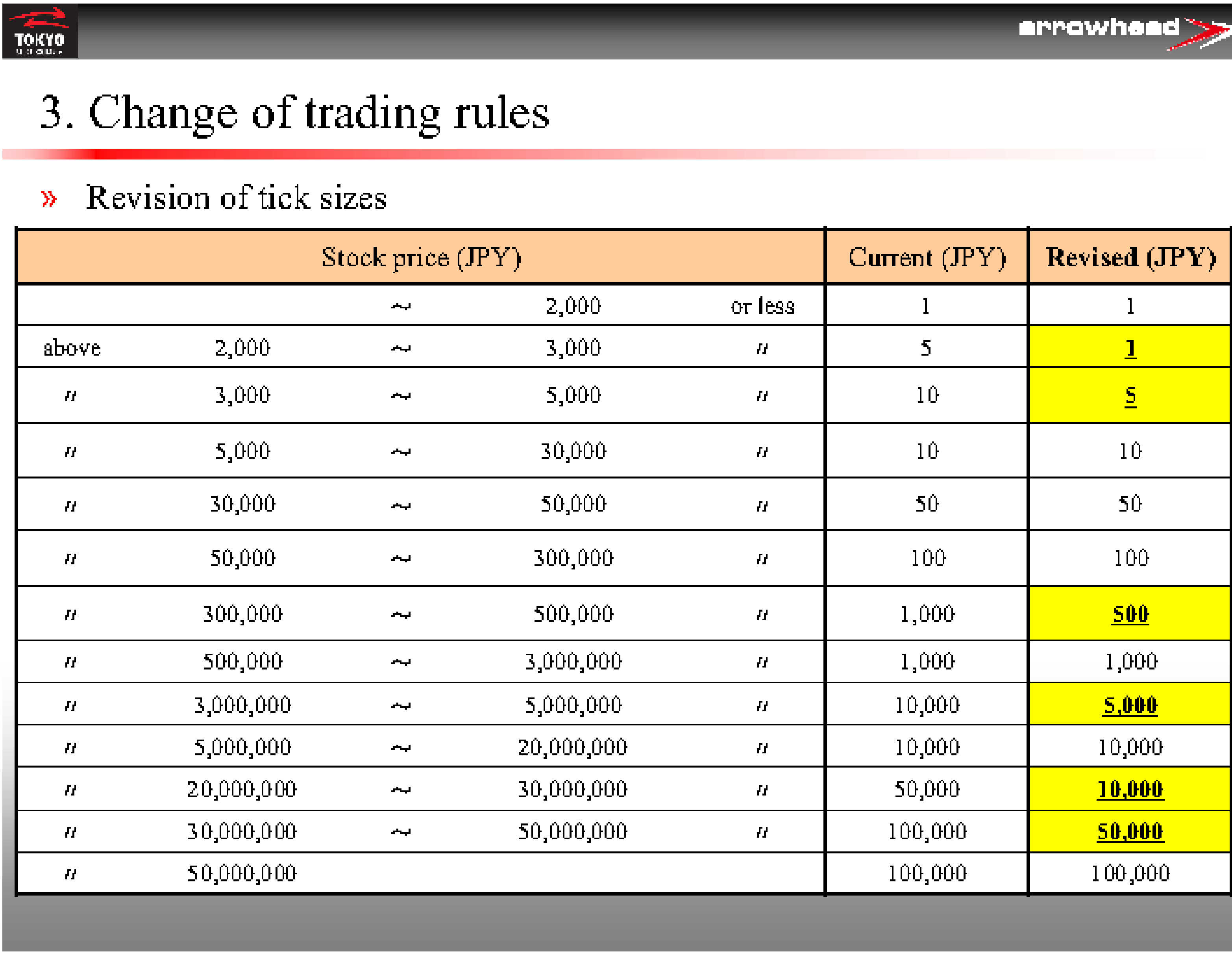
	M.O.	6
2	112	
1	111	
	110	
4	108	
	104	
	101	
	100	
		23

Execute 4 units at JPY108

		M.O.		2
2	#	112		
1		111		
		110	K	
		108		
		104		
		101		
		100		
		99		23

(Assumptions on the above chart)
·Zaraba
·Last execution price: JPY100
·Special quote renewal price interval: JPY5
·Sequential trade quote: JPY10*
* Sequential trade quote is the last execution price plus twice the special quote renewal price interval.

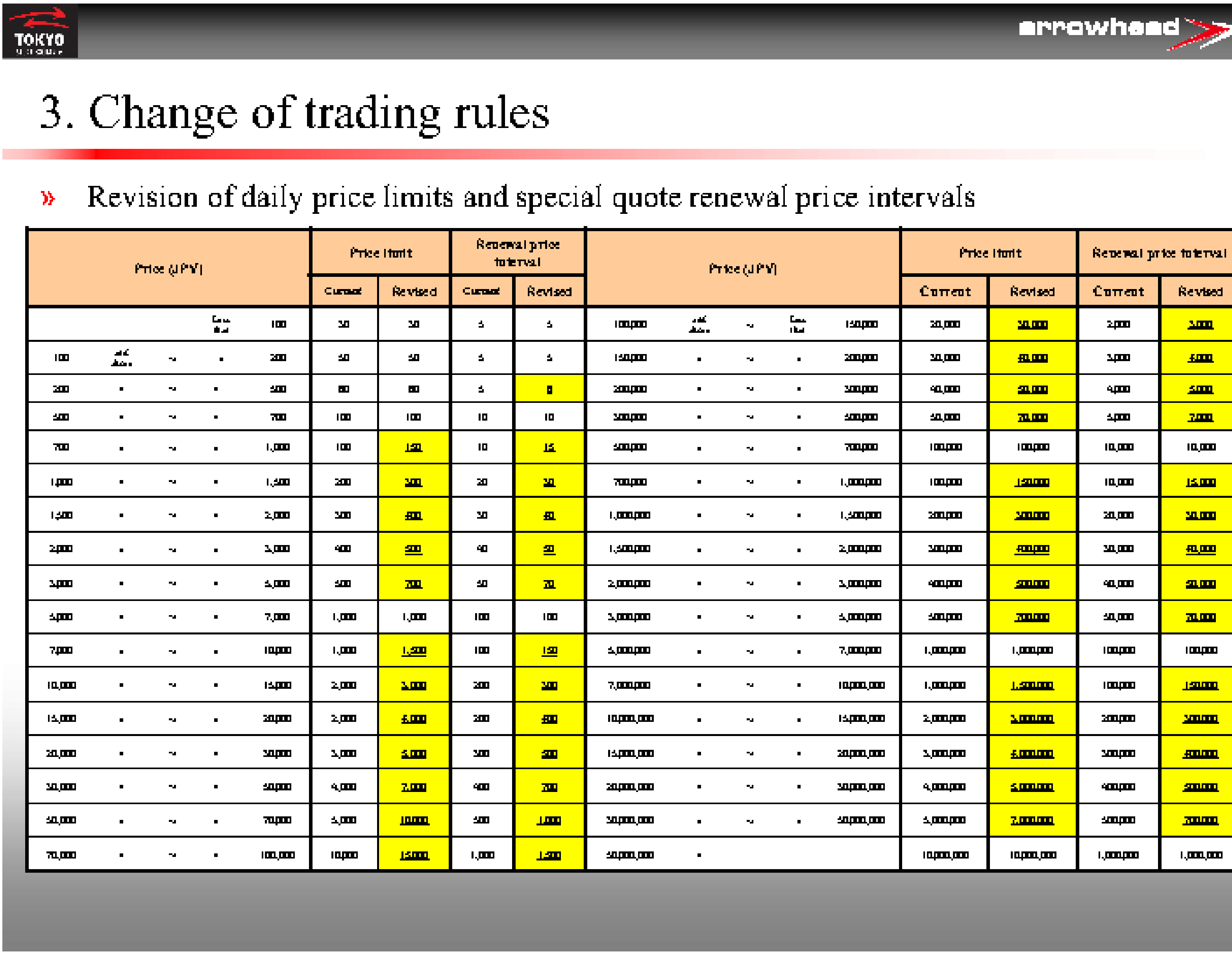
You might expect that 1 unit will be executed at JPY111. However, because this price exceeds the sequential trade quote (JPY110), a sequential trade quote (K) is displayed for 1 minute.



3. Change of trading rules

» Revision of tick sizes

Stock price (JPY)					Current (JPY)	Revised (JPY)
		～	2,000	or less	1	1
above	2,000	～	3,000	〃	5	**1**
〃	3,000	～	5,000	〃	10	**5**
〃	5,000	～	30,000	〃	10	10
〃	30,000	～	50,000	〃	50	50
〃	50,000	～	300,000	〃	100	100
〃	300,000	～	500,000	〃	1,000	**500**
〃	500,000	～	3,000,000	〃	1,000	1,000
〃	3,000,000	～	5,000,000	〃	10,000	**5,000**
〃	5,000,000	～	20,000,000	〃	10,000	10,000
〃	20,000,000	～	30,000,000	〃	50,000	**10,000**
〃	30,000,000	～	50,000,000	〃	100,000	**50,000**
〃	50,000,000				100,000	100,000

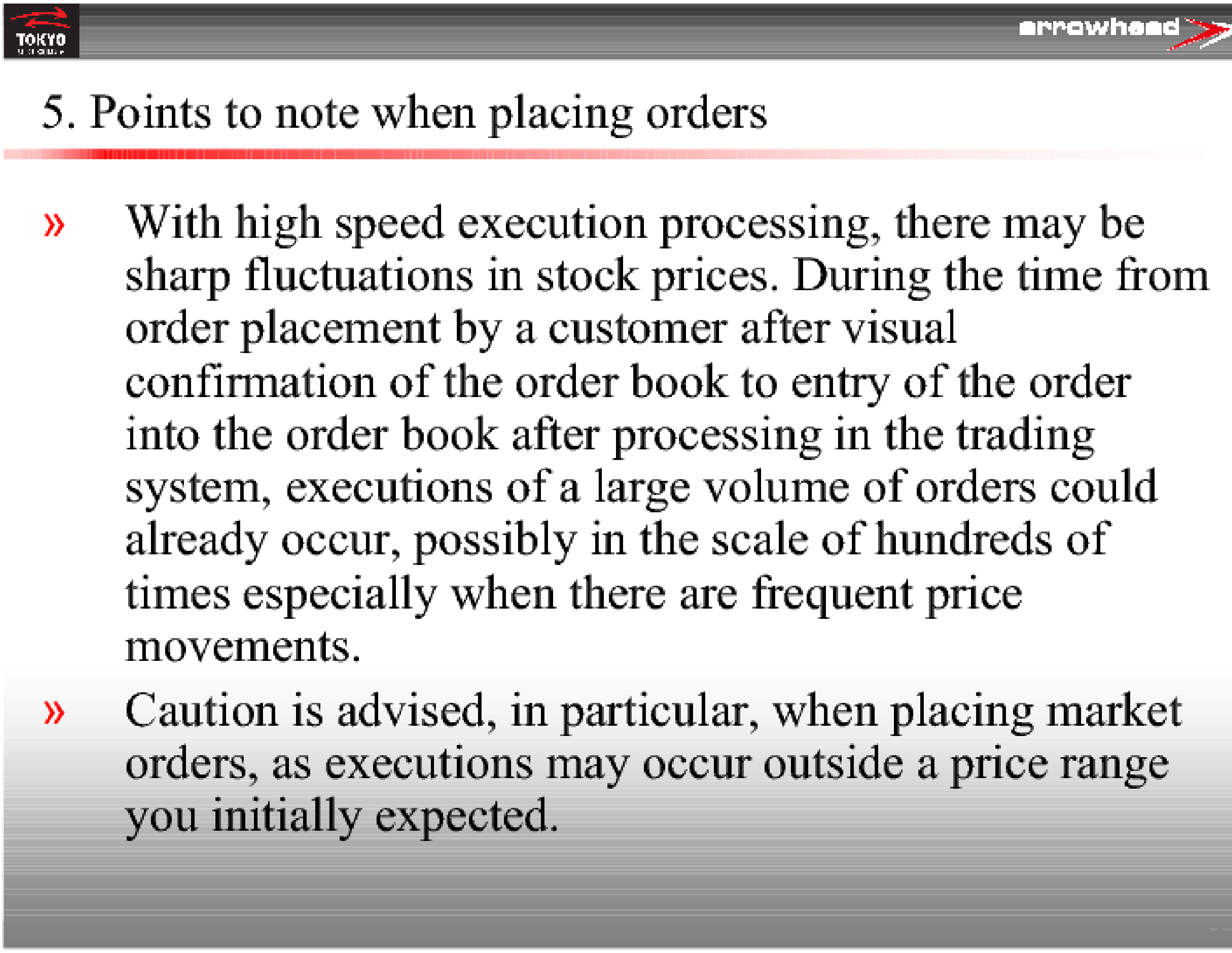


3. Change of trading rules

» Revision of daily price limits and special quote renewal price intervals

Price (JPY)					Price limit		Renewal price interval		Price (JPY)					Price limit		Renewal price interval	
					Current	Revised	Current	Revised						Current	Revised	Current	Revised
			Less than	100	30	30	5	5	100,000	and above	~	Less than	150,000	20,000	30,000	2,000	3,000
100	and above	~	・	200	50	50	5	5	150,000	・	~	・	200,000	30,000	40,000	3,000	4,000
200	・	~	・	500	80	80	5	8	200,000	・	~	・	300,000	40,000	50,000	4,000	5,000
500	・	~	・	700	100	100	10	10	300,000	・	~	・	500,000	50,000	70,000	5,000	7,000
700	・	~	・	1,000	100	150	10	15	500,000	・	~	・	700,000	100,000	100,000	10,000	10,000
1,000	・	~	・	1,500	200	300	20	30	700,000	・	~	・	1,000,000	100,000	150,000	10,000	15,000
1,500	・	~	・	2,000	300	400	30	40	1,000,000	・	~	・	1,500,000	200,000	300,000	20,000	30,000
2,000	・	~	・	3,000	400	500	40	50	1,500,000	・	~	・	2,000,000	300,000	400,000	30,000	40,000
3,000	・	~	・	5,000	500	700	50	70	2,000,000	・	~	・	3,000,000	400,000	500,000	40,000	50,000
5,000	・	~	・	7,000	1,000	1,000	100	100	3,000,000	・	~	・	5,000,000	500,000	700,000	50,000	70,000
7,000	・	~	・	10,000	1,000	1,500	100	150	5,000,000	・	~	・	7,000,000	1,000,000	1,000,000	100,000	100,000
10,000	・	~	・	15,000	2,000	3,000	200	300	7,000,000	・	~	・	10,000,000	1,000,000	1,500,000	100,000	150,000
15,000	・	~	・	20,000	2,000	4,000	200	400	10,000,000	・	~	・	15,000,000	2,000,000	3,000,000	200,000	300,000
20,000	・	~	・	30,000	3,000	5,000	300	500	15,000,000	・	~	・	20,000,000	3,000,000	4,000,000	300,000	400,000
30,000	・	~	・	50,000	4,000	7,000	400	700	20,000,000	・	~	・	30,000,000	4,000,000	5,000,000	400,000	500,000
50,000	・	~	・	70,000	5,000	10,000	500	1,000	30,000,000	・	~	・	50,000,000	5,000,000	7,000,000	500,000	700,000
70,000	・	~	・	100,000	10,000	15,000	1,000	1,500	50,000,000	・				10,000,000	10,000,000	1,000,000	1,000,000

