Dines Bjørner*

From Domains to Requirements

The Triptych Approach to Software Engineering

August 22, 2009: 11:58

To be submitted, Summer 2009, to Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

*Fredsvej 11, DK-2840 Holte, Danmark, E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

s1

Dedication

My parents:

Else Margrethe Sigrid Bjørner (Christensen) 1905–1993
Ivar Hainau (Christensen) 1907–1971

— for a classical childhood and youth.

Part I

Opening

Preface

acm-abs

This Textbook is Different ! s2

This textbook is different in a number of ways:

1. **The Triptych Dogma** : The dogma "says":
   - Before software can be designed one must understand the requirements.
   - Before requirements can be prescribed one must understand the domain.

   This dogma carries the two main parts of the book:
   - Part IV: Domains and
   - Part V: Requirements.

   No other 'Software Engineering' textbook (other than [17] of the approximately 2400 page [15–17]) propagates this dogma.
2. **Domain Engineering** : s3
   This is a new phase of software development. It is thoroughly treated in Chap. 7. It is explained and motivated in Chaps. 2–3.

   No other 'Software Engineering' textbook (other than [17]) covers 'Domain Engineering' — and the present volume covers that topic in a novel (read: "improved") way.
3. **Derivation of Requirements from Domain Models** : s4
   Requirements development is here presented in a way which differs fundamentally and significantly from how it has been presented by past textbooks on 'Software Requirements Engineering'. This novel and simpler approach, as based on careful domain descriptions, both in narrative and in formal form is thoroughly treated in Chap. 8.

   No other 'Software Engineering' textbook (other than [17]) covers Requirements Engineering in this novel and logical way and the current treatment significantly improves that of [17].
4. **Proper Conceptualisation** (Part III): s5
   Software development is a highly intellectual process. Among the constituent sets of theories, principles and techniques of software development are those of 'Abstraction & Modelling', 'Semiotics' and 'Specification Ontology'. These are treated in separate chapters of this book.

   No other 'Software Engineering' textbook (other than [15–17]) covers these three concepts, 'Abstraction & Modelling', 'Semiotics' and 'Specification Ontology', in this simplified way.

   We shall very briefly explain these three concepts. s6

   4.1 Abstraction & Modelling Chapter 4:
   Abstraction relates to conquering complexity of description through the judicious use of abstraction, where abstraction, briefly, is the act and result of omitting consideration of (what would then be called) details while, instead, focusing on (what would therefore be called) important aspects.

s7

Modelling relates to choosing between (i) property- and model-oriented specification; (ii) a suitable balance between analogic, analytic and iconic modelling; (iii) descriptive and prescriptive modelling as for domain modelling, respectively requirements modelling; and (iv) extensional versus intentional models.

Modelling also has to decide "for which purposes" a model shall serve: to gain understanding, to get inspiration and to inspire, to present, educate and train, to assert and predict and to implement requirements derived from domain models.

4.2 Semiotics Chapter 5: s8

Semiotics deal with the form, i.e., the **syntax**, in which we express concepts; the meaning, i.e., the **semantics**, of what is being expressed; and the reason, i.e., the **pragmatics**, of why we express something and the chosen form of expression.

Since all we really ever do when expressing domains, requirements and software is to produce textual documents it is of utmost importance that we command these three facets of semiotics.

4.3 Specification Ontology [1] Chapter 6: s9

How do we present descriptions ? The technical means of expressing the phenomena and concepts of domains form a meta-ontology. And the description itself is an ontology of the domain. In Chap. 6 we advance three "faces" of ontological nature: (i) the simple entity, operation, event and behaviour approach to description; (ii) the mereology of simple entities; and (iii) the laws of description !

Our treatment of 'Abstraction & Modelling', 'Semiotics' and 'Specification Ontology', above are quite novel and constitute, in our opinion, quite a significant improvement of [17].

5. Examples : s10

The book carries more that 140 substantial both informal and formal examples. Almost half are several pages long.

No other 'Software Engineering' textbook (not even [15–17]) carries so many informal-&formal examples, examples that are substantial — and the present volume ties the many examples more strongly together.