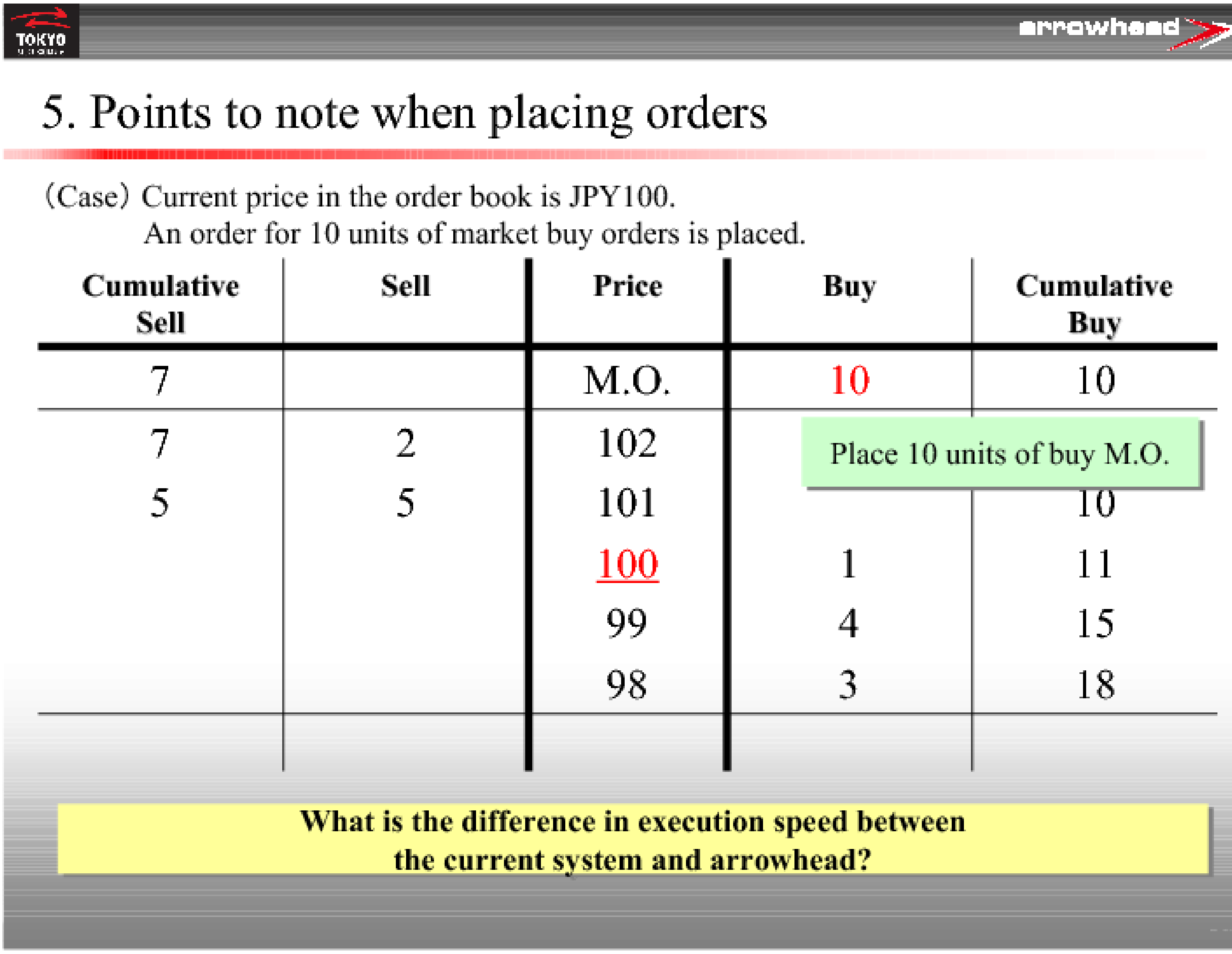
V.5.2 Points to note when placing orders


TOKYO
arrowhead

5. Points to note when placing orders

» With high speed execution processing, there may be sharp fluctuations in stock prices. During the time from order placement by a customer after visual confirmation of the order book to entry of the order into the order book after processing in the trading system, executions of a large volume of orders could already occur, possibly in the scale of hundreds of times especially when there are frequent price movements.

» Caution is advised, in particular, when placing market orders, as executions may occur outside a price range you initially expected.



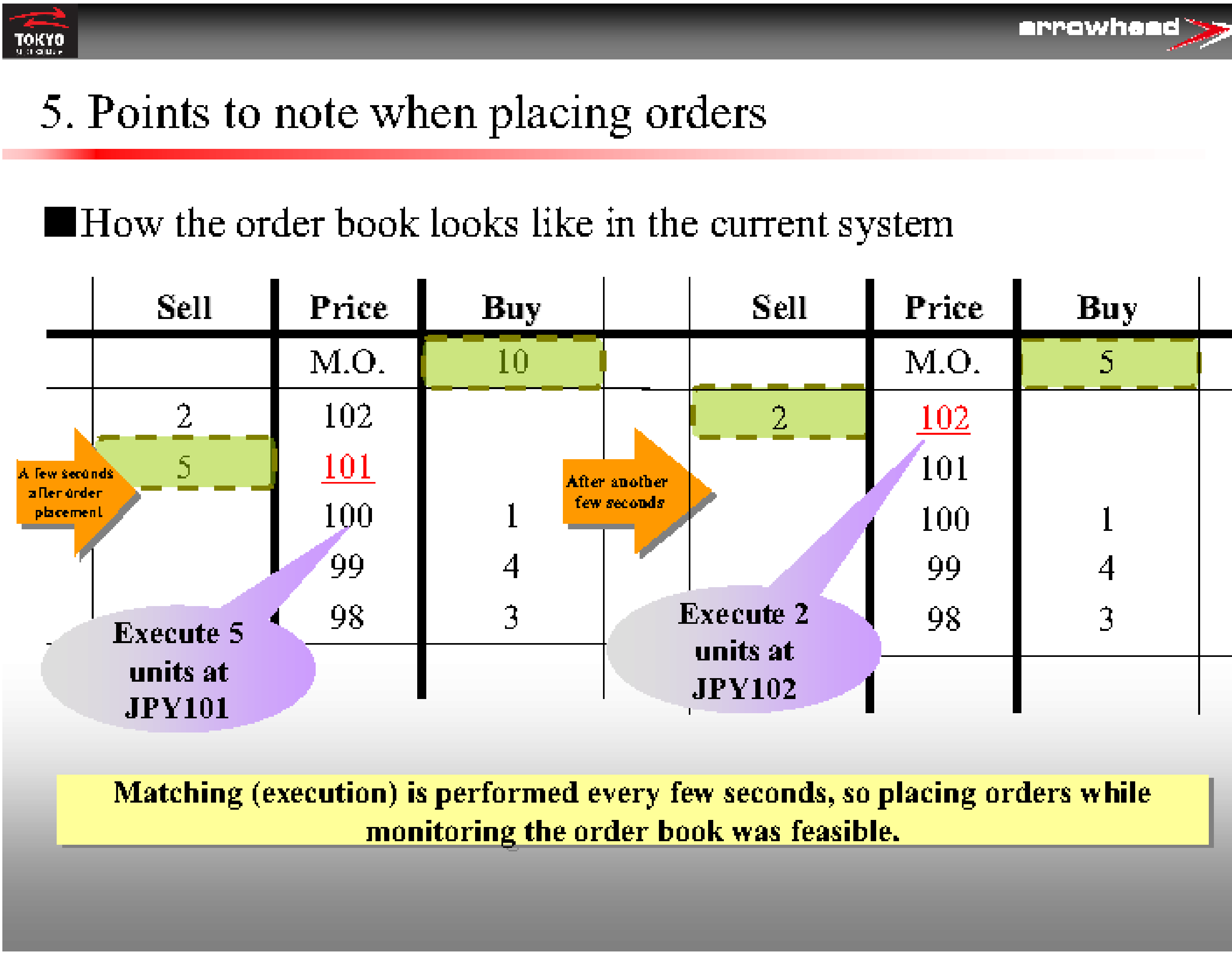
5. Points to note when placing orders

（Case）Current price in the order book is JPY100.
An order for 10 units of market buy orders is placed.

Cumulative Sell	Sell	Price	Buy	Cumulative Buy
7		M.O.	10	10
7	2	102		
5	5	101		10
		100	1	11
		99	4	15
		98	3	18

Place 10 units of buy M.O.

What is the difference in execution speed between the current system and arrowhead?



5. Points to note when placing orders

■How the order book looks like in the current system

Sell	Price	Buy
	M.O.	10
2	102	
5	101	
	100	1
	99	4
	98	3

A few seconds after order placement

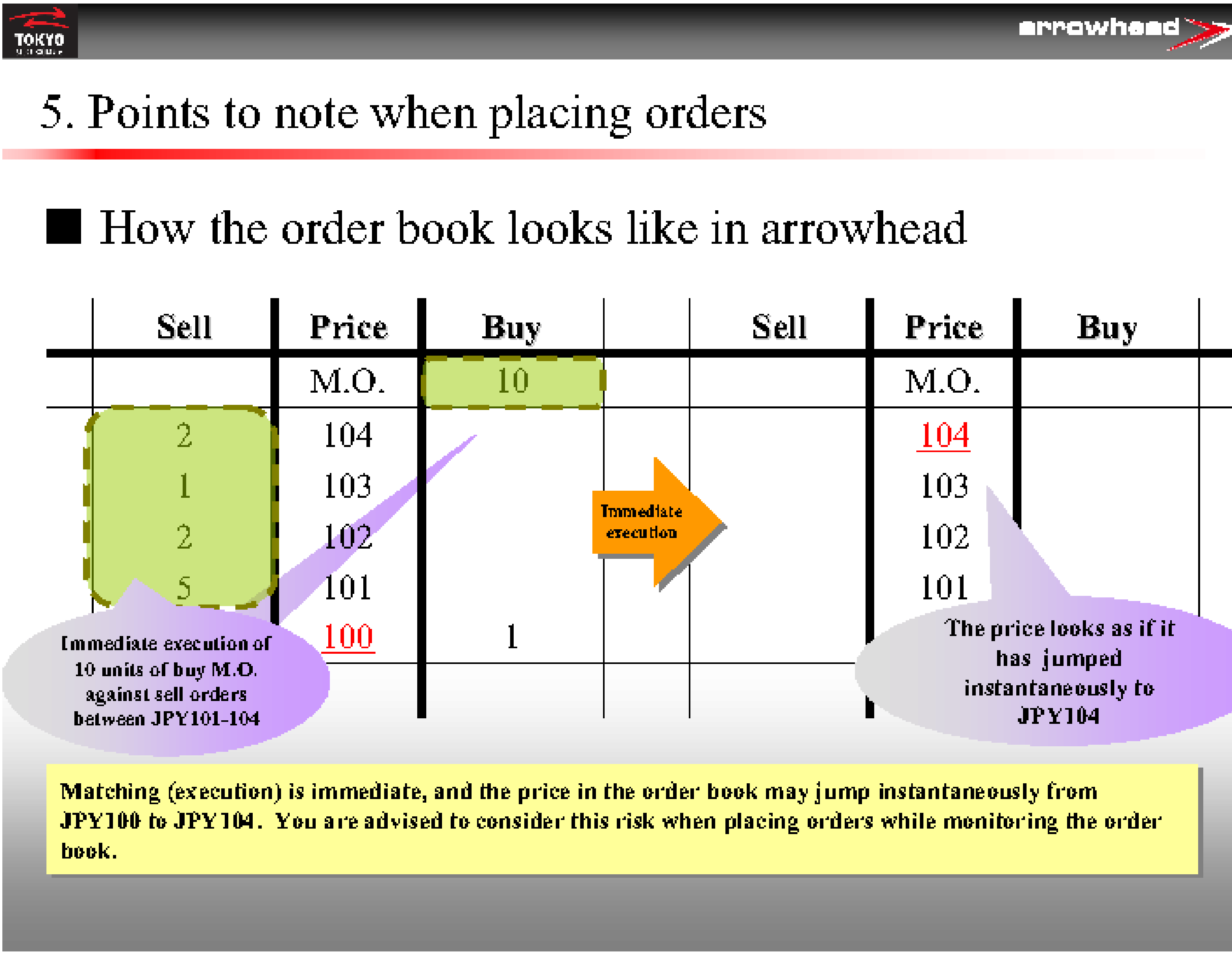
Execute 5 units at JPY101

After another few seconds

Sell	Price	Buy
	M.O.	5
2	102	
	101	
	100	1
	99	4
	98	3

Execute 2 units at JPY102

Matching (execution) is performed every few seconds, so placing orders while monitoring the order book was feasible.



5. Points to note when placing orders

How the order book looks like in arrowhead

Sell	Price	Buy
	M.O.	10
2	104	
1	103	
2	102	
5	101	
	100	1

Immediate execution of 10 units of buy M.O. against sell orders between JPY101-104

Immediate execution

Sell	Price	Buy
	M.O.	
	104	
	103	
	102	
	101	

The price looks as if it has jumped instantaneously to JPY104

Matching (execution) is immediate, and the price in the order book may jump instantaneously from JPY100 to JPY104. You are advised to consider this risk when placing orders while monitoring the order book.

W What Is Logistics ?

W.1 Abstract

We examine the concept of logistics, exemplify it by some *"use cases"*, bring a definition of the term 'logistics' from Wikipedia (Sect. W.6: *What is Logistics*), and then we rigorously and stepwise unravel the constituent concepts of *Transport Networks* (Sect. W.7), *Containers and Freight Items* (Sect. W.8), *Transport Companies, Vehicles and Timetables* (Sect. W.9), *Handling* (Sect. W.10), *Logistics Traffic* (Sect. W.11) and *Senders and Receivers* (Sect. W.12). In Sects. W.13–W.14, *Model Extensions*, we discuss possible additional phenomena and concepts of logistics.

The document presents a domain model (in the form of a both English narrative and a formal RSL description), that is, it does not present requirements to a computerised logistics system, let alone software for such systems.

A concluding section, *Logistics System Functions* (Sect. W.15) — to be written — surveys some standard software and hardware support for logistics.

We constrain the treatment of logistics to that of shipping companies handling (optimal) freight consignments (cf. waybills and bill of ladings) involving possibly multiple vehicles from possibly multiple transport companies.

Thus we do not cover the logistics of, say, container stowage aboard container vessels. In **http://www2.imm.dtu.dk/˜db/container-paper.pdf** we cover that aspect.

W.2 Methodology

This document applies the domain engineering principles of [14, 15, 16, 19, 21] to the domain of logistics. The specification language used is RSL of the RAISE method [49]. The three volume [11, 13, 14] gives an overall, 2400 page introduction to software engineering, the RAISE specification language RSL, to abstraction and and modelling principles and techniques, and to the triptych of software engineering: domain engineering as a basis for requirements engineering and the latter as a basis for software design. Included in [13] are introductions to Automata and Machines, Modules and Class Diagrams, Petri Nets [73], Message Sequence Charts [57], State Charts [54] and Temporal Logic (in the form of DC for Duration Calculus, [85]). In the present document we shall not tackle problems that cannot be expressed in RSL. A most recent and comprehensive intriduction to domain engineering is the less than 200 page document: http://www2.imm.dtu.dk/˜db/de+re-p.pdf.

W.3 A Series of Domain Descriptions

This document is one in an emerging series of documents that describe indidual domains: a financial service industry (banks, securities trading, etc.), a container line industry[52],

[52] http://www2.imm.dtu.dk/ db/container-paper.pdf

pipe line systems[53], railways[54], etc.

W.4 Obviously Missing Diagrams &c.

The current version is relative complete: In Sect. W.10.2 on page 316 we reach a "current" high in expressing the generation of waybills from requests for consignment and optimal transport wrt. different criteria. But what is missing for the lay reader is: (i) diagrams to easen the intuitive understanding of text and formulas and (ii) explanations of the formula.

W.5 Why This Document ?

W.5.1 Facts

There is no document which describes logistics in a precise manner. Thus there is no student text from which one can learn about logistics in a professionally responsible way.

W.5.2 Aims & Objectives

By *aims* we mean: *what is being covered in this document ?* By *objectives* we mean: *what do we wish to achieve by presenting this document ?*

Aims

We aim to cover all facets of logistics: a detailed description of the multi-modal transport nets along which suitable vehicles transport freight, from initial hub or link position origins of the net along routes of the net to hubs or link positions of the net to final hub or link position destinations of the net possibly changing from vehicles to vehicles of same or different modalities (trucks, trains, air-cargo or vessels) while possibly being temporarily warehouse stored for further shipment; a detailed description of the functions of senders, shipping companies and receivers: senders making inquiries, placing requests for transportation, accepting shipper proposed routes and fares, etc.; shipping companies finding optimal freight routes with respect to any one or a composition of requirements, and with respect to transport company time– and fare tables; and accepting responsibility for shipments, providing senders and receivers with regular information as to the whereabouts of the consigned freight, etc.; a description of those aspects of transport companies, their vehicles the timetables according to which vehicles perform transport; etc., etc.

Objectives

It is our objective to achieve the following with this document: (i) to show that one can indeed provide a concise English narrative as well as a precise mathematical formalisation of all of the above-mentioned and many more aspects of logistics; (ii) to implicitly convince the

[53] http://www2.imm.dtu.dk/~db/de+re-p.pdf

[54] http://www.railwaydomain.org/PDF/tb.pdf

reader that no software development ought begin without a clear, consistent and relative complete domain description of 'logistics' — including that it can be done; and (iii) to suggest that education and training, of students of shipping, and research into logistics be based on domain descriptions like the one of this document.