6. Projects : s11

In Appendix B there is a list of annotated course project proposals. A course — based on this book — is proposed to consist of both 'formal' class lectures — covering this book — and 'informal' tutoring sessions —advising students on how to proceed using the book in engineering both a domain description and a requirements prescription for one of the projects listed in Appendix B. That appendix will give some hints, to both lecturers (course project tutors) and
s12 students. Hints to lecturers on how to use this book in the 'formal' class lectures is given in a separate booklet that is (i.e., will be) available on the Internet.

We cannot overemphasise the pedagogical and didactical need to both give the 'formal' class lectures and the course project 'informal' tutoring sessions:

- *"learn by doing"*
- *"but on a science-based foundation"*.

Chapter-by-Chapter Overview

s13
s14 (Chapter 1) We start by providing a background for this study.

[1] Ontology is the philosophical study of the nature of being, existence or reality in general, as well as of the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences [Wikipedia].

(Chapter 2) We introduce the concepts of domains, that is, potential or actual application domains for software. s15

(Chapter 3) We then motivate the study of domains where such studies aim at creating both precise informal and formal descriptions of domains – (and) where formal descriptions are limited to what we can today mathematically formalise. s16

(Chapter 4) Abstraction and modelling are keywords in specifications and we shall therefore very briefly summarise a few key concepts – including property- and model-oriented abstractions. We shall also, likewise very briefly, overview a tool for formal abstraction: the main specification language. `RSL`, of this book and its use in achieving abstractions. s17

(Chapter 5) We take a very brief look at issues of semiotics: pragmatics, semantics and syntax. The prime goal of software engineering work is description, prescription and specification, that is: producing documents, that is, informal and formal texts. Texts have syntax — what we write has meaning (i.e., semantics), and the reason we wrote it down is motivated, i.e., is pragmatics. s18

(Chapter 6) What is it that we are describing (as for domains), prescribing (as for requirements) and specifying (as for software designs)? We shall suggest that the descriptions (etc.) focus on entities and behaviours, functions and events – and shall therefore briefly summarise these concepts (and likewise briefly exemplify their abstract modelling) before deploying this "specification ontology" in domain and in requirements engineering. s19

(Chapter 7) Domain engineering is then outlined in terms of its many stages: [i] information document creation, [ii] identification of domain stake-holders, [iii] business process rough sketching, [iv] domain acquisition, [v] domain analysis and concept formation, [vi] domain terminologisation, [vii] domain modelling – the major stage – [viii] domain model verification (checking, testing), [ix] domain description validation, and [x] domain theory creation. Emphasis is put on business process description (Sect. 7.2) and on the s20
six sub-stages of domain modelling: (a) intrinsics, (b) support technologies, (c) management and organisation, (d) rules and regulations, (e) scripts and (f) human behaviour (Sects. 7.3–7.8). A final section, Sect. 7.9 summarises the opening and closing stages of domain engineering: stakeholder identification and liaison, acquisition, business processes, terminoligisation, respectively verification, model checking, testing, validation and domain theory issues. s21

(Chapter 8) It is finally outlined, in some detail, how major parts of requirements can be systematically "derived" from domain descriptions: in three major sub-stages: [A] domain requirements, [B] interface requirements and [C] machine requirements – where our contribution is sôlely placed in substages [A–B]. In this part it is briefly argued why current requirements engineering appears to be based s22
on a flawed foundation. Emphasis is put on the pivotal steps of domain requirements in which (a) business processes are re-engineering (Sect. 8.4); (b) domain requirements are projected, instantiated, made more deterministic, extended and fitted (Sect. 8.6); (c) interface requirements are "created" while considering the simple entities, functions, events and behaviours shared that are (to be) shared between the domain and the machine (Sect. 8.8); and (d) machine requirements are laboriously enumerated and instantiated (Sect. 8.9). A final section, Sect. 8.10 summarises the opening and closing stages of requirements engineer- s23
ing: stakeholder identification and liaison, acquisition, business process re-engineering, terminoligisation, respectively verification, model checking, testing, validation, satisfiability & feasibility and requirements theory issues.

A Lecture Schedule Proposal

Each of the lectures, Lectures 1–18, listed below, is thought of as a double lecture session of (two) 50 minute lectures separated by a 10 minute break.