W.6 What is Logistics

W.6.1 The "Players"

Figure 16 indicates the five major "players" on the 'logistics' scene, from left to right: the senders and receivers of freight, the shipping companies, the transport companies and their vehicles, and the transport net.

The reader may observe that we have not indicated, by any symbol, the "real" object of logistics, namely the freight items !

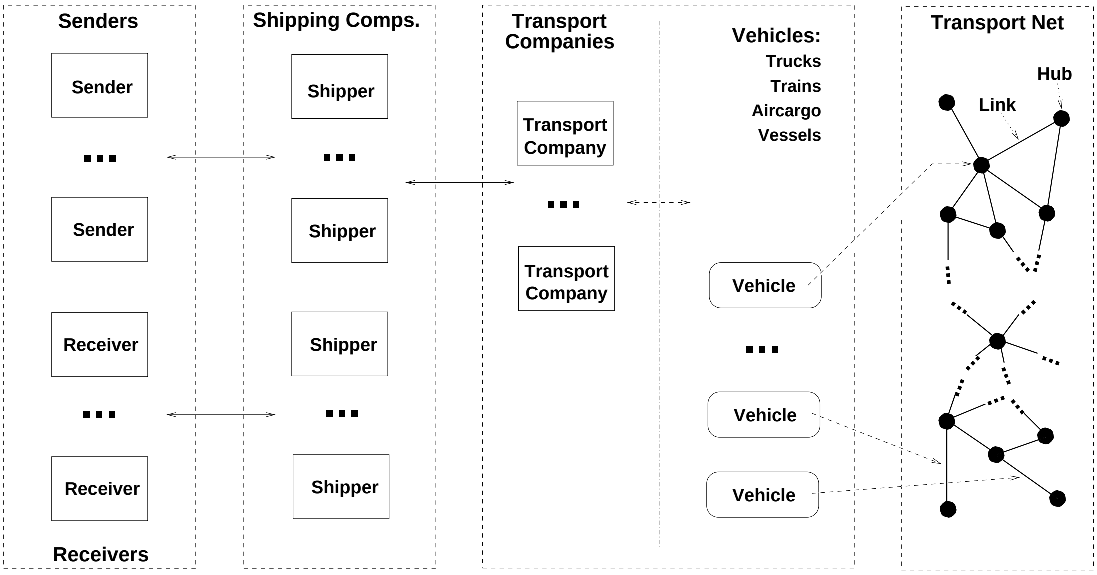


Figure 16: The Logistics "Players"

W.6.2 Some Use Cases

We present three use cases.

Consignment and Transport

1. You are a **sender**[55]: a person who, or a company which, wishes to send a consignment of a number of one or more pieces of freight from location O (origin), say in Asia, to location D (destination), say in Europe.

[55] The **bold face** terms appear on Fig. 16.

2. So you contact a **shipper**, that is, a **shipping company**.
3. You inform them of
   a) number of pieces of freight, the individual measures (height, width, breadth and weight) of this freight,
   b) from whom, i.e., the **sender**, name, etc., when (date and time) and where (address, **hub** or **link**[56] position) it is to be fetched,
   c) to whom, i.e., the **receiver**, name, etc., and where (address, **hub** or **link** position) it to be delivered,
   d) whether the freight items are already packed,
   e) whether the freight is fragile
   f) and/or flammable,
   g) value of each freight item,
   h) et cetera.
4. The **shipping company**,
   a) based on knowledge about **transport companies**,
   b) the timetables of their **vehicles** and
   c) the **transport net** of these vehicles,
5. suggests a route of transport
   a) with this route usually composed from several transport segments:
   b) **truck, train, air-cargo** or **vessel**, etc., ending possibly with train and truck delivery.
6. The **shipping company** informs the **sender** of
   a) transportation price,
   b) whether **receiver** pays for local delivery or you do;
   c) transportation dates and times:
      i. initial fetch (from a **link** position),
      ii. intermediate transfers and possible warehousing (at **hubs**),
      iii. and final delivery (from a **link** position).
7. You agree,

[56] Items 3a–3b, when specifying link positions assume truck fetch or delivery — as trains, aircraft and vessels can only pause at hubs.

   a) after some negotiation
   b) that might involve alternative routes (et cetera),

8. and sign appropriate papers
   a) bill of lading[57]
   b) and waybills[58].
9. Your freight is fetched (from a **link** position).
10. You are — perhaps — regularly or irregularly informed of status of transport.
11. Finally freight arrives and is delivered to receiver (at a **link** position).

Discussion

Items 9–11 are not logistics actions. They are not performed by the shipper, maybe except for cases of Item 10. Instead they are performed by the transport company and its vehicles. Thus you see that the rôle of a shipper is to arrange, to accommodate — i.e., to manage ! The management of overall vehicle coordination with respect to (wrt.) senders, shippers and receivers is done by the transport companies and is not considered an issue of logistics. The management individual vehicles is done by the truck driver, the train engine man, the aircraft captain (pilot), respectively the ship captain and is likewise not considered an issue of logistics.

Inquiry

You are a person who, or a company which, wishes to send a consignment of a number of one or more pieces of freight from location O (origin), say in Asia, to location D (destination), say in Europe. You are wondering about costs, transportation times, etc. So you "shop around": inquiring with a number of (one or more) shipping companies as for shipping route, times, costs, packaging, insurance, et cetera.

Therefore several of the actions mentioned above take place.

[57] Wikipedia: A bill of lading (sometimes referred to as a BOL, or B/L) is a document issued by a carrier to a shipper, acknowledging that specified goods have been received on board as cargo for conveyance to a named place for delivery to the consignee who is usually identified. A through bill of lading involves the use of at least two different modes of transport from road, rail, air, and sea. The term derives from the noun "bill", a schedule of costs for services supplied or to be supplied, and from the verb "to lade" which means to load a cargo onto a ship or other form of transport.

[58] Wikipedia: A waybill is a document issued by a carrier giving details and instructions relating to the shipment of a consignment of goods. Typically it will show the names of the consignor and consignee, the point of origin of the consignment, its destination, route, and method of shipment, and the amount charged for carriage. Unlike a bill of lading, which includes much of the same information, a waybill is not a document of title.

Tracing

You are a person who, or a company which, has commits the consignment of a number of one or more pieces of freight from location O (origin), say in Asia, to location D (destination), say in Europe. There is therefore a set of bill of ladings and a waybill — all with appropriate reference identifications. Now, after initial send-off of freight, you wish to know the status of the ongoing transport, or why it appears that there is a delay in shipping. Tracing therefore takes place: the shipping company via the transport companies, finding out about the whereabouts of the freight. Et cetera,

W.6.3 A Wikipedia Definition of 'Logistics'

According to Wikipedia (http://en.wikipedia.org/wiki/Logistics):

> *"Logistics is the management of the transport of goods, information and other resources, including energy and people, between the point of origin and the point of destination in order to meet the requirements of consumers (frequently, and originally, military organizations). Logistics involves the integration of information, transportation, inventory, warehousing, material-handling, packaging, and occasionally security*[59]*. Logistics is a channel of the supply chain which adds the the value of time and place utility."*

W.6.4 A Definition of 'Transport'

By transport[ation] we shall mean

> (i) the movement (ii) of goods (iii) on a vehicle (iv) along a route of a network of hubs and [two way] links[60] (v) from a source (point of origin) to a sink (a point of destination).

(i) Movement is a behaviour, that is, a function over time. (ii) Goods are items of freight that have value, volume, maybe perishable (that is, whose value diminishes rapidly with excess transportation time). (iii) Vehicles are like actors: they convey freight, they can accommodate a maximum of freight volume and weight, they can move at certain velocities within a specified range of distances — along roads, rails, or air or sea lanes. (iv) Routes are sequences of hub visits "infixed" with travels along links, that is, a sequence staring with a hub (of origin), then a link, then a hub, etc., and ending with a (destination) hub. Hubs are like road intersections, train stations, airports and harbours, including production centers, warehouses, distribution centers and customer locations. Links are like road segments, rail tracks (between train stations), air lanes or sea lanes. (v) Sources and sinks are hubs.

[59] We have covered one facet of security extensively elsewhere [43, in [21]] and shall therefore not cover this aspect in this report.

[60] A network is a graph: hubs are nodes and links are edges.

W.6.5 Structure of Report

We shall therefore focus on the following concepts — some of which are *highlighted in this type font* above: Sect. W.7: *Transport Networks* of *hubs* and *links* (incl. *origins, destinations*) — covering both road, rail, air and sea transport nets; Sect. W.8: *Containers and Freight Items*; Sect. W.9: *Transport Companies, Vehicles and Timetables* (trucks, busses, trains, aircraft and sea vessels) and *timetables*; Sect. W.10: *Handling* (consignments, bill of ladings, waybills, et cetera); Sect. W.11: *Logistics Traffic*; Sect. W.12: *Senders and Receivers* (temporary storage before, during and after transport); and Sect. W.13: various miscellaneous issues (*packaging, tracing, notifications* *et cetera*).

W.7 Transport Networks

20. We shall introduce the notions of (transport) nets, hubs and links.

Sub-sets of a transport net may be road, rail, air traffic or sea vessel nets.

21. A transport net contains two or more hubs

22. and one or more links

Examples of hubs are: street intersections of road net, train stations of a rail net, airports of an air traffic net and harbours of a sea vessel net. Examples of links are: street segments between two intersections of road net, tracks between two train stations of a rail net, air lanes between two airports of an air traffic net and sea lanes between two harbours of a sea vessel net.

```
type
   20  N, H, L
value
   21  obs_Hs: N → H-set
axiom
   21  ∀ n:N • card obs_Hs(n) ≥ 2
```

```
value
   22  obs_Ls: N → L-set
axiom
   22  ∀ n:N • card obs_Ls(n) ≥ 1
```

W.7.1 Nets, Hubs and Links

Mereology of Nets

We wish to express how hubs and links are connected.

23. To express how hubs and links are connected we need identify hubs and links uniquely.

24. From a hub we can observe its unique hub identifier.

25. From a link we can observe its unique link identifier.

```
type
   23  HI, LI
value
   24  obs_HI: H → HI
   25  obs_LI: L → LI
axiom
   ∀ n:N,h,h′:H,l,l′:L •
   24     {h,h′}⊆obs_Hs(n) ⇒ (h≠h′ ⇒ obs_HI(h) ≠ obs_HI(h′)) ∧
   25     {l,l′}⊆obs_Ls(n) ⇒ (l≠l′ ⇒ obs_LI(l) ≠ obs_LI(l′))
```

Axioms 24–25 express uniqueness of identifiers.

26. From a hub we can observe the link identifiers of all the links connected to the hub.

27. From a link we can observe the hub identifiers of the two distinct hubs to which the link is connected.

```
value
   26  obs_LIs: H → LI-set
   27  obs_HIs: L → HI-set
axiom
       ∀ n:N, h:H, l:L • h ∈ obs_Hs(n) ∧ l ∈ obs_Ls(n) ⇒
   26      ∀ li:LI • li ∈ obs_LIs(h) ⇒ ∃ l′:L • l′ ∈ obs_Ls(n) ∧ obs_LI(l′)=li
   27      ∀ hi:HI • hi ∈ obs_HIs(l) ⇒ ∃ h′:H • h′ ∈ obs_Hs(n) ∧ obs_HI(h′)=hi
```

28. Given a net one can obtain all it link and all its hub identifiers.

29. Given a net and a link identifier of that net one can obtain the so-identified link.

30. Given a net and a hub identifier of that net one can obtain the so-identified hub.

```
value
   28  xtr_LIs: N → LI-set
   29  xtr_L: N → LI $\overset{\sim}{\to}$ L
   30  xtr_H: N → HI $\overset{\sim}{\to}$ H

   28  xtr_LIs(n) ≡ {obs_LI(l)|l:L•l ∈ obs_Ls(n)}
   28  xtr_HIs(n) ≡ {obs_HI(h)|h:H•h ∈ obs_Hs(n)}

   29  xtr_L(n)(li) ≡ let l:L•l ∈ obs_Ls(n)∧li=obs_LI(l) in l end
                      pre li ∈ xtr_LIs(n)
   30  xtr_H(n)(hi) ≡ let h:H•h ∈ obs_Hs(n)∧hi=obs_HI(h) in h end
                      pre hi ∈ xtr_HIs(n)
```

Reference Nets

31. A net defines a reference net.

31. A reference net maps hub identifiers to sets of one or more link identifiers.

31. Thus from a net one can calculate its reference net: For every hub its identifier is mapped into the link identifiers observable from that hub.

```
type
   31    RN = HI →m (HI −m> LI-set)
value
   31.1   calc_RN: N → RN
   31.2   calc_RN(n) ≡
   31.3     [ hi ↦ [ hi′ ↦ {obs_LI(l)
   31.4          | l:L•l ∈ obs_Ls(n)∧hi ∈ obs_HIs(l)∧hi′ ∈ obs_HIs(l)\{hi}} ]
   31.5          | h:H•h ∈ obs_Hs(n)∧hi=obs_HI(h) ]
```

- We refer to
  - the hi definition set elements (leftmost hi of 31.3) of the reference net as the *origin hub* identifier;
  - the rightmost hi′ of 31.3 as a *target hub* identifier, and
  - the range set of link identifiers as 'the range set of link identifiers' !

32. A reference net, n_{s_r}, is a sub-reference net, n_r, if

    a) the origin hub identifiers, hi, of n_{s_r}, form a subset of the origin hub identifiers of n_r;

    b) the set of target hub identifiers, hi′, for origin hub identifier hi, of n_{s_r}, form a subset of those of n_r; and

    c) the range set of link identifiers in n_{s_r} is a subset of those of the corresponding range set of link identifiers in n_r.

```
value
   32    is_sub_ref_net: RN × RN → Bool
   32    is_sub_ref_net(rn′,rn) ≡
   32a      dom rn′ ⊆ dom rn ∧
   32b      ∀ hi:HI • hi ∈ dom rn ⇒ dom rn′(hi) ⊆ dom rn(hi) ∧
   32c      ∀ hi′:HI • hi′ ∈ dom rn′(hi) ⇒ (rn′(hi))(hi′)⊆(rn(hi))(hi′)
```

Attributes of Hubs and Links

33. Hubs have a number of attributes:

    a) spatial (i.e., geographic) location which, since we simply hubs a points, can be represented by three coordinates: longitude, latitude and altitude;

    b) duration (time) of

       i. entering,
       ii. traversing and
       iii. leaving

       a hub[61];

    c) et cetera.

34. Links have a number of attributes:

    a) spatial (i.e., geographic) location which, since we simply links as lines that can be described in the way that we describe Bezier curves[62];

    b) length;

    c) cost of transporting a unit of freight volume per unit of length along the link;

    d) duration (time) of

       i. entering,
       ii. traversing and
       iii. leaving

       a link[63];

    e) et cetera.

type
33a HLoc
33b TimDur
33c ...
34a Bezier
34b Length
34c Cost
value
33a obs_HLoc: H → HLoc
33b obs_InTime: H×... → TimDur
34a obs_LLoc: L → Bezier
34b obs_Length: L → Length

[61] The time intervals are specific to each hub and depends on direction of traversal, type of vehicle and its load status

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

[63] We disregard the possibility that traversing a link in one direction may take longer time than traversing it in the opposite direction.

```
34c  obs_Cost: L → Cost
34d  obs_InTime: L×... → TimDur
34e  ...
```

W.7.2 Routes

Hub Traversals, Entries and Exits

35. A hub traversal is here represented by a triple

    a) a(n input) link identifier, ili,

    b) a hub identifier, hi and

    c) a(n output) link identifier, oli,

    such that

    d) the identifiers are those of links and hubs of the network,

    e) the two link identifiers are observable from the hub identified by hi.