Lectures marked ($\ominus$) can be omitted entirely.

Lectures marked ($\frac{1}{2}$) can be reduced to one lecture.

In this way the listed 18 lectures can be "cut" to 14.

Lectures 1–7: Introductory Concepts

1. Summary	Slides XII–XIII
1. Background	Slides 3–4
1. What are Domains ?	Slides 5–18
2. Motivation for Domain Engineering	Slides 19–20

$\frac{1}{2}$ 3.–4.	Abstraction & Modelling	Slides 23–35
5.	Semiotics	Slides 37–53
$\frac{1}{2}$ 6.–7.	A Specification Ontology	Slides 55–73

Lectures 8–12: Domain Engineering

8.	Opening Stages and Intrinsics	Slides 77–88
9.	Supp.Techns., Mgt. & Org. and Rules & Regs.	Slides 88–98
$\frac{1}{2}$ 10.–11.	Scripts	Slides 98–123
12.	Human Behaviour and Closing Stages	Slides 123–128
12.	Opening and Closing DE Stages	Slides 128–129

Lectures 13–16: Requirements Engineering

13.	Acquisition and Business Processes	Slides 133–141
14.	Domain Requirements	Slides 141–154
15.	Interface Requirements	Slides 154–166
16.	Machine Requirements	Slides 166–175
16.	Opening and Closing RE Stages	Slides 175–175

Lectures 17–18: Closing Concepts

$\ominus$ 17.	Domain Demos	
$\ominus$ 17.	Domain-based Innovation	
18.	Conclusion and Acknowledgements	Slides 179–181

Appendix A (Pages 193–194). provides a lecturers' guide to using this textbook.

Course Project Tutoring

In addition to the lectures it is strongly suggested that each day, for example, a morning double lecture is given, there is also an afternoon two hour project tutoring session. More about this in Appendix. B (Pages 195–200).

Contents

Part III Proper Conceptualisation

Part II

Introduction

1 Background

s24 acm-intro

This book is written on the background of three more-or-less independent lines of (a) more than 40 years of speculations, by our community, about, proposals for, and, obviously, practice of software engineering [111]; (b) about 40 years of progress in program verification [76], and (c) of almost 50 years of formal specification of first programming language semantics [9, 100, 104, 109, 132], then software designs, then requirements, and now, finally domains. s25

This book is also written on the background of many efforts that seek to merge these lines — as witnessed in strand (c) above: (d) notably there is the effort to express abstractions and their refinement [77], for example, (e) such as these abstractions and refinements, with respect abstract data structures and abstract operations, were (first) facilitated in `VDM` [24,25,53,54], and, (g) with respect to process abstractions, such as they were facilitated in `CSP` [78, 79, 127, 131]. s26

Over 30+ years of determined efforts in the areas of formal specification languages [23] and refinement [25,89–91]; and of their deployment in many industrial projects, this line of research and experimental development has been manifested in at least three notable forms: (i) the systematic-to-rigorous development of an `Ada` compiler using `VDM` [26]; (ii) the commercialisation of an industry-strength tool set for the `VDM` `Specification Language`, `VDM-SL`, by the Japanese software house CSK[1]; and (iii) the publication of [15–17]. s27

All this research and development, (1) 35+ years of doing advanced type experimental, explorative and actually overseeing real, industry-strength commercial software developments, (2) 30+ years of teaching the underlying approaches, semantics, formal specification, and refinement in a software engineering setting, and (3) putting students on the road to found and direct some eight software houses (now with some 600 former students) — based on student MSc and PhD projects — and survive in the business, makes me conclude that the basic elements to be included in a proper software engineering education and to be regularly practised by the graduating software engineers include the following concepts: (a) a firm grasp *on the simple use* of discrete mathemat- s28
ics: sets, Cartesians, sequences, maps, functions (including the λ-Calculus), and simple universal algebras; (b) a firm grasp *on the simple use* of mathematical logic; (c) a firm grasp *on the simple use* of abstract and concrete types and their values, sub-types and derived types (such as found in several formal specification languages); (d) a firm grasp *on the simple use* of the semiotics concepts of syntax, semantics and pragmatics, including the formalisation of syntax and semantics — in various forms: "classical" operational semantics, denotational semantics, structural operational semantics, etc. [38, 64, 125, 130, 138, 141]; (e) a firm grasp *on the simple use* of property- as well s29
as model-oriented abstractions as facilitated by such formal specification languages as `Alloy` [87], `B` and `Event B` [1,31], `RAISE` [15–17,58,59,61], `VDM` [24,25,53,54] or `Z` [74,75,134,135,142]; (f) a firm grasp *on the simple use* of the diagrammatic specification approaches provided by `finite state automata` and `finite state machines` (any reasonable textbook on formal languages and automata theory should do), `MSC` (message sequence charts) [83–85], `Petri nets` [88,112,120–122] and `Statecharts` [68–71,73]; (g) a firm grasp *on the use* some temporal logic approach to specify s30

[1] http://www.csk.com/support_e/vdm/index.html

time dependent behaviours, `DC` (duration calculus) [145,146], `ITL` (interval temporal logic) [51,110], the Pnueli/Manna approach [97–99], or `TLA+` [93, 94, 102, 103]; and (h) a firm grasp *on the simple use* of `CSP` (communicating sequential processes) [78, 79, 127, 131].

This book will overview some, we think crucial, aspects of software engineering on this background. (We shall not cover Items (f–g).)

2

What are Domains?

s31 acm-wad

2.1 Delineation

Definition 1 – **Domain:** *By a domain, or, more precisely an application domain, we shall understand (i) a suitably delineated area of human activity, that is, (ii) a universe of discourse, something for which we have what we will call a domain-specific terminology, (iii) such that this domain has reasonably clear interfaces to other such domains.* ■

s32

Definition 2 – **Domain Description:** *By a domain description we shall understand (i) a set of pairs of informal, for ex., English language, and formal, say mathematical, texts, (ii) which are commensurate, that is, the English text "reads" the formulas, and (iii) which describe the simple entities, operations, events and behaviours of a domain in a reasonably comprehensive manner.* ■

We use the simpler term 'domain' in lieu of the longer term 'application domain'. The prefix (of the latter) shall indicate that we are eventually referring to computer application.

2.1.1 Elements, Aims and Objectives of Domain Science(I) s33

What will emerge from this book are the contours of 'domain science': the study and knowledge of domains. We shall here start the sketching of these contours.

In order to better understand what domain engineering is about we contrast it to physics. But first we must make a distinction between the terms 'phenomenon' (phenomena) and 'concept' (concepts).

s34

Definition 3 – **Phenomenon:** *By a phenomenon we understand an observable fact, that is, a temporal or spatio/temporal individual (particular, "thing") of sensory experience as distinguished from a noumenon*[1]*, that is a fact of scientific interest susceptible to scientific description and explanation.* ■

s35

Definition 4 – **Concept:** *By a concept we understand something conceived in the mind, a thought, an abstract or generic idea generalized from particular instances.* ■

[1] Noumenon: a posited object or event as it appears in itself independent of perception by the senses.

2.1.2 Physics versus Domain Science s36

General

Physicists study 'mother nature':

> *"Physics (Greek: physis φυσις meaning 'nature'), a natural science, is the study of matter and its motion through space-time and all that derives from these, such as energy and force. More broadly, it is the general analysis of nature, conducted in order to understand how the world and universe behave."* [Wikipedia]

Domain scientists and engineers study 'domains':

> *"Domains are here seen as predominantly man-made universes, that is, as areas of human activity, where the emphasis is on the structures (entities) conceived and built by humans (the domain owners, managers, designers, domain enterprise workers, etc.), and the operations that are initially requested, or triggered, by humans (the domain users)."*

s37 In physics (as characterised above) the physicists, in principle, **do not include** human actions and behaviour in their study.[2]

In domain science and engineering the scientists and engineers, in principle, **do include** human
s38 actions and behaviours in their study.

In physics physicists model **usually continuous** state values of the chosen sub-universe, that is, the dynamics of observable or postulated state component values, and their principle tools are
s39 those of differential equations, integral calculi, statistics, etc. Space and time plays a core rôle.