36. A hub “entry” is here represented by the pair of the first two elements of a hub traversal.

37. A hub “exit” is here represented by the pair of the two two elements of a hub traversal.

```
type
  35  HubTrav = LI × HI × LI
  36  HubEntry = LI × HI
  37  HubExit = HI × LI
axiom
  35d  ∀ n:N, (ili,hi,oli):HubTrav • (ili,hi,oli) ∈ HubTraversals(n)
  35b  ∀ n:N, (ili,hi):HubEntry • (ili,hi) ∈ HubEntries(n)
  35c  ∀ n:N, (oli):HubExit • (hi,oli) ∈ HubExits(n)
  ... et cetera
value
  HubTraversals: N → HubTrav-set
  HubTraversals(n) ≡
     {(ili,hi,oli)|(ili,hi,oli):HubTrav, h:H • hi=obs_HI(h)∧{ili,oli}⊆obs_LIs(h)}
  HubEntries: N → HubEntry-set
  HubEntries(n) ≡ {(li,hi)|(ili,hi):HubEntry, h:H • hi=obs_HI(h)∧li ∈ obs_LIs(h)}
  HubExits: N → HubExit -set
  HubExits(n) ≡ {(hi,li)|(hi,oli):HubExit, h:H • hi=obs_HI(h)∧li ∈ obs_LIs(h)}
```

Link Traversals, Entries and Exits

38. A link traversal is here represented by a triple
    a) a(n input) hub identifier, ihi,
    b) a link identifier, li and
    c) a(n output) hub identifier, ohi,

    such that

    d) the identifiers are those of links and hubs of the network,
    e) the two hub identifiers are observable from the link identified by hi.
39. A link “entry” is here represented by the pair of the first two elements of a link traversal.
40. A link “exit” is here represented by the pair of the two two elements of a link traversal.

```
type
   38 LinkTrav = HI × LI × HI
   39 LinkEntry = HI × LI
   40 LinkExit = LI × HI
axiom
   38d ∀ n:Nii, (ihi,li,oli):HubTrav • (ihi,li,ohi) ∈ LinkTraversals(n)
   38b ∀ n:N, (ihi,li):HubEntry • (ihi,li) ∈ LinkEntries(n)
   38c ∀ n:N, (li,ohi):HubExit • (li,ohi) ∈ LinkExits(n)
   ... et cetera
value
   LinkTraversals: N → LinkTrav-set
   LinkTraversals(n) ≡
       {(ihi,li,ohi)|(ihi,li,ohi):LinkTrav, l:L • li=obs_LI(h)∧{ihi,ohi}=obs_HIs(l)}
   LinkEntries: N → LinkEntry-set
   LinkEntries(n) ≡ {(ihi,li)|(ihi,li):LinkEntry, l:L • li=obs_LI(l)∧hi ∈ obs_HIs(l)}
   LinkExits: N → HubExit -set
   LinkExits(n) ≡ {(li,ohi)|(li,ohi):LinkExit, l:L • li=obs_HI(l)∧hi ∈ obs_HIs(l)}
axiom
   ...
```

First and Last Hubs of Link Traversals

41. If (hi,li,hi′) is a link traversal then
    a) hi identifies the *first* hub of that traversal, and

b) hi′ identifies the *last* hub of that traversal

```
value
    41a  fstHI: LinkTrav → HI
    41a  fstHI(hi,li,hi′) ≡ hi
    41b  lstHI: LinkTrav → HI
    41b  lstHI(hi,li,hi′) ≡ hi′
```

Routes

42. Routes are sequences of one or more link traversals and defined as follows:

    a) **Basis Clause:** A sequence of one link traversal is a route.

    b) **Induction Clause:** If r and r' are routes such that the

        i. last hub identifier of the last traversal of r

        ii. is the same as the first hub identifier of the first traversal of r'

        iii. then $r^\frown r'$ is a route.

    c) **Extremal Clause:** Only sequences of link traversals that can be formed from a finite number of uses of the basis and the induction clauses are routes.

```
type
    42  Route′,R′ = LinkTrav*
    42  Route,R = {|r:R′ • len r≥1 ∧ wf_R(r)|}
value
    42  wf_R: R′ → Bool
    42  wf_R(r) ≡
            case r of
    42a       ⟨⟩ → true,
    42a       ⟨(hi,li,hi′)⟩ → true,
    42b       r⌢⟨(hi,li,hi′)⟩⌢⟨(hi″,li′,hi‴)⟩⌢r′ → wf_R(r)∧wf_R(r′)∧hi′=hi″
            end
    gen_Rs: N → R-infset
    gen_Rs(n) ≡
    42a  let rs = {⟨lt⟩|lt:LinkTrav•lt ∈ LinkTraversals(n)}
    42b            ∪ {r⌢r′|r,r′:R • {r,r′}⊆rs∧lstHI(r(len r))=fstHI(r′(1))} in
            rs end
```

The gen_Rs function generates all routes of a network. For technical reasons we have defined the well-formedness of routes predicate, wf_R, to also apply to empty sequences of link traversals although they are not (proper) routes. Whereas the definition of routes did not refer to the net whereby well-formedness of routes was just a "syntactic" matter, the function that generates routes (from a net) secures "semantic" well-formedness of routes.

43. Given a net and two distinct hub identifiers (of that net)
    a) one can calculate whether there is a route from the one identified hub to the other (and, since all links are two way links, vice versa);
    b) and, if there is such a route then one can calculate the set of all such routes.

```
value
  43a  is_route: N × (HI×HI) → Bool
  43a  is_route(n,(fhi,thi)) ≡ {r|r:R•fstHI(r(1))=fhi∧lstHI(r(len r))=thi}≠{}
  43b  routes: N × (HI×HI) → R-set
  43b  routes(n,(fhi,thi)) ≡ {r|r:R•fstHI(r(1))=fhi∧lstHI(r(len r))=thi}
```

44. Since all links are two-way links one can speak of reverse links.

```
value
  44  reverse_route: R → R
  44  reverse_route(r) ≡
  44    case r of
  44      ⟨⟩ → ⟨⟩,
  44      ⟨(hi,li,hi′)⟩⁀r′ → reverse_route(r′)⁀⟨(hi′,li,hi)⟩
  44    end
```

W.7.3 Connected and Disconnected Nets

We assume, throughout, that all links can be traversed in both directions, that is, there are no *cul de sacs* (*sackgasse*, “blind” streets).

45. A net is said to be connected if for every pair of distinct hubs of the net there is a route that connects them, i.e., from the one hub to the other.

46. Two otherwise, i.e., respectively connected nets, n_i, n_j, are said to be disconnected if they share no hubs and links.

47. A net defines a set of one or more disconnected nets.

```
value
  45  is_connected: N → Bool
  45  is_connected(n) ≡
  45    ∀ h,h′:H •{h,h′}⊆obs_Hs(n)⇒is_route(n,(obs_HI(h),obs_HI(h′)))

  46  are_disjoint: N×N → Bool
  46  are_disjoint(n,n′) ≡
```

```
46    obs_Hs(n)∩ obs_Hs(n′)={}∧obs_Ls(n)∩ obs_Ls(n′)={}

47  disconnected_nets: N → N-set
47  disconnected_nets(n) as ns
47    post ∪{n|n:N•n ∈ ns}=n
```

W.7.4 Subnets

48. A given net, n, defines a set of one or more subnets $\{n_1, n_2, \ldots, n_m\}$.

49. A net, n_s, is a subnet of another net, n,
    a) if the reference net, nr_s, of n_s
    b) is a sub-reference-net, rn, of n.

```
48  subnets: N → N-set
48  subnets(n) as ns
48    post ∀ n′:N • n′ ∈ ns ⇒ sub_ref_net(calc_RN(n′),calc_RN(n))
49  is_subnet: N × N → Bool
49  is_subnet(ns,n) ≡ ns ∈ subnets(n)
```

W.7.5 Route Attributes

50. Routes have lengths — “measured” as the sum of the lengths of all the links denoted by link traversal link identifiers.
    a) Thus a route from a first hub h to a last hub h'
    b) has same length as the reverse route (from a first hub h' to a last hub h).

51. Routes have travel times — “measured” as the sum of the travel times of all the links denoted by link traversal link identifiers.

52. Given two distinct hubs (say, by their hub identifiers) one can calculate
    a) the shortest route(s) between these two hubs; and
    b) the fastest route(s) between these two hubs given the attributes of the vehicle which is supposed to travel the route.

```
value
  50  length: R × N → Length
  50  +: Length × Length → Length
  50  length(r,n) ≡
  50    case r of
```

```
50      ⟨⟩ → 0,
50      ⟨(_,li,_)⟩⌢r′ → obs_Length(xtr_L(n)(li)) + length(r′,n)
50    end
51  travel_time: R × N → Time
51  +: Length × Length → Length
51  travel_time(r,n) ≡
51    case r of
51      ⟨⟩ → 0,
51      ⟨(_,li,_)⟩⌢r′ → obs_TravTime(xtr_L(n)(li)) + travel_time(r′,n)
51    end
```

One can prove:

```
lemma:
    ∀ n:N,r:R • r ∈ routes(n) ⇒
        length(r)(n) = length(reverse_route(r))(n)
        travel_time(r)(n) = travel_time(reverse\_route(r))(n)
```

Some “interesting” functions:

```
value
    52a  shortest_route: N × (HI×HI) → R×Length
    52a  shortest_route(n,(fhi,thi)) ≡
    52a    let rs = routes(n,(fhi,thi)) in
    52a    {r|r:R•r ∈ rs∧∼∃ r′:R•r′isin rs∧length(r′)<length(r)}
    52a    end

    52b  fastest_route: N × (HI×HI) → R×Days
    52b  fastest_route(n,(fhi,thi)) ≡
    52b    let rs = routes(n,(fhi,thi)) in
    52b    {r|r:R•r ∈ rs∧∼∃ r′:R•r′isin rs∧travel_time(r′)<travel_time(r)}
    52b    end

    52b  least_costly_route: N × (HI×HI) → R×Cost
    52b  least_costly_route(n,(fhi,thi)) ≡
    52b    let rs = routes(n,(fhi,thi)) in
    52b    {r|r:R•r ∈ rs∧∼∃ r′:R•r′isin rs∧cost(r′)<cost(r)}
    52b    end
```

W.7.6 Link, Hub, Route and Net Modalities

Link and Hub Modalities

53. With a link we now associate a further attribute: that of is transport modality which is either that of `road`, `rail`, `air`, or `sea`.

54. To provide for “smooth” transfer of freight from respective vehicle modalities (`truck`, `train`, `air-cargo`, respectively `vessel`),

55. we expect hubs connected to n links to have up to four hub modalities, that is, any subset of the set {`truck`,`train`,`air-cargo`,`vessel`}.

```
type
  53 TM == road | rail | air | sear
  54 VM == truck | train | aircargo | vessel
value
  53 obs_TM: Link → TM
  54 obs_VM: Vehicle → VM
  55 obs_TMs: Hub → TM-set
```

where we presuppose the vehicle phenomenon.

56. Links incident upon a hub in a net must be of a modality also represented by that hub, and for all links and hubs.

57. A hub of a net must have exactly the modalities of the links connected to that hub.

```
axiom
  ∀ n:N, l:L, h:H •
     l ∈ obs_Ls(h)∧h ∈ obs_Hs(h)∧obs_LI(l) ∈ obs_LIs(h)∧obs_HI(h) ∈ obs_HIs(l)⇒
  56      obs_TM(l) ∈ obs_TMs(h) ∧
  57      ∀ li:LI • li ∈ obs_LIs(h) ⇒
  57        obs_TM(xtr_LI(li)(n))∈ obs_TMs(h)
```

Route Modalities

58. A route is said to be a single modularity route if all its links are of the same modality.

59. A route is said to have the set of 1, 2, 3 or 4 modalities that are those of its links.

```
value
  58 is_sgl_TM: Route → N → Bool
  59 route_TMs: Route → N → RM-set

  58 is_sgl_TM(r)(n) ≡
  58   ∀ i,j:Nat • {i,j}⊆indes(r)
  58     let (_,li,_)=r(i),(_,lj,_)=r(j) in
  58     obs_TM(xtr_L(n)(li))=obs_TM(xtr_L(n)(lj)) end

  59 route_TMs(r)(n) ≡
  59   {obs_TM(xtr_L(n)(li))|(_,li,_):LTrav • (_,li,_)∈ elems r}
```

Net Modalities

60. A net is said to be a single modality net if all its routes are of the same modality.

61. The modality of a net is the set of modalities of its routes.

```
value
  60  is_sgl_TM: N → Bool
  60  is_sgl_TM(n) ≡
  60    ∀ r,r′:R • {r,r′}⊆routes(n) ⇒
  60      route_TMs(r)=route_TMs(r′)∧card route_TMs(r)=1

  61  net_modalities: N → TM-set
  61  net_modalities(n) ≡
  61    ∪{route_TMs(r)(n)|r:R • r ∈ routes(n)}
```

W.8 Containers and Freight Items

W.8.1 Containers

62.

63.

64.

65.

```
  62
  63
  64
  65
```

W.8.2 Freight Items

66.

67.

68.

69.

```
  66
  67
  68
  69
```

W.9 Transport Companies, Vehicles and Timetables

W.9.1 Transport Companies

For simplicity, but with no loss of generality, we assume that each company is "mono-modal", that is offering either

truck, train, aircargo, or vessel

transport; and we assume that all such transport is line transport, that is, freight can be carried, without reloading, along either of a standard set of routes. For each such line there is a timetable which repeats itself at regular intervals.

More precisely:

70. A transport company operates
    a) a finite number of one or more vessels, identified by their unique vessel identifiers, and
    b) is focused on a a finite number of one or more timetables. and
    c) has a unique (transport company) identification.

```
type
    70  TransComp
    70a  Vid
    70b  Timetable, TT
    70c  TCId
value
    70a  obs_VIds: TransComp → VId-set
    70b  obs_Timetable, obs_TT: TransComp → Timetable-set
    70c  obs_TCId: TransComp → TCId
```

W.9.2 Vehicles

Without loss of generality we assume all vessels to be container vessels.

71. There are vehicles.
72. Vehicles have unique vehicle identification
    a) from which one can observe the identification of the transport company which operates the vehicle.
73. A vehicle is either a truck, a train, an aircargo (aircraft, aircargo for short) or a vessel.

74. A vehicle location is either at

    a) at a hub, identified by that hub's unique identifier, or

    b) or along a link (identified by that link's unique identifier), from some hub (identified by that hub's unique identifier)

    c) a fraction, f, of the distance to another hub (identified by that hub's unique identifier).

75. From a vehicle one can observe which freight the vehicle is conveying (at the moment, the time, of being observed), where we simplify the freight observation to

    a) observing the set of the bill-of-ladings for each freight item and

    b) the identification of the container in which it is packed.

76. One might wish to add such possibly observable information as:

    a) expected arrival (date and time) at next hub,

    b) velocity,

    etc.

```
type
  71     Vehicle, CId, Velocity
  72     VId
  72a  TCId
  73     Vehicle_type == truck | train | aircargo | vessel
  74     VLoc =  VHLoc | VLLoc
  74a  VHLoc == atH(hi:HI)
  74b  VLLoc == onL(thi:HI,li:LI,f:Frac,thi:HI)
  74c   Frac = {|r:Real•0<r<1|}
  75a   BoL
value
  72      obs_VId: Vehicle → VId
  72a   obs_TCId: VId → TCId
  73      obs_Vehicle_type: Vehicle → Vehicle_type
  74      obs_VLoc: Vehicle → VLoc
  75a   obs_BoLs: Vehicle → BoL-set
  75b   obs_Cid: Vehicle × BoL ∼→ CId
  76a   obs_Arrival: Vehicle → (Date × Time)
  76b   obs_Velocity: Vehicle → Velocity
```

W.9.3 Timetables

77. Timetables are wellformed relative to a net.[64]

78. There is a concept of timetable identifiers.

79. A timetable

    a) has a timetable identifier;

    b) features a reference net; and finally the timetable also

    c) lists a sequence of timed *link traversals*

80. From a timetable identifier one may observe the identifier of the transport company which operates a freight service according to that timetable.

81. From a timetable identifier one may observe the identification of the vehicle that has been allocated to serve the timetabled schedule.

Two or more timetables of different names may feature identical timetables — in which case only the observable transport company identifiers are different[65].

```
value
   77      n:N
type
   78      TTId
   79      TT′ =
   79a        TTId
   79b          × RN
   79c            × TLT*
value
   80       obs_TCId: TTId → TCId
   81       obs_VId: TTId → Vid
```

82. Timetables must be well-formed, that is, the link traversals of a timetable

    a) must visit exactly $m+1$ hubs where m is the length of the list of link traversals;

    b) must be commensurate with the timetable reference net (‘commensurability’ is expressed by the tt_is_ref_net_commensurable predicate below),

    c) the timetable link traversal list must be well-formed, and,

[64] When in formula line 77 we postulate a net: **value** n:N, then that value declaration should be seen as ranging over any net.

[65] that is: “competition to the line”

d) given a net, n, and a timetable, tt, the timetable reference net, rn, must be commensurate with the net n (that is, refnet_is_tt_commensurable(rn,n)).

```
type
  82      TT = {|tt:TT′•wf_TT(tt)(n)|}
value
  82      wf_TT: TT′ → N → Bool
  82      wf_TT(tt:(_,rn,tltl))(n) ≡
  82a        card{hi|(hi,li,hi′):LTrav•(_,(hi,li,hi′),_)∈ elems tltl}=len tltl+1 ∧
  82b        tt_is_refnet_commensurable(tt) ∧
  82c        wf_TLT*(tltl) ∧
  82d        refnet_is_net_commensurable(rn,n)
```

83. (cf. Item 82b on the preceding page.) Commensurability of a timetable's lists of link traversals with respect to that timetable's reference net is defined as follows:

    a)

    b)

    c)

84. (cf. Item 82d.) Commensurability of a timetable's reference net with respect to the (global) net is defined as follows:

    a)

    b)

    c)

```
  83      tt_is_refnet_commensurable: TT → Bool
  83a   tt_is_refnet_commensurable(_,rn,tltl) ≡
  83b
  83c

  84      refnet_is_net_commensurable: RN × N → Bool
  84      refnet_is_net_commensurable(rn,n) ≡
```

85. Instead of representing a set of timetables as a set of the timetables as defined above we may represent them as a map from timetable identifiers to pairs of reference net and lists of timed link traversals.

86. Such maps must be well-formed.

87. The well-formedness conditions can be referred back to well-formednes of the previously defined timetables.

```
type
  85  $TTs' = TTId \xrightarrow[m]{} RN \times TLT^*$
  86  $TTs = \{| tts{:}TTs' \bullet wf\_TTs(tts)(n) |\}$
value
  87  $wf\_TTs\colon TTs' \to N \to \mathbf{Bool}$
  87  $wf\_TTs(tts)(n) \equiv$
  87    $\forall\, ttid{:}TTId \bullet ttid \in \mathbf{dom}\ tts \Rightarrow$
  87      $\mathbf{let}\ (rn.tltl) = tts(ttid)\ \mathbf{in}\ wf\_TT(ttid,rn,tltl)(n)\ \mathbf{end}$
```

Timed Link Traversals

88. Timed link traversals, besides the link traversal, contains the date/times of entering and leaving the link and the

89. cost to the user (sender/receiver) per unit of freight volume for getting such a unit of freight volume transported along the identified link.

90. Well-formed timed link traversals must be understood in the context of the global net[66] in which transport takes place.

```
value
  fn:66  n:Net
type
  88  $TLT' = (Date \times Time) \times LinkTrav \times Cost \times (Date \times Time)$
  89  Cost −−− see also Item 34c on page 295
  90  $TLT = \{| tlt{:}TLT' \bullet wf\_TLT(tlt)(n) |\}$
```

91. For each timed link traversal the date/time of entering the link must precede the date/time of leaving the link;

92. the interval, TI, between these date/times must be commensurate with the length and "normative" velocity of the identified link; and

93. the user cost of transporting a unit of freight along the link must be commensurate with the normative cost of moving a vehicle along that link.

[66] That is why we bring the **value** declaration n:Net in formula line fn:66 Page 308.

```
value
  90    wf_TLT: TLT′ → N → Bool
  90    wf_TLT(tlt:((d,t),(hi,li,hi′),c,(d′,t′)))(n) ≡
  91      precede((d,t),(d′,t′)) ∧
  92      commensurate_time(interval((d,t),(d′,t′)),obs_TravTime(xtr_L(n)(li))) ∧
  93      commensurate_cost(c,xtr_L(n)(li))
  91    precede: (Date×Time)×(Date×Time) → Bool
type
  92    TI[67]
value
  92    commensurate_time: TI × TI → Bool
  92    interval: (Date×Time) × (Date×Time) → TI
  93    commensurate_cost: Cost × L → Bool
  93    commensurate_cos(c,l) ≡
  93      ... c = f(obs_Length(l),obs_Cost(l),...) ...
  93      [ where f is a real valued function over two arguments:  ]
  93      [ length and cost typically yielding a value larger than 1 ]
```

94. Lists of timed link traversals must be time-wise ordered:
    a) for all adjacent positions, i and i+1, in the list
    b) the ith departure date/time and the i+1st arrival time
    c) most have the former precede the latter.
    d) the reference net (implicitly) expressed by the list of timed link traversals must be a sub reference net of the timetable reference net.

```
value
  94     wf_TLT*: TLT* → Bool
  94     wf_TLT*(tltl) ≡
  94a     ∀ i:Nat•{i,i+1}⊆inds tltl⇒
  94b        let (_,_,_,(d,t))=tltl(i),((d′,t′),_,_,_)=tltl(i+1) in
  94c        precede((d,t),(d′,t′)) end ∧

  94d     is_sub_refnet(xtr_RN(tltl),rn)

  94d     xtr_RN: TLT* → RN
  94d     xtr_RN(tltl) ≡ [ hi↦[ hi′↦{li} ]|(hi,li,hi′):LTrav•(hi,li,hi′)∈ elems tltl ][68]
```

[67] Time intervals arise when one date/time is subtracted from another date/time. One can add time intervalsto get a time interval ; one can add a time interval to a date/time to obtain a date/time; one can multiply a time interval with a number (whether natual or real; etc.

[68] The constraint expressed in Item and formula line 82a secures that there is only one link in the list of link traversals, hence $\{li\}$, between hub identifiers hi and hi'.

W.10 Handling

We shall look at only a single aspect of handling, namely that of responding to a request from sender c: provide an optimal shipping, s_o, of such-and-such, a, freight, f, from origin h to receiver c', destination h' at this time, t, or at some earliste time, t', thereafter; a stands for attributes of freight f.

W.10.1 Shipping Requests and Responses

Shipping Requests

95. A shipping request contains the following information:
    a) Name, c, of sender;
    b) origin, h_i, of freight, i.e., where to be sent from;
    c) destination, h'_j, of freight, i.e., where to be sent to;
    d) attributes, a, of freight;
    e) Name, c', of receiver;
    f) some optimality criterion: *“fastest”* route, *“least costly”* route, or *“earliest arrival date”*, or other; and
    g) the date/time of submission of the request.
96. A negative response to a shipping request has the form of a `‘‘request is not feasible’’`.

```
type
          SndrId, RcvrId, FreightAttrs, Neg_Resp
  95      Ship_Req′ =
  95a         SndrId
  95b         × HI                    [ from ]
  95c         × HI                    [ to ]
  95d         × RcvrId
  95e         × Freight_Attrs
  95f         × Optimality
  95g         × (Date × Time)         [ earliest send date ]
  95f     Optimality == fastest|cheapest|earliest_arrival|...
  96        Neg_Resp        × TT*
```

97. For a shipping request, shipreq:Ship_Req′, to be well-formed
    a) the sender and receiver identifiers must be different and
    b) the origin and destination hubs must be different.

```
value
  97    wf_Ship_Req: Ship_Req → Bool
  97    wf_Ship_Req(sid,hi,hi′,rid,fas,o,dt) ≡
  97a     sid ≠ rid
  97b     hi ≠ hi′
```

Positive Shipping Request Responses: Waybills

98. A positive response to a shipping request has the form of a waybill, WB, which contains the following information:

    a) sender's identification, *c*;

    b) from where, hi:HI, freight is to originate (fetched);

    c) to where, hi′:HI, freight is to be destined (delivered);

    d) the receiver's identification, *c*′;

    e) attributes, a, of the freight;

    f) the list of one or more timetables, i.e., the possibly optimal shipping;

    g) the total cost of shpping;

    h) the date/time of start of transport;

    i) the date/time of earliest delivery of freight; and

    j) the total elapsed time interval of transport, measured in number of days.

```
type
  98j    Days
  98       WB =
  98a        SndrId
  98b        × HI                  [from]
  98c        × HI                  [to]
  98d        × RcvrId
  98e        × Freight_Attrs
  98f        × TT*
  98g        × Cost
  98h        × (Date × Time)    [send date]
  98i        × (Date × Time)    [receipt date]
  98j        × Days                [duration]
```

Waybill Wellformedness

Well-formedness of waybills must be expressed in terms of the global transportation net and the set of timetables available to the shipping company which produces the waybill.

99. The waybill is well-formed in the context of the net and a set of shipping agent timetables
    a) waybill sender and receiver identifications must be different;
    b) waybill from and to hub identifications must be different;
    c) waybill timetable list must not be empty;
    d) if the timetable list of the waybill is well-formed with respect to the set of shipping agent timetables;
    e) if the first hub identifier of the timetable list of the way bill equals the ‘from’ hub identifier of the waybill and the last hub identifier of the timetable list of the way bill equals the ‘to’ hub identifier of the waybill;
    f) waybill specified cost must be commensurate with the costs of each of the transports stated in the waybill timetable list;
    g) freight departure date/time must precede freight arrival date/time; and
    h) the total elapsed time interval of transport must be commensurate with the interval between the freight departure date/time and freight arrival date/time.

value

```
 99   wf_WB: WB → (N × TTs) → Bool
 99   wf_WB(sid,fhi,thi,rid,fas,ttl,c,sdt,rdt,dur)(n,tts) ≡
 99a    sid ≠ rid ∧
 99b    fhi ≠ thi ∧
 99c    ttl ≠ ⟨⟩ ∧
 99d    wf_tt_arguments(ttl,tts) ∧
 99e    from_to((fhi,thi),ttl) ∧
 99f    commensurate_costs(c,ttl) ∧
 99g    precede(sdt,rdt) ∧
 99h    commensurate_duration((sdt,rdt),duration(ttl))
```

100. (99e) The timetable arguments (contained in `ttl` and `tts`) are well-formed
    a) if the timetables mentioned in `ttl` all have distinct timetable identifiers;
    b) if the timetables mentioned in `ttl` are defined in `tts`;
    c) if the list of timed link traversals contained in the time table named `ttid` in `ttl` is a sublist of the time table named `ttid` in `tts`;

d) if the list of timed link traversal lists are connected;

e) if the sublists do not specify the revisit hubs.

value

```
100   wf_tt_arguments: TT* × TTs → Bool
100   wf_tt_arguments(ttl,tts)
100a    let ttids = {ttid|i:Nat•i ∈ inds ttl⇒(ttid,_,_)=ttl(i)} in card ttids = len ttl ∧
100b    ttids ⊆ dom tts end ∧
100c    ∀ i:Nat•i ∈ inds ttl ⇒
100c      let (ttid,rn,tltl)=ttl(i) in let (rn′,tltl′)=tts(ttid) in is_sublist(tltl,tltl′) end end ∧
100d    ∀ i:Nat•{i,i+1}⊆inds ttl ⇒ lstHI((ttl(i))(len ttl(i)))=fstHI((ttl(i+1))(1)) ∧
100e    no_hub_revisits(ttl)
```

101. (102) A timed link traversal list, tltl, is a sublist, is_sublist(tltl,tltl′), of another timed link traversal list, tltl′,

a) if there are two indices into tltl′

b) such that the elements in tltl′ between and including these index positions equals tltl.

value

```
101     is_sublist: TLT* × TLT* → Book
101     is_sublist(tltl,tltl′) ≡
101a      ∃ i,j:Nat • i≤j ∧ {i,j}⊆inds tltl′ ⇒
101b        tltl = ⟨tltl′(k)|i≤k≤j⟩
```

102. The no_hub_revisits predicate[69] is specified as follows:

a) first a single list, ltlt, of time link traversals is constructed from the **conc**atenation of the list of time link traversals contained in each of the timetables of the waybill;

b) then the set, his, of distinct hub identifiers of ltlt is constructed;

c) the number of hub identifiers in that set, that is, **card** his, must be equal to one plus the length of the consolidated list ltlt — a larger number would mean that the individual lists of time link traversals contained in each of the timetables of the waybill were not connected, and if it was smaller then there would be revisits.

[69] The no_hub_revisits predicate tests that the sublists of timed link traversal lists contained in its single ttl argument do not describe the revisit hubs

value

```
102    no_hub_revisits: TT* → Bool
102    no_hub_revisits(ttl) ≡
102a     let ltlt = conc⟨tlti|i:[1..len ttl]•let (_,_,tlti′)=ttl(i) in tlti=tlti′ end⟩ in
102b     let his = {hi,hi′|hi:HI•(hi″,_,hi‴):LinkTrav•(hi,_,hi′)∈ elems ltlt∧hi=hi″∧hi′=hi‴} in
102c     card his = len ltlt+1 end end
```

103. (99e) The predicate from_to expresses

  a) that the first hub identifier of the timetable list of the way bill equals the 'from' hub identifier of the waybill, and

  b) that the last hub identifier of the timetable list of the way bill equals the 'to' hub identifier of the waybill;

value

```
103    from_to: (HI×HI) × TT* → Bool
103    from_to((fhi,thi),ttl) ≡
103a     fhi = fstHI((ttl(1))(len ttl(1))) ∧
103b     thi = lstHI((ttl(len ttl))(len ttl(len ttl)))
```

104. The commensurate_costs(c,accumulated_cost(ttl)) (99f) predicate

  a) sums the costs of the summing of costs of each individual list of timed (and costed) link traversals given in each of the waybill timetables

  b) and compares that to the cost directly described in the waybill; the comparison is non-determinate, that is, we do not describe precise means of comparing these costs.

value

```
104   commensurate_costs: Cost × Cost → Bool
104   commensurate_costs(c,ttl) ≡
104a    let costs = sum_of_sums_of_costs(ttl) in
104b    costs ≃ cost end

      ≃: Cost × Cost → Bool
```

105. The sum_of_sums_of_costs function calculates its cost result by recursion:

  a) if the argument list is empty the cost is zero (0),

  b) else the cost is the sum of the cost described in the first link traversal and the sum_of_sums_of_costs of the rest of the argument list.

value

```
105  sum_of_sums_of_costs: TT* → Cost
105  sum_of_sums_of_costs(ttl) ≡
105a     if ttl = ⟨⟩ then 0 else
105b     let (_,_,c,_) = hd ttl in c ⊕ sum_of_sums_of_costs(tl ttl) end end

⊕: Cost × Cost → Cost
```

106. The precede(sdt,rdt) (99g) predicate is left undefined.

   Once a specific representation of dates and time has been decided upon one can then easily define this function.

value

```
106  precede: (Date×Time) × (Date×Time) → Bool
106  precede(sdt,rdt) ≡ sdt ≪ rdt

≪: (Date×Time) × (Date×Time) → Bool
```

107. The commensurate_duration((sdt,rdt),duration(ttl)) (wfwbi) predicate also requires a specific representation of dates and time in order to be calculated, that is:

   a) one must somehow subtract sdt from rdt

   b) and then perform the commensurateness test.

value

```
107  commensurate_duration: ((Date×Time)×(Date×Time))×Days → Bool
107  commensurate_duration((sdt,rdt),duration(ttl)) ≡
107a     let dur = rdt − sdt in
107b     dur ≃ duration(ttl) end

     duration: TT* → Days
     duration(ttl) ≡
         if ttl = ⟨⟩ then 0
         else let (dt,_,_,dt′) = hd ttl in (dt′ ⊖ dt) ⊕ duration(tl ttl) end end

     ⊖: (Date×Time)×(Date×Time) → Days
     ⊕: Days × Days → Days
```

W.10.2 Generation of Waybills

108. A well-formed shipping request (sid,fhi,thi,rid,fas,o,dt) in the context of a net, n,

109. and a set of transport companies' timetables, tts, now denotes, $\mathcal{M}$, a set, wbs, of n waybills: { wb_1, wb_2, ..., wb_i, ..., wb_n } where individual wb_is are of the form ($\mathsf{sid,fhi,thi,rid,fas,ttl}_i,\mathsf{c}_i,\mathsf{sdt}_i,\mathsf{rdt}_i,\mathsf{dur}_i$)

110. which all satisfy wf_WB(sid,fhi,thi,rid,fas,ttl,c,sdt,rdt,dur)(n,tts).

```
108  ℳ: Ship_Req → (Net × TTs) → WB-set
109  ℳ(sid,fhi,thi,rid,fas,o,dt)(n,tts) as wbs
108    pre: wf_Ship_Req(sid,fhi,thi,rid,fas,o,dt)(n)
110    post: ∀ wb:WB • wb ∈ wbs ⇒ wf_WB(wb)(n,tts)
```