In domain science and engineering the scientists and engineers model (i) algebraic[3] structures of the chosen sub-universe (in addition to their **usually discrete** "state" values and operations), (ii) how simple entities are composed, (not only just their atomic but also composite values), (iii) how these structures may expand or retract, that is, operations on structures, not just on values.
s40 Space and time normally plays only a secondary rôle.

The tools of domain scientists and engineers are those of careful, precise informal (i.e., narrative) natural language and likewise careful abstractions expressed in some formal specification languages emphasising the algebraic nature of entities and their operations. That is, tools that
s41 originate with computer and computing scientists.

Why not use the same tools as physicists do? Well, they are simply not suited for the problems at hand. Firstly the states of physics typically vary continuously, whereas those of domains typically vary in discrete steps. Secondly the number of state variables of physics do usually not vary, whereas those of domain do — whole structures "collapse" or "expand" (sometimes "wholesale", sometimes "en detail"). Thirdly the models of physics, by comparison to those of domains, contain "only a few types" of oftentimes thousands of state variables — almost all modelled as reals, or vectors, matrices, tensors, etc., of reals, whereas those of domains contain very many, quite different types
s42 — sometimes atomic, sometimes composite, but rarely modelled in matrix form.

Models of physics, as already mentioned, express continuous phenomena. Models of domains, as also already mentioned, express logic properties of discrete, algebraic structures.

For those and several other reasons the tools of physicists are quite different from the tools of
s43 domain scientists and engineers.

In theoretical physics there is no real concern for computability. Mathematical models themselves provide the answers. For domain engineering there is a real concern for computability. The mathematical models often serve as a basis for requirements for software, that is, for computing. Hence it was natural that the tools of domain science and engineering originated with the formal specification languages that were and are used for specifying software.

---

[2] The claimed possibility that humans are the origin, through their use of fossil energy sources, of the depletion of the ozon layer, does not mean that the physicists, in their model include human actions and behaviour: if physicists do consider humans as the "culprits", then that still does not enter into their models !

[3] Recall that an algebra is a usually finite set of possibly infinite classes (i.e., types) of usually discrete entities and a usually finite set of operations whose signature ranges of the entity types.

Spatial Attributes of Phenomena and Concepts s44

Some phenomena (p:P) (types over and values of such types) enjoy the meta-linguistic $\mathcal{L}$ property, $\mathcal{L}$ for $\mathcal{L}$ocation. Let any phenomenon be subject to the meta-linguistic predicate, has_$\mathcal{L}$, and function, obs_$\mathcal{L}$:

type
  P,C
value
  has_$\mathcal{L}$: P $\rightarrow$ **Bool**
  obs_$\mathcal{L}$: P $\xrightarrow{\sim}$ $\mathcal{L}$, **pre** obs_$\mathcal{L}$(e): has_$\mathcal{L}$(e)
axiom
  $\forall$ p,p$'$:P •
    has_$\mathcal{L}$(p)$\wedge$has_$\mathcal{L}$(p$'$) $\wedge$ p$\neq$p$'$ $\Rightarrow$ obs_$\mathcal{L}$(p) $\neq$ obs_$\mathcal{L}$(p$'$)

We consider $\mathcal{L}$ (for $\mathcal{L}$ocations) to be an attribute of those phenomena which satisfy the has_$\mathcal{L}$ property. s45

Simple Entities versus Attributes s46

We make a distinction between simple entities and attributes. Simple entities are phenomena or concepts that may be separable parts of other simple entities; that may (thus) be composed into other (composite) simple entities; and that otherwise possess one or more attributes. Attributes s47
are properties of simple entity phenomena that together form atomic simple entities, or characterise composite entities apart, i.e., in isolation from their sub-entities; that cannot be "removed" from a simple entity otherwise possessing such attributes; and that may be modelled as values of simple or composite types.

2.1.3 Constituent Sciences of Domain Science s48

Knowledge Engineering

> *"Knowledge engineering is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise."* [Wikipedia]

Knowledge (science and) engineering [52], what humans know and believe, promise and commit, what is necessary, probable and/or possible, is a proper part of domain science — but we omit any treatment of this fascinating topic.