111. The set of optimal waybills depend on the optimality criterion, o:

    a) if o=`fastest` then the set of waybills with the same smallest duration, dur is chosen;

    b) if o=`cheapest` then the set of waybills with the same lowest cost, c is chosen; and

    c) if o=`earliest_arrival` then the set of waybills with the same earliest arrival date/time, rdt is chosen.

```
111   optimal_WBs: WB-set → Optimality → WB-set
111   optimal_WBs(wbs)(o) ≡
111     {wb|wb:WB • wb ∈ wbs ⇒
111         let (sid,fhi,thi,rid,fas,ttl,c,sdt,rdt,dur) = wb in
111         ∼∃ wb′:(sid,fhi,thi,rid,fas,ttl,c′,sdt,rdt′,dur′):WB•wb′ ∈ wbs ∧
111           case o of
111a              fastest → dur′ ≺ dur,
111b              cheapest → c′ ≺ c,
111c              earliest_arrival → precede(rdt,rdt′)
111         end end}

     ≺: (Days×Days)|(Cost×Cost) → Bool
```

W.11 Logistics Traffic

112. By logistics traffic, traf:TRAFFIC, we mean a continuous function from time to pairs of nets and vehicle positions.

113. That continuous function must satisfy some well-formedness conditions.

```
value
   n:N
type
   112  TRAFFIC′ = T → (N × (Vehicle →m VLoc))
   113  TRAFFIC = {|tra:TRAFFIC′•wf_TRAFFIC(tra)(n)|}
```

114. The well-formedness conditions for logistics traffics are:

    a) If at two times, close to one another, a vehicle is in the traffic — at both of these times — then that vehicle is in the traffic at any time beween the two times.

    b) At no time can two or more vehicles occupy the same location.

    c) Et cetera.

```
value
   114   wf_TRAFFIC: TRAFFIC → N → Bool
   114   wf_TRAFFIC(tra)(n) ≡
   114a     ∀ t,t′:T • {t,t′}⊆dom tra ∧ 0<t′−t<δ_T ⇒
   114a        ∀ v:Vehicle • v ∈ dom(tra(t))∩ dom(tra(t′)) ⇒
   114a           ∀ t″:T • t<t″<t′ • v ∈ dom(tra(t″)) ∧
   114b           ∀ v′:Vehicle • v≠v′ ∧ v′ ∈ dom(tra(t)) ⇒ (tra(t))(v)≠(tra(t))(v′) ∧
   114c     et cetera.
```

W.12 Senders and Receivers

W.12.1 Senders

115.

116.

117.

118.

```
   115
   116
   117
   118
```

W.12.2 Receivers

119.

120.

121.

122.

119
120
121
122

W.13 Miscellaneous

123.

124.

125.

126.

W.14 Model Extensions

W.15 Logistics System Computing Functions

W.16 Conclusion

X A Pipeline System

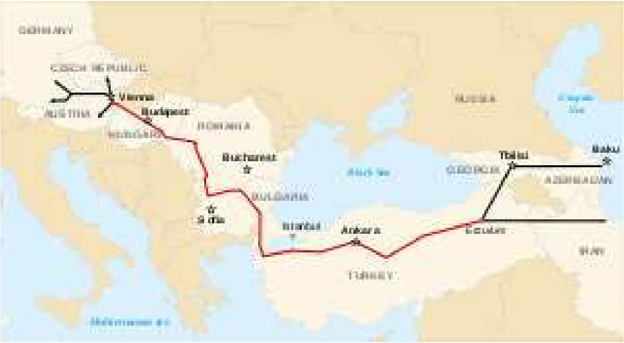


Figure 17: The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

- Named after Verdi's opera
- Gas pipeline
- 3300 kms
- 2011–2014, first gas flow: 2014; 2017–2019, more pipes
- 8 billion Euros
- Max flow: 31 bcmy: billion cubic meters a year
- **http://www.nabucco-pipeline.com/**

Figure 18: The Planned Nabucco Pipeline: http://en.wikipedia.org/wiki/Nabucco_Pipeline

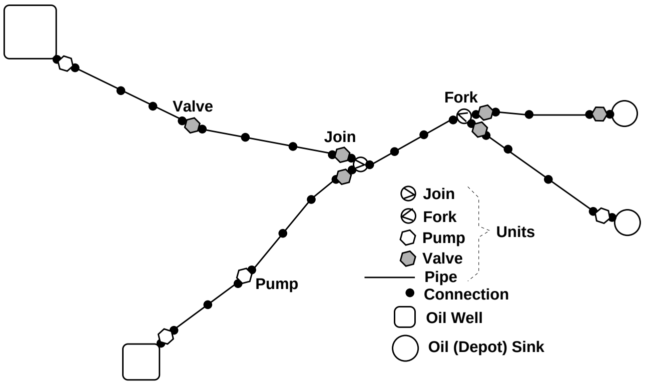


Figure 19: An oil pipeline system

X.1 Non-Temporal Aspects of Pipelines

These are some nn-temporal aspects of pipelines. nets and units: wells, pumps, pipes, valves, joins, forks and sinks; net and unit attributes; amd units states, but not state changes. We omit, in early (i.e., next) chapters, consideration of "pigs" and "pig"-insertion and "pig"-extraction units.

X.1.1 Nets of Pipes, Valves, Pumps, Forks and Joins

127. We focus on nets, $n : N$, of pipes, $\pi : \Pi$, valves, $v : V$, pumps, $p : P$, forks, $f : F$, joins, $j : J$, wells, $w : W$ and sinks, $s : S$.

128. Units, $u : U$, are either pipes, valves, pumps, forks, joins, wells or sinks.

129. Units are explained in terms of disjoint types of PIpes, VAlves, PUmps, FOrks, JOins, WElls and SKs.[70]

```
type
   127  N, PI, VA, PU, FO, JO, WE, SK
   128  U = Π | V | P | F | J | S| W
   128  Π == mkΠ(pi:PI)
   128  V == mkV(va:VA)
   128  P == mkP(pu:PU)
   128  F == mkF(fo:FO)
   128  J == mkJ(jo:JO)
   128  W == mkW(we:WE)
   128  S == mkS(sk:SK)
```

[70] This is a mere specification language technicality.

X.1.2 Unit Identifiers and Unit Type Predicates

130. We associate with each unit a unique identifier, $ui : UI$.

131. From a unit we can observe its unique identifier.

132. From a unit we can observe whether it is a pipe, a valve, a pump, a fork, a join, a well or a sink unit.

```
type
   130  UI
value
   131  obs_UI: U → UI
   132  is_Π: U → Bool, is_V: U → Bool, ..., is_J: U → Bool
      is_Π(u) ≡ case u of mkPI(_) → true, _ → false end
      is_V(u) ≡ case u of mkV(_) → true, _ → false end
       ...
      is_S(u) ≡ case u of mkS(_) → true, _ → false end
```

X.1.3 Unit Connections

A connection is a means of juxtaposing units. A connection may connect two units in which case one can observe the identity of connected units from “the other side”.

133. With a pipe, a valve and a pump we associate exactly one input and one output connection.

134. With a fork we associate a maximum number of output connections, m, larger than one.

135. With a join we associate a maximum number of input connections, m, larger than one.

136. With a well we associate zero input connections and exactly one output connection.

137. With a sink we associate exactly one input connection and zero output connections.

```
value
   133 obs_InCs,obs_OutCs: Π|V|P → {|1:Nat|}
   134 obs_inCs: F → {|1:Nat|}, obs_outCs: F → Nat
   135 obs_inCs: J → Nat, obs_outCs: J → {|1:Nat|}
   136 obs_inCs: W → {|0:Nat|}, obs_outCs: W → {|1:Nat|}
   137 obs_inCs: S → {|1:Nat|}, obs_outCs: S → {|0:Nat|}
axiom
   134 ∀ f:F • obs_outCs(f) ≥ 2
   135 ∀ j:J • obs_inCs(j) ≥ 2
```

If a pipe, valve or pump unit is input-connected [output-connected] to zero (other) units, then it means that the unit input [output] connector has been sealed. If a fork is input-connected to zero (other) units, then it means that the fork input connector has been sealed. If a fork is output-connected to n units less than the maximum fork-connectability, then it means that the unconnected fork outputs have been sealed. Similarly for joins: "the other way around".

X.1.4 Net Observers and Unit Connections

138. From a net one can observe all its units.

139. From a unit one can observe the the pairs of disjoint input and output units to which it is connected:

    a) Wells can be connected to zero or one output unit — a pump.

    b) Sinks can be connected to zero or one input unit — a pump or a valve.

    c) Pipes, valves and pumps can be connected to zero or one input units and to zero or one output units.

    d) Forks, f, can be connected to zero or one input unit and to zero or n, $2 \leq n \leq$obs_Cs(f) output units.

    e) Joins, j, can be connected to zero or n, $2 \leq n \leq$obs_Cs(j) input units and zero or one output units.

```
value
  138  obs_Us: N → U-set
  139  obs_cUIs: U → UI-set × UI-set
         wf_Conns: U → Bool
         wf_Conns(u) ≡
           let (iuis,ouis) = obs_cUIs(u) in iuis ∩ ouis = {} ∧
           case u of
  139a  mkW(_) → card iuis ∈ {0} ∧ card ouis ∈ {0,1},
  139b  mkS(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0},
  139c  mkΠ(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  139c  mkV(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  139c  mkP(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0,1},
  139d  mkF(_) → card iuis ∈ {0,1} ∧ card ouis ∈ {0}∪{2..obs_inCs(j)},
  139e  mkJ(_) → card iuis ∈ {0}∪{2..obs_inCs(j)} ∧ card ouis ∈ {0,1}
           end end
```

X.1.5 Well-formed Nets, Actual Connections

140. The unit identifiers observed by the obs_cUIs observer must be identifiers of units of the net.

```
axiom
    140  ∀ n:N,u:U • u ∈ obs_Us(n) ⇒
    140    let (iuis,ouis) = obs_cUIs(u) in
    140    ∀ ui:UI • ui ∈ iuis ∪ ouis ⇒
    140      ∃ u′:U • u′ ∈ obs_Us(n) ∧ u′≠u ∧ obs_UI(u′)=ui end
```

X.1.6 Well-formed Nets, No Circular Nets

141. By a route we shall understand a sequence of units.

142. Units form routes of the net.

```
type
    141  R = UI^ω
value
    142  routes: N → R-infset
    142  routes(n) ≡
    142    let us = obs_Us(n) in
    142    let rs = {⟨u⟩|u:U•u ∈ us} ∪ {r⌢r′|r,r′:R• {r,r′}⊆rs∧adj(r,r′)} in
    142    rs end end
```

143. A route of length two or more can be decomposed into two routes

144. such that the least unit of the first route “connects” to the first unit of the second route.

```
value
    143    adj: R × R → Bool
    143    adj(fr,lr) ≡
    143      let (lu,fu)=(fr(len fr),hd lr) in
    144      let (lui,fui)=(obs_UI(lu),obs_UI(fu)) in
    144      let ((_,luis),(fuis,_))=(obs_cUIs(lu),obs_cUIs(fu)) in
    144      lui ∈ fuis ∧ fui ∈ luis end end end
```

145. No route must be circular, that is, the net must be acyclic.

```
value
    145  acyclic: N → Bool
    145    let rs = routes(n) in
    145    ∼∃ r:R•r ∈ rs⇒∃ i,j:Nat•{i,j}⊆inds r∧i≠j∧r(i)=r(j) end
```

X.1.7 Well-formed Nets, Special Pairs, wfN_SP

146. We define a “special-pairs” well-formedness function.

    a) Fork outputs are output-connected to valves.

    b) Join inputs are input-connected to valves.

    c) Wells are output-connected to pumps.

    d) Sinks are input-connected to either pumps or valves.

value

```
146    wfN_SP: N → Bool
146    wfN_SP(n) ≡
146      ∀ r:R • r ∈ routes(n) in
146        ∀ i:Nat • {i,i+1}⊆inds r ⇒
146          case r(i) of ∧
146a           mkF(_) → ∀ u:U•adj(⟨r(i)⟩,⟨u⟩) ⇒ is_V(u),_→true end ∧
146          case r(i+1) of
146b           mkJ(_) → ∀ u:U•adj(⟨u⟩,⟨r(i)⟩) ⇒ is_V(u),_→true end ∧
146          case r(1) of
146c           mkW(_) → is_P(r(2)),_→true end ∧
146          case r(len r) of
146d           mkS(_) → is_P(r(len r−1))∨is_V(r(len r−1)),_→true end
```

The **true** clauses may be negated by other **case** distinctions’ is_V or is_V clauses.

X.1.8 Special Routes, I

147. A pump-pump route is a route of length two or more whose first and last units are pumps and whose intermediate units are pipes or forks or joins.

148. A simple pump-pump route is a pump-pump route with no forks and joins.

149. A pump-valve route is a route of length two or more whose first unit is a pump, whose last unit is a valve and whose intermediate units are pipes or forks or joins.

150. A simple pump-valve route is a pump-valve route with no forks and joins.

151. A valve-pump route is a route of length two or more whose first unit is a valve, whose last unit is a pump and whose intermediate units are pipes or forks or joins.

152. A simple valve-pump route is a valve-pump route with no forks and joins.

153. A valve-valve route is a route of length two or more whose first and last units are valves and whose intermediate units are pipes or forks or joins.

154. A simple valve-valve route is a valve-valve route with no forks and joins.

```
value
    147-154 ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr: R → Bool
            pre {ppr,sppr,pvr,spvr,vpr,svpr,vvr,svvr}(n): len n≥2

    147 ppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
    148 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
    149 pvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_P(fu) ∧ is_V(r(len r)) ∧ is_πfjr(ℓ)
    150 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
    151 vpr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_P(lu) ∧ is_πfjr(ℓ)
    152 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)
    153 vvr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ is_V(fu) ∧ is_V(lu) ∧ is_πfjr(ℓ)
    154 sppr(r:⟨fu⟩⌢ℓ⌢⟨lu⟩) ≡ ppr(r) ∧ is_πr(ℓ)

    is_πfjr,is_πr: R → Bool
    is_πfjr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)∨is_F(u)∨is_J(u)
    is_πr(r) ≡ ∀ u:U•u ∈ elems r⇒is_Π(u)
```

X.1.9 Special Routes, II

Given a unit of a route,

155. if they exist (∃),

156. find the nearest pump or valve unit,

157. “upstream” and

158. “downstream” from the given unit.

```
value
    155 ∃UpPoV: U × R → Bool
    155 ∃DoPoV: U × R → Bool
    157 find_UpPoV: U × R ∼→ (P|V), pre find_UpPoV(u,r): ∃UpPoV(u,r)
    158 find_DoPoV: U × R ∼→ (P|V), pre find_DoPoV(u,r): ∃DoPoV(u,r)
    155 ∃UpPoV(u,r) ≡
    155   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧{is_V|is_P}(r(i))∧u=r(j)
    155 ∃DoPoV(u,r) ≡
    155   ∃ i,j Nat•{i,j}⊆inds r∧i≤j∧u=r(i)∧{is_V|is_P}(r(j))
    157 find_UpPoV(u,r) ≡
    157   let i,j:Nat•{i,j}⊆indsr∧i≤j∧{is_V|is_P}(r(i))∧u=r(j) in r(i) end
    158 find_DoPoV(u,r) ≡
    158   let i,j:Nat•{i,j}⊆indsr∧i≤j∧u=r(i)∧{is_V|is_P}(r(j)) in r(j) end
```

X.2 State Attributes of Pipeline Units

By a state attribute of a unit we mean either of the following three kinds: (i) the open/close states of valves and the pumping/not_pumping states of pumps; (ii) the maximum (laminar) oil flow characteristics of all units; and (iii) the current oil flow and current oil leak states of all units.

159. Oil flow, $\phi : \Phi$, is measured in volume per time unit.

160. Pumps are either pumping or not pumping, and if not pumping they are closed.

161. Valves are either open or closed.

162. Any unit permits a maximum input flow of oil while maintaining laminar flow. We shall assume that we need not be concerned with turbulent flows.

163. At any time any unit is sustaining a current input flow of oil (at its input(s)).

164. While sustaining (even a zero) current input flow of oil a unit leaks a current amount of oil (within the unit).

```
type
    159  Φ
    160  PΣ == pumping | not_pumping
    160  VΣ == open | closed
value
           −,+: Φ × Φ → Φ, <,=,>: Φ × Φ → Bool
    160      obs_PΣ: P → PΣ
    161      obs_VΣ: V → VΣ
    162–164  obs_LamiΦ.obs_CurrΦ,obs_LeakΦ: U → Φ
    is_Open: U → Bool
       case u of
          mkΠ(_)→true,mkF(_)→true,mkJ(_)→true,mkW(_)→true,mkS(_)→true,
          mkP(_)→obs_PΣ(u)=pumping,
          mkV(_)→obs_VΣ(u)=open
       end
    acceptable_LeakΦ, excessive_LeakΦ: U → Φ
axiom
    ∀ u:U • excess_LeakΦ(u) > accept_LeakΦ(u)
```

X.2.1 Flow Laws

The sum of the current flows into a unit equals the the sum of the current flows out of a unit minus the (current) leak of that unit. This is the same as the current flows out of a unit equals the current flows into a unit minus the (current) leak of that unit. The above represents an interpretation which justifies the below laws.

165. When, in Item 163, for a unit u, we say that at any time any unit is sustaining a current input flow of oil, and when we model that by obs_CurrΦ(u) then we mean that obs_CurrΦ(u) - obs_LeakΦ(u) represents the flow of oil from its outputs.

value

165 $\text{obs\_in}\Phi\text{: U} \rightarrow \Phi$

165 $\text{obs\_in}\Phi\text{(u)} \equiv \text{obs\_Curr}\Phi\text{(u)}$

165 $\text{obs\_out}\Phi\text{: U} \rightarrow \Phi$

law:

165 $\forall\ \text{u:U} \bullet \text{obs\_out}\Phi\text{(u)} = \text{obs\_Curr}\Phi\text{(u)}-\text{obs\_Leak}\Phi\text{(u)}$

166. Two connected units enjoy the following flow relation:

a) If

i. two pipes, or

ii. a pipe and a valve, or

iii. a valve and a pipe, or

iv. a valve and a valve, or

v. a pipe and a pump, or

vi. a pump and a pipe, or

vii. a pump and a pump, or

viii. a pump and a valve, or

ix. a valve and a pump

are immediately connected

b) then

i. the current flow out of the first unit's connection to the second unit

ii. equals the current flow into the second unit's connection to the first unit

law:

166a $\forall\ \text{u,u}':\text{U} \bullet \{\text{is\_}\Pi\text{,is\_V,is\_P,is\_W}\}(\text{u}'|\text{u}'') \wedge \text{adj}(\langle\text{u}\rangle,\langle\text{u}'\rangle)$

166a $\quad\text{is\_}\Pi\text{(u)}\vee\text{is\_V(u)}\vee\text{is\_P(u)}\vee\text{is\_W(u)} \wedge$

166a $\quad\text{is\_}\Pi(\text{u}')\vee\text{is\_V}(\text{u}')\vee\text{is\_P}(\text{u}')\vee\text{is\_S}(\text{u}')$

166b $\quad\Rightarrow \text{obs\_out}\Phi\text{(u)}=\text{obs\_in}\Phi(\text{u}')$

A similar law can be established for forks and joins. For a fork output-connected to, for example, pipes, valves and pumps, it is the case that for each fork output the out-flow equals the in-flow for that output-connected unit. For a join input-connected to, for example, pipes, valves and pumps, it is the case that for each join input the in-flow equals the out-flow for that input-connected unit. We leave the formalisation as an exercise.

X.2.2 Possibly Desirable Properties

167. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is a valve, v and whose intermediate units are all pipes: if the pump, p is pumping, then we expect the valve, v, to be open.

168. Let r be a route of length two or more, whose first unit is a pump, p, whose last unit is another pump, p' and whose intermediate units are all pipes: if the pump, p is pumping, then we expect pump p'', to also be pumping.

169. Let r be a route of length two or more, whose first unit is a valve, v, whose last unit is a pump, p and whose intermediate units are all pipes: if the valve, v is closed, then we expect pump p, to not be pumping.

170. Let r be a route of length two or more, whose first unit is a valve, v', whose last unit is a valve, v'' and whose intermediate units are all pipes: if the valve, v' is in some state, then we expect valve v'', to also be in the same state.

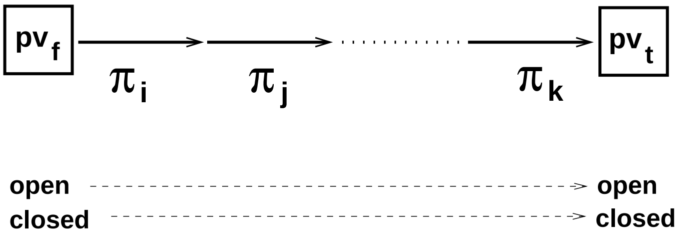


Figure 20: pv: Pump or valve, π: pipe

desirable properties:

167 ∀ r:R • spvr(r) ∧
167 **spvr_prop(r):** obs_PΣ(**hd** r)=pumping ⇒ obs_PΣ(r(**len** r))=open

168 ∀ r:R • sppr(r) ∧
168 **sppr_prop(r):** obs_PΣ(**hd** r)=pumping⇒obs_PΣ(r(**len** r))=pumping

169 ∀ r:R • svpr(r) ∧
169 **svpr_prop(r):** obs_PΣ(**hd** r)=open⇒obs_PΣ(r(**len** r))=pumping

170 ∀ r:R • svvr(r) ∧
170 **svvr_prop(r):** obs_PΣ(**hd** r)=obs_PΣ(r(**len** r))

X.3 Pipeline Actions

X.3.1 Simple Pump and Valve Actions

171. Pumps may be set to pumping or reset to not pumping irrespective of the pump state.

172. Valves may be set to be open or to be closed irrespective of the valve state.

173. In setting or resetting a pump or a valve a desirable property may be lost.

```
value
    171  pump_to_pump, pump_to_not_pump: P → N → N
    172  valve_to_open, valve_to_close: V → N → N
```

```
value
    171  pump_to_pump(p)(n) as n′
    171    pre p ∈ obs_Us(n)
    171    post let p′:P•obs_UI(p)=obs_UI(p′) in
    171           obs_PΣ(p′)=pumping∧else_equal(n,n′)(p,p′) end
    171  pump_to_not_pump(p)(n) as n′
    171    pre p ∈ obs_Us(n)
    171    post let p′:P•obs_UI(p)=obs_UI(p′) in
    171           obs_PΣ(p′)=not_pumping∧else_equal(n,n′)(p,p′) end
    172  valve_to_open(v)(n) as n′
    171    pre v ∈ obs_Us(n)
    172    post let v′:V•obs_UI(v)=obs_UI(v′) in
    171           obs_VΣ(v′)=open∧else_equal(n,n′)(v,v′) end
    172  valve_to_close(v)(n) as n′
    171    pre v ∈ obs_Us(n)
    172    post let v′:V•obs_UI(v)=obs_UI(v′) in
    171           obs_VΣ(v′)=close∧else_equal(n,n′)(v,v′) end
```

```
value
  else_equal: (N×N) → (U×U) → Bool
  else_equal(n,n′)(u,u′) ≡
     obs_UI(u)=obs_UI(u′)
   ∧ u ∈ obs_Us(n)∧u′ ∈ obs_Us(n′)
   ∧ omit_Σ(u)=omit_Σ(u′)
   ∧ obs_Us(n)\{u}=obs_Us(n)\{u′}
   ∧ ∀ u″:U•u″ ∈ obs_Us(n)\{u} ≡ u″ ∈ obs_Us(n′)\{u′}

  omit_Σ: U → U_no_state −−− ″magic″ function
  =: U_no_state × U_no_state → Bool
axiom
  ∀ u,u′:U•omit_Σ(u)=omit_Σ(u′) ≡ obs_UI(u)=obs_UI(u′)
```

X.3.2 Events

Unit Handling Events

174. Let n be any acyclic net.

174. If there exists p, p', v, v', pairs of distinct pumps and distinct valves of the net,

174. and if there exists a route, r, of length two or more of the net such that

175. all units, u, of the route, except its first and last unit, are pipes, then

176. if the route "spans" between p and p' and the *simple desirable property*, sppr(r), does not hold for the route, then we have a possibly undesirable event — that occurred as soon as sppr(r) did not hold;

177. if the route "spans" between p and v and the *simple desirable property*, spvr(r), does not hold for the route, then we have a possibly undesirable event;

178. if the route "spans" between v and p and the *simple desirable property*, svpr(r), does not hold for the route, then we have a possibly undesirable event; and

179. if the route "spans" between v and v' and the *simple desirable property*, svvr(r), does not hold for the route, then we have a possibly undesirable event.

```
events:
  174  ∀ n:N • acyclic(n) ∧
  174    ∃ p,p′:P,v,v′:V • {p,p′,v,v′}⊆obs_Us(n)⇒
  174      ∧ ∃ r:R • routes(n) ∧
  175          ∀ u:U • u ∈ elems(r)\{hd r,r(len r)} ⇒ is_Π(i) ⇒
  176            p=hd r∧p′=r(len r) ⇒ ∼sppr_prop(r) ∧
  177            p=hd r∧v=r(len r) ⇒ ∼spvr_prop(r) ∧
  178            v=hd r∧p=r(len r) ⇒ ∼svpr_prop(r) ∧
  179            v=hd r∧v′=r(len r) ⇒ ∼svvr_prop(r)
```

Foreseeable Accident Events

A number of foreseeable accidents may occur.

180. A unit ceases to function, that is,
    a) a unit is clogged,
    b) a valve does not open or close,
    c) a pump does not pump or stop pumping.
181. A unit gives rise to excessive leakage.
182. A well becomes empty or a sunk becomes full.
183. A unit, or a connected net of units gets on fire.
184. Or a number of other such “accident”.

```
180
181
182
183
184
```

X.3.3 Well-formed Operational Nets

185. A well-formed operational net
186. is a well-formed net
    a) with at least one well, *w*, and at least one sink, *s*,
    b) and such that there is a route in the net between *w* and *s*.

```
value
  185  wf_OpN: N → Bool
  185  wf_OpN(n) ≡
  186    satisfies axiom 140 on page 323 ∧ acyclic(n): Item 145 on page 323  ∧
  186    wfN_SP(n): Item 146 on page 324 ∧
  186    satisfies flow laws, 165 on page 327 and 166 on page 327 ∧
  186a     ∃ w:W,s:S • {w,s}⊆obs_Us(n) ⇒
  186b        ∃ r:R• ⟨w⟩⌢r⌢⟨s⟩ ∈ routes(n)
```

X.3.4 Orderly Action Sequences

Initial Operational Net

187. Let us assume a notion of an initial operational net.

188. Its pump and valve units are in the following states

    a) all pumps are not_pumping, and

    b) all valves are closed.

```
value
    187   initial_OpN: N → Bool
    188   initial_OpN(n) ≡ wf_OpN(n) ∧
    188a      ∀ p:P • p ∈ obs_Us(n) ⇒ obs_PΣ(p)=not_pumping ∧
    188b      ∀ v:V • v ∈ obs_Us(n) ⇒ obs_VΣ(p)=closed
```

Oil Pipeline Preparation and Engagement

189. We now wish to prepare a pipeline from some well, $w : W$, to some sink, $s : S$, for flow.

    a) We assume that the underlying net is operational wrt. w and s, that is, that there is a route, r, from w to s.

    b) Now, an orderly action sequence for engaging route r is to "work backwards", from s to w

    c) setting encountered pumps to pumping and valves to open.

In this way the system is well-formed wrt. the desirable sppr, spvr, svpr and svvr properties. Finally, setting the pump adjacent to the (preceding) well starts the system.

```
value
    189    prepare_and_engage: W × S → N ~→ N
    189    prepare_and_engage(w,s)(n) ≡
    189a       let r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n) in
    189b       action_sequence(⟨w⟩^r^⟨s⟩)(len⟨w⟩^r^⟨s⟩)(n) end
    189      pre ∃ r:R • ⟨w⟩^r^⟨s⟩ ∈ routes(n)

    189c   action_sequence: R → Nat → N → N
    189c   action_sequence(r)(i)(n) ≡
    189c     if i=1 then n else
    189c     case r(i) of
    189c       mkV(_) → action_sequence(r)(i−1)(valve_to_open(r(i))(n)),
    189c      mkP(_) → action_sequence(r)(i−1)(pump_to_pump(r(i))(n)),
    189c      _ → action_sequence(r)(i−1)(n)
    189c    end end
```

X.3.5 Emergency Actions

190. If a unit starts leaking excessive oil

    a) then nearest up-stream valve(s) must be closed,

    b) and any pumps in-between this (these) valves and the leaking unit must be set to not_pumping — following an orderly sequence.

191. If, as a result, for example, of the above remedial actions, any of the desirable properties cease to hold

    a) then — a ha !

    b) Left as an exercise.

X.4 Connectors

The interface , that is, the possible "openings", between adjacent units have not been explored. Likewise the for the possible "openings" of "begin" or "end" units, that is, units not having their input(s), respectively their "output(s)" connected to anything, but left "exposed" to the environment. We now introduce a notion of connectors: abstractly you may think of connectors as concepts, and concretely as "fittings" with bolts and nuts, or "weldings", or "plates" inserted onto "begin" or "end" units.

192. There are connectors and connectors have unique connector identifiers.

193. From a connector one can observe its uniwue connector identifier.

194. From a net one can observe all its connectors

195. and hence one can extract all its connector identifiers.

196. From a connector one can observe a pair of "optional" (distinct) unit identifiers:

    a) An optional unit identifier is

    b) either a unit identifier of some unit of the net

    c) or a ``nil'' "identifier".

197. In an observed pair of "optional" (distinct) unit identifiers

    - there can not be two ``nil'' "identifiers".

- or the possibly two unit identifiers must be distinct

```
type
    192  K, KI
value
    193  obs_KI: K → KI
    194  obs_Ks: N → K-set
    195  xtr_KIS: N → KI-set
    195  xtr_KIs(n) ≡ {obs_KI(k)|k:K•k ∈ obs_Ks(n)}
type
    196  oUIp′ = (UI|{|nil|})×(UI|{|nil|})
    196  oUIp = {|ouip:oUIp′•wf_oUIp(ouip)|}
value
    196  obs_oUIp: K → oUIp
    197  wf_oUIp: oUIp′ → Bool
    197  wf_oUIp(uon,uon′) ≡
    197    uon=nil⇒uon′≠nil∨uon′=nil⇒uon≠nil∨uon≠uon′
```

198. Under the assumption that a fork unit cannot be adjacent to a join unit

199. we impose the constraint thet no two distinct connectors feature the same pair of actual (distinct) unit identifiers.

200. The first proper unit identifier of a pair of "optional" (distinct) unit identifiers must identify a unit of the net.

201. The second proper unit identifier of a pair of "optional" (distinct) unit identifiers must identify a unit of the net.

```
axiom
   198   ∀ n:N,u,u′:U•{u.u′}⊆obs_Us(n)∧adj(u,u′)⇒ ∼(is_F(u)∧is_J(u′))

   199   ∀ k,k′:K•obs_KI(k)≠obs_KI(k′)⇒
             case (obs_oUIp(k),obs_oUIp(k′)) of
                 ((nil,ui),(nil,ui′)) → ui≠ui′,
                 ((nil,ui),(ui′,nil)) → false,
                 ((ui,nil),(nil,ui′)) → false,
                 ((ui,nil),(ui′,nil)) → ui≠ui′,
                 _ → false
             end
```

```
    ∀ n:N,k:K•k ∈ obs_Ks(n) ⇒
        case obs_oUIp(k) of
  200        (ui,nil) → ∃UI(ui)(n)
  201        (nil,ui) → ∃UI(ui)(n)
  200-201  (ui,ui′) → ∃UI(ui)(n)∧∃UI(ui′)(n)
        end
value
    ∃UI: UI → N → Bool
    ∃UI(ui)(n) ≡ ∃ u:U•u ∈ obs_Us(n)∧obs_UI(u)=ui
```

X.5 Temporal Aspects of Pipelines

The else_qual(u,u′)(n,n′) function definition represents a gross simplification. It ignores the actual flow which changes as a result of setting alternate states, and hence the net state. We now wish to capture the dynamics of flow. We shall do so using the Duration Calculus — a continuous time, integral temporal logic that is semantically and proof system "integrated" with RSL:

> Zhou ChaoChen and Michael Reichhardt Hansen
>
> Duration Calculus: A Formal Approach to Real-time Systems
>
> Monographs in Theoretical Computer Science
>
> The EATCS Series
>
> Springer 2004

MORE TO COME

X.6 A CSP Model of Pipelines

We recapitulate Sect. X.4 — now adding connectors to our model:

202. From an oil pipeline system one can observe units and connectors.

203. Units are either well, or pipe, or pump, or valve, or join, or fork or sink units.

204. Units and connectors have unique identifiers.

205. From a connector one can observe the ordered pair of the identity of the two from-, respectively to-units that the connector connects.

```
type
202  OPLS, U, K
204  UI, KI
value
202  obs_Us: OPLS → U-set, obs_Ks: OPLS → K-set
203  is_WeU, is_PiU, is_PuU, is_VaU, is_JoU, is_FoU, is_SiU: U → Bool [mutually exclusive]
204  obs_UI: U → UI, obs_KI: K → KI
205  obs_UIp: K → (UI|{nil}) × (UI|{nil})
```

Above, we think of the types OPLS, U, K, UI and KI as denoting semantic entities. Below, in the next section, we shall consider exactly the same types as denoting syntactic entities !