Computer Science s49

Definition 5 – **Computer Science:** *Computer science is the study and knowledge about the "things" that may exist inside computers.* ■

Computing Science

Definition 6 – **Computing Science:** *is the study and knowledge how to construct the "things" that may exist inside computers.* ■

2.1.4 Elements, Aims and Objectives of Domain Science (II) s50

So computing science and knowledge (science and) engineering are both part of domain science. Computer science, notably with its emphasis on algebraic structures and mathematical (modal) logics provide some of the foundations for the studies of computing science and knowledge (science and) engineering

2.2 Informal Examples s51

We will give several informal examples. For each of these examples we shall very briefly mention
s52 observable simple entity, operation, event and behaviour phenomena as well as concepts.

Example 1 – **Air Traffic (I):** The domain-specific terminology includes such entities as: aircraft, ground, terminal, area and continental control towers and centers, air-lanes, etc. The modelled atomic and composite structures and operations include airspace as consisting of air-lanes, airports and various control towers; traffic modelled as function from time to aircraft positions in airspace; operations of aircraft take-off, guidance and landing; events of communication between pilots and control towers; et cetera, [10] ∎

s53 See Example 50: Air Traffic Business Processes starting Page 81.

Example 2 – **Banking:** The domain-specific terminology includes such terms as: *clients*; *banks* with *demand/deposit* accounts with *yield* and *interest* rates and *credit limits* and with *open, deposit, withdraw, transfer, statement* and *close* operations; and with *mortgage* accounts and *loan approval*, *payment installation*, *loan defaulting*, etc.; and *bankruptcy*, *payment due*, *credit limit exceeded*, etc. events; et cetera; et cetera. ∎

s54

See Examples 66–69.

Example 3 – **Container Line Industry:** The domain-specific terminology includes such terms as: container, container line, container vessel (bay, row, stack, etc.), container terminal port (quay, crane, stack/stacking, etc.), sea lane, etc, container stowage, et cetera, [18]. ∎

s55

Example 4 – **Health Care:** The domain-specific terminology includes such terms as: citizen cum patient, medical staff, hospital, ward, bed, operating theatre, patient medical journal, anamnese[4], analysis, diagnostics, treatment, etc., hospitalisation plan, et cetera, [42]. ∎

See Example 63: A Health Care License Language starting Page 103.

s56

Example 5 – **"The Market":** The domain-specific terminology includes such terms as: consumer, retailer, wholesaler and producer; merchandise, order, price, quantity, in-store, back-order, etc.; supply chain; inquire, order, inspect delivered goods, accept goods, pay; failure of delivery, default on payments, etc.; et cetera. [12]. ∎

s57

Example 6 – **Oil Industry:** The domain-specific terminology includes such terms as: oil field, pump and platform; oil pipeline, pipe, flow pump, valve, etc.; oil refinery; oil tanker, harbour, etc. [19].

MORE TO COME

∎

s58 See Example 55: An Oil Pipeline System starting Page 84.

Example 7 – **Public Government:** The domain-specific terminology includes such terms as: citizens, lawmakers, administrators, judges, etc., law-making, law-enforcing (central and local government administration) and law-judging ("the judiciary"), documents: law drafts, laws, public administration templates, forms and letters, verdicts, etc., document creation, editing, reading, copying, distribution and shredding, etc. [43].

MORE TO COME

∎

s59 See Example 64: A Public Administration License Language starting Page 105.

---

[4] Anamnese: the patients' history of illness, including the most present.

Example 8 – **Railways:** The domain-specific terminology includes such terms as: railway net with track units such as linear, simple switches, simple crossover, crossover switches, signals, etc. [13]; trains; passengers, tickets, reservations; timetable and train traffic; train schedules [116], train rostering [137], train maintenance plan [117], etc.

MORE TO COME

■

See Example 56: Railway Switch Support Technology starting Page 88. s60

Example 9 – **Road System:** The domain-specific terminology includes such terms as: hubs (intersections) and links (road segments), open and close hub and link traversal directions, hub semaphores, etc. [20].

MORE TO COME

■

See Example 10: Transport Net (I) starting Page 9 and Example 34: Transport Net (II) starting Page 55.

2.3 An Initial Domain Description Example s61

Before we delve into pragmatic and methodological issues of domain engineering we need an example which show the both informal and formal form in which we express a domain description.

The example is that of describing a transport net. s62

acm-transportnet

Example 10 – **Transport Net (I):** acm-transportnet

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.

type
  1 H, L,
  2 N = H-**set** × L-**set**
axiom
  2 ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ks≥1

RSL Explanation

- 1: The type clause **type** H, L, defines two abstract types, also called sorts, H and L, of what is meant to abstractly model "real" hubs and nets. H and L are hereby introduced as type (i.e., sort) names. C.1.2 Pg.204
  (The fact that the type clause (1) is "spread" over two lines is immaterial.)
- 2: the type clause **type** N = H-**set** × L-**set** defines a concrete type N (of what is meant to abstractly model "real" nets).
  - ⋆ The equal sign, , defines the meaning of the left-hand side type name, N, to be that of the meaning of
  - ⋆ H-**set**×L-**set**, namely Cartesian groupings of, in this case, pairs of sets of hubs (H-**set**) and sets of links (L-**set**), that is, C.4.3
  - ⋆ × is a type operator which, when infix applied to two (or more) type expressions yields the type of all groupings of values from respective types, and Pg.202[9]
  - ⋆ -**set** is a type operator which, when suffix applied, to, for example H, i.e., H-**set**, constructs, the type power-set of H, that is, the type of all finite subsets of type H. Pg.202[7] C.4.2
  - ⋆ Similarly for L-**set**.

(The fact that type clause (2), as it appears in the formalisation, is not preceded immediately by the literal **type**, is (still) immaterial: it is part of the type clause starting with **type** and ending with the clause 2.)

- 2: The axiom **axiom** ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ks≥1 C.2,C.4.6
- Thus we see that a type clause starts with the keyword (or literal) **type** and ends just before another such specification keyword, here **axiom**. That is, a type clause syntactically consists of the keyword **type** followed by one or more sort and concrete type definitions (there were three above). Pg.208[30]
- C.10 And we see that a fragment of a formal specification consists of either type clauses, or axioms, or of both, or, as we shall see later, "much more" !

End of RSL Explanation

s63

3. There are hub and link identifiers.
4. Each hub (and each link) has an own, unique hub (respectively link) identifiers (which can be observed from the hub [respectively link]).

type
  3 HI, LI
value
  4a obs_HI: H → HI, obs_LI: L → LI
axiom
  4b ∀ h,h′:H, l,l′:L • h≠h′ ⇒
    obs_HI(h)≠obs_HI(h′) ∧ l≠l′⇒obs_LI(l)≠obs_LI(l′)

RSL Explanation

- 3: introduces two new sorts;
- C.6.5 4a: introduces two new observer functions:
  - Pg.202[13] ⋆ → is here an infix type operators.
  - ⋆ Infixing L and LI it constructs the type of functions (i.e., function values) which apply to values of type L and yield values of type LI.

  and
- 4b: expresses the uniqueness of identifiers.

End of RSL Explanation

s64

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:

5. From any link of a net one can observe the two hubs to which the link is connected.
   a) 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 one or more links to which are connected to the hub.
   a) Again: by observing their distinct link identifiers.
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

We used, above, the concept of 'identifiers of hubs' and 'identifiers of links' of nets. We define, below,
s65 functions (iohs, iols) which calculate these sets.

value
  5a obs_HIs: L → HI-**set**,
  6a obs_LIs: H → LI-**set**,
axiom
  5b ∀ l:L • **card** obs_HIs(l)=2 ∧
  6b ∀ h:H • **card** obs_LIs(h)≥1 ∧

$\forall$ (hs,ls):N •
5a) $\forall$ h:H • h $\in$ hs $\Rightarrow$ $\forall$ li:LI • li $\in$ obs_LIs(h) $\Rightarrow$
$\exists$ l′:L • l′ $\in$ ls $\wedge$ li=obs_LI(l′) $\wedge$ obs_HI(h) $\in$ obs_HIs(l′) $\wedge$
6a) $\forall$ l:L • l $\in$ ls $\Rightarrow$
$\exists$ h′,h″:H • {h′,h″}$\subseteq$hs $\wedge$ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}
7 $\forall$ h:H • h $\in$ hs $\Rightarrow$ obs_LIs(h) $\subseteq$ iols(ls)
8 $\forall$ l:L • l $\in$ ls $\Rightarrow$ obs_HIs(h) $\subseteq$ iohs(hs)

value
iohs: H-**set** $\rightarrow$ HI-**set**, iols: L-**set** $\rightarrow$ LI-**set**
iohs(hs) $\equiv$ {obs_HI(h)|h:H•h $\in$ hs}
iols(ls) $\equiv$ {obs_LI(l)|l:L•l $\in$ ls}