206. There is given an oil pipeline system, opls.

207. To every unit we associate a CSP behaviour.

208. Units are indexed by their unique unit identifiers.

209. To every connector we associate a CSP channel.

     Channels are indexed by their unique "k"onnector identifiers.

210. Unit behaviours are cyclic and over the state of their (static and dynamic) attributes, represented by u.

211. Channels, in this model, have no state.

212. Unit behaviours communicate with neighbouring units — those with which they are connected.

213. Unit functions, $\mathcal{U}_i$, change the unit state.

214. The pipeline system is now the parallel composition of all the unit behaviours.

Editorial Remark: Our use of the term unit and the RSL literal **Unit** may seem confusing, and we apologise. The former, unit, is the generic name of a well, pipe, or pump, or valve, or join, or fork, or sink. The literal **Unit**, in a function signature, before the → "announces" that the function takes no argument.[71] The literal **Unit**, in a function signature, after the → "announces", as used here, that the function never terminates.

```
value
206  opls:OPLS
channel
209  {ch[ ki ]|k:KI,k:K•k ∈ obs_Ks(opls)∧ki=obs_KI(k)} M
value
```

[71] **Unit** is a type name; () is the only value of type **Unit**.

```
214  pipeline_system: Unit → Unit
214  pipeline_system() ≡
207    ‖ {unit(ui)(u)|u:U•u ∈ obs_Us(opls)∧ui=obs_UI(u)}

208  unit: ui:UI → U →
212       in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
212                      let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  Unit
210  unit(ui)(u) ≡ let u′ = $\mathcal{U}_i$(ui)(u) in unit(ui)(u′) end

213  $\mathcal{U}_i$: ui:UI → U →
213       in,out {ch[ ki ]|k:K,ki:KI•k ∈ obs_Ks(opls)∧ki=obs_KI(k)∧
213                      let (ui′,ui″)=obs_UIp(k) in ui ∈{ui′,ui″}\{nil} end}  U
```

MORE TO COME

X.7 Photos of Pipeline Units and Diagrams of Pipeline Systems

Figure 21: Pipes

Figure 22: Valves

When combining joins and forks we can construct sitches. Figure 24 on the next page shows some actual switches.

[71] See http://en.wikipedia.org/wiki/Nabucco_Pipeline

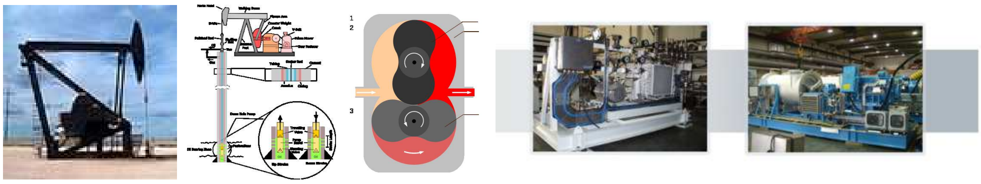


Figure 23: Oil Pumps and Gas Compressors

Figure 24: Oil and Gas Switches

Figure 25: **To be treated in a later version of this report:** Pig Launcher, Receiver and New and Old Pigs

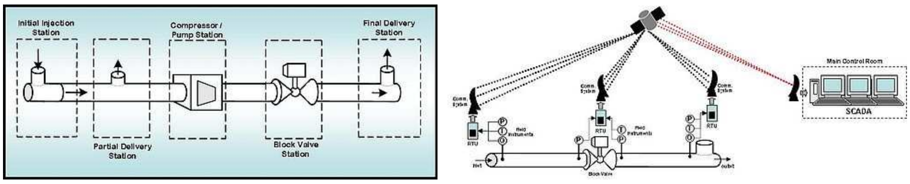


Figure 26: Pipeline Diagrams

Part VI
On Course Projects

Y On Course Projects

Y.1 General

- Course students must complete a course project.
- The enture class chooses one of the 7 projects listed in Sect. Y.2.
- Typically a class of 12 students form 4 (four) project groups.
- Each group chooses a specific sub-domain of the chosen project — e.g. those indicated by `* texts`.

Y.2 Course Topics

1. **Air (Passenger) Transport Industry:**

```
* Airlines
     (time tables, booking, reservations)
* Airports
     (passenger, baggage and fuel handling, catering)
* Air Traffic
     (air space, air lanes [holding zones], various control towers)
```

2. **Container Line Industry:**

```
* Containers
* Container Vessels
* Container Terminal Ports
* Stowage
* Container Lines
```

3. **Financial Service Industry:**

```
* Banking
* Commodities Exchange [General]
     (Brokers, Traders, Stock Exchanges, ...)
* Stock Exchanges [Specific]
     (placing, matching, cancelling and executing orders)
* Credit Cards
* Insurance
* Portfolio Management
* Regulatory Agencies
```

4. **Health Care:**

```
* Hospitals
* Private Physicians / Family Doctors
* Pharmacies
* Insurance
* National Health Service
```

5. **Logistics and Transportation:**

```
* Nets (roads, rails, sea and air)
* Transport vehicles
     (trucks, freight trains, ships, cargo aircraft)
* Bill-of-Ladings, Way Bills
```

6. **“The Market”:**

```
* Consumers
* Retailers
* Wholesalers
* Producers
* Distribution Chain
```

7. **The Gas/Oil Industry:**

```
* Pipelines
* Refineries
* Shipping
* Distribution
```

Y.3 Day-by-Day Course Tutoring

Y.3.1 Tutoring Session Overview: Afternoons 2pm – 4pm

Day	Topic	References	Pages
1	Group Formation	Item 0, cf. below	
2	Ontology	Item 1; Sect. 2	21–27
3	Domain Engineering	Item 2; Sect. 3	28–36
4	Requirements Engineering	Item 3; Sect. 4	37–44
5	Mereology I	Item 4; Sect. C.1	45–47 + 126–133
6	Mereology II	Item 5; Sects. C.2–C.4	134–152
7	Types	Item 6; Sect. A	57–66
8	Values & Operations	Item 7; Sect. A.2	67–81
9	Predicates	Item 8; Sect. A.3	82–83
10	Functions	Item 9; Sects. A.4–A.5	84–98
11	Imperativeness	Item 10; Sect. A.6	99–100
12	Concurrency	Item 11; Sect. A.7	101–108
13	Specifications	Item 12; Sect. A.8	109–113
14	(No session)	—	—
15	Submission of Report 8:45 am	—	—

Y.4 Tutoring Session Contents

0. **Group Formation:**

   a) Full class is expected to participate.

   b) No-shows is interpreted as having dropped out of class.

   c) Lecturer very briefly overview the 7 project proposals: each gets max. 5 minutes coverage by Lecturer.

   d) Lecturer asks course participants to suggest other projects.

   e) Lecturer suggests sub-domains for each of the 7 project proposals.

   f) Class to settle

      i. on one project and
      ii. on group decomposition
      iii. including each group's choice of sub-domain

      before start of next tutoring session.

   g) Lecturer emphasises importance of reading/studying the lecture notes: after all, there is only about 10 pages max. per day of lectures to study.

— and also Project Report Format

For each of the next 12 tutoring sessions the requests, by the lecturer, and as pre-noted below, refer always to the chosen project topic, let us refer to it as $\mathcal{L}_{\mathcal{T}}$.

For each of the n project groups a reasonably well delineated sub-domain is then determined. Group answers to all the 12 items (and their sub-items) are only with respect to the sub-domain assigned to the group.

As you proceed through the days of the project — corresponding to and paced by the items 1–12 below — you may later discover and/or get some ideas about your chosen domain and your chosen, related requirements. Please do not "go back" and insert them in earlier days' work. Insert them where you are wrt. the items.

In each of the below sketched tutoring sessions the tutor (cum lecturer) freely uses the (while|green|black) board to sketch partial answers to each of the sub-enumerated items.

1. **Ontology:** (Sect. 2, Pages 21–27) Informally name and briefly characterise

   a) $\mathcal{L}_{\mathcal{T}}$ **entities** (Report Sect. 1a),

   b) $\mathcal{L}_{\mathcal{T}}$ **functions** (operations/actions) (Report Sect. 1b),

   c) $\mathcal{L}_{\mathcal{T}}$ **events** (Report Sect. 1c) and

   d) $\mathcal{L}_{\mathcal{T}}$ **behaviours** (Report Sect. 1d).

   Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

2. **Domain Engineering:** (Sect. 3, Pages 28–36) Informally name and briefly narrate a few of each of the below

   a) **intrinsic** $\mathcal{L}_{\mathcal{T}}$ domain entities, functions (etc.), events and behaviours (Report Sect. 2a),

   b) $\mathcal{L}_{\mathcal{T}}$ domain **support technologies** (Report Sect. 2b),

   c) $\mathcal{L}_{\mathcal{T}}$ domain **rules & regulations** (Report Sect. 2c),

   d) $\mathcal{L}_{\mathcal{T}}$ domain **scripts** (Report Sect. 2d),

   e) $\mathcal{L}_{\mathcal{T}}$ domain **management & organisation** phenomena and concepts (Report Sect. 2e),

   f) $\mathcal{L}_{\mathcal{T}}$ domain **human behaviour** phenomena and concepts (Report Sect. 2f),

   Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

3. **Requirements Engineering:** (Sect. 4, Pages 37–44) Informally name and briefly narrate a suitable fragment of each of the below items and sub-items:

   a) $\mathcal{L}_{\mathcal{T}}$ Business Process Re-engineering

b) $\mathcal{L}_{\mathcal{T}}$ Domain Requirements:

   i. Projection,
   ii. Instantiation,
   iii. Determination,
   iv. Extension and
   v. Fitting

c) $\mathcal{L}_{\mathcal{T}}$ Interface Requirements

Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

4. **Mereology I:** (Sect. C.1, Pages 45–47 and 126–133)

   Informally name and briefly narrate a suitable fragment of each of the below items based on your answers to Items 1–3 as well as based on later "discoveries" (ideas, etc.) with respect to domain and requirements.

   a) (Simple and) Composite $\mathcal{L}_{\mathcal{T}}$ Entities
   b) (Simple and) Composite $\mathcal{L}_{\mathcal{T}}$ Functions
   c) (Simple and) Composite $\mathcal{L}_{\mathcal{T}}$ Events
   d) (Simple and) Composite $\mathcal{L}_{\mathcal{T}}$ Behaviours

   Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

5. **Mereology II:** (Sects. C.2–C.4, Pages 134–152) Informally name and briefly narrate a suitable fragment of each of the below items based on your answers to Items 1–4 as well as based on later "discoveries" (ideas, etc.) with respect to domain and requirements.

   a) Simple and Composite $\mathcal{L}_{\mathcal{T}}$ Entities
   b) Simple and Composite $\mathcal{L}_{\mathcal{T}}$ Functions
   c) Simple and Composite $\mathcal{L}_{\mathcal{T}}$ Events
   d) Simple and Composite $\mathcal{L}_{\mathcal{T}}$ Behaviours

   Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

6. **Types:** (Sect. A.1, Pages 57–66) We refer to partial answers to Items 1a, 2a–2e, 3b–3b and 4–5 [domains and requirements]. (Electronically) Copy the possibly revised entity-describing text (i.e., entity narratives) into this report section and provide formal types:

a) $\mathcal{L}_{\mathcal{T}}$ sort,

b) $\mathcal{L}_{\mathcal{T}}$ set,

c) $\mathcal{L}_{\mathcal{T}}$ Cartesian,

d) $\mathcal{L}_{\mathcal{T}}$ list,

e) $\mathcal{L}_{\mathcal{T}}$ map and

f) $\mathcal{L}_{\mathcal{T}}$ function types.

Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

7. **Values and Operations:** (Sect. A.2, Pages 67–81) We refer to partial answers to Items 1a, 2a–2e, 3b–3b and 4–6 [domains and requirements]. (Electronically) Copy the possibly revised entity-describing text (i.e., entity narratives) into this report section and provide examples of values and operations.

8. **Predicates:** (Sect. A.3, Pages 82–83) We refer to partial answers to Items 1a, 2a–2e, 3b–3b and 4–7 [domains and requirements].

   a) Focus on narrated axioms for $\mathcal{L}_{\mathcal{T}}$ entities: (electronically) copy the possibly revised entity describing text (i.e., entity narratives) into this report section and provide appropriate formal predicates.

   b) Focus on narrated pre/post conditions of $\mathcal{L}_{\mathcal{T}}$ functions: (electronically) copy the possibly revised function-describing text (i.e., function narratives) into this report section and provide appropriate formal predicates (Sect. A.4.7, Page 88).

   Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

9. **Functions:** (Sects. A.4 – A.5, Pages 84–98) We refer to partial answers to Items 1a, 2a–2e, 3b–3b and 4–8 [domains and requirements].
   Focus on narrated functions descriptions: (electronically) copy the possibly revised $\mathcal{L}_{\mathcal{T}}$ function-describing text into this report section and provide appropriate formal function signatures and function definition bodies – in the following function definition styles.

   a) Use the direct function definition style (Sect. A.4.7, Page 88).

   b) Use the implicit function definition style (Sect. A.4.7, Page 88).

   Document this in an orderly, readable fashion. Make sure that any line items can be referred to in future Report sections (below).

10. **Imperativeness:** (Sect. A.6, Pages 99–100) We refer back to partial answers to Items 1a, 2a–2e, 3b–3b and 4–9 [domains and requirements].
    Focus on already (axiomatically or applicatively) defined functions.

a) Find (or if not already defined) “come up with” an applicatively defined $\mathcal{L}_\mathcal{T}$ function which can be “converted/transformed” into an imperatively defined $\mathcal{L}_\mathcal{T}$ operation based on (one or more) global variables.

b) Similar to Item 10a (just above) but with an operation definition based on one (or more) locally defined variable(s).

11. **Concurrency:** (Sect. A.7, Pages 99–100) We refer to partial answers to Items 1a, 2a–2e, 3b–3b and 4–10 [domains and requirements].

    a) Argue which functions (defined earlier) can or may qualify as $\mathcal{L}_\mathcal{T}$ behaviours ‘behaving’ (occurring) concurrently with other behaviours.

    b) Declare the channels over which such concurrent $\mathcal{L}_\mathcal{T}$ behaviours interact.

    c) Postulate an overall $\mathcal{L}_\mathcal{T}$ ‘system’ behaviour and define this behaviour in terms of a parallel composition of concurrent behaviours.

    d) Redefine these latter $\mathcal{L}_\mathcal{T}$ behaviours wrt. the above signatures and wrt. previously defined functions.

    e) Consider which of the latter $\mathcal{L}_\mathcal{T}$ behaviours may be imperatively defined.

12. **Specifications:** (Sect. A.8, Pages 99–100) We refer back to partial answers to Items 1a, 2a–2e, 3b–3b and 4–11 [domains and requirements]. Now gather, into suitable $\mathcal{L}_\mathcal{T}$ “clusters”, all

    a) **type** definitions,

    b) **variable** declarations,

    c) **channel** declarations,

    d) **value** definitions and

    e) **axiom** definitions.

    You are left to decide on what might be meant by ‘suitable $\mathcal{L}_\mathcal{T}$ clusters’.

Compiled: April 22, 2010: 16:08 ECT
Fredsvej 11, DK-2840 Holte, Denmark; bjorner@gmail.com