RSL Explanation

- 5a,6a: Two observer functions are introduced.
- 5b,6b: Universal quantification secure that all hubs and links have prerequisite number of unique (reference) identifiers. C.3
  - ⋆ 5a): We read $\forall$ h:H • h $\in$ hs $\Rightarrow$ $\forall$ li:LI • li $\in$ obs_LIs(h) $\Rightarrow$ $\exists$ l′:L • l′ $\in$ ls $\wedge$ li=obs_LI(l′) $\wedge$ obs_HI(h) $\in$ obs_HIs(l′): For all hubs (h) of the net ($\forall$h:H•h$\in$hs) it is the case ($\Rightarrow$) that for all link identifiers (li) of that hub ($\forall$li:LI•li$\in$obs_LIs(h)) it is the case that there exists a link of the net ($\exists$l′:L•l′$\in$ls) where that link's (l′'s) identifier is li and the identifier of h is observed in the link l′.
  - ⋆ 6a): We read $\forall$ l:L • l $\in$ ls $\Rightarrow$ $\exists$ h′,h″:H • {h′,h″} $\subseteq$hs $\wedge$ obs_HIs(l) = {obs_HI(h′), obs_HI(h″)}: for all ... further reading is left as exercise to the reader.
- 7: Reading is left as exercise to the reader.
- 8: Reading is left as exercise to the reader.
- iohs,iols: These two lines define the signature: name and type of two functions. C.6.5
- iohs(hs) calculates the set ({...}) of all hub identifiers (obs_HI(h)) for which h is a member of the set, hs, of net hubs. C.4.2 Pg.205
- iols(ls) calculates in the same manner as does iohs(hs).
  We can read the set comprehension expression to the left of the definition symbol $\equiv$: "the set of all obs_LI(l) for which (|) l is of type L and such that (•) l is in ls". C.4.2

End of RSL Explanation Pg.205

s66

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', or, as we shall later discuss it, 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.

To illustrate the concept of operations[5] on transport nets we postulate those which "build" and "maintain" the transport nets, that is those road net or rail net (or other) development constructions which add or remove links. (We do not here consider operations which "just" add or remove hubs.) By an operation designator we shall understand the syntactic clause whose meaning (i.e., semantics) is that of an action being performed on a state. The state is here the net. We can also think of an operation designators as a "command".

Initialising a net must then be that of inserting a link with two new hubs into an "empty" net. Well, the notion of an empty net has not been defined. The axioms, which so far determine nets and which has been given above, appears to define a "minimal" net as just that: two linked hubs ! s67

First we treat the syntax of operation designators ("commands").

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

[5] We use the terms functions and operations synonymously.

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.

10. From a net one can remove a link. The removal command specifies a link identifier.

s68

type
9 Insert == Ins(s_ins:Ins)
9 Ins = 2xHubs | 1x1nH | 2nHs
9a) 2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
9b) 1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
9c) 2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)
axiom
9d) ∀ 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″)}
type
10 Remove == Rmv(s_li:LI)

RSL Explanation

Pg.202[16]
Pg.203

- 9: 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.
  - ⋆ 9a): 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.
  - ⋆ 9b): Reading of this type clause is left as exercise to the reader.
  - ⋆ 9c): Reading of this type clause is left as exercise to the reader.
  - ⋆ 9d): 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).
- 10; The type clause **type** Remove == Rmv(s_li:LI)
  - ⋆ (as for Items 9b) and 9c))
  - ⋆ 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

s69

Then we consider the meaning of the Insert operation designators.

11. 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.
12. 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.
13. If the command is of the form 2oldH(hi′,l,hi′) then
    $\star 1$ hi′ must be the identifier of a hub in hs,
    $\star s2$l must not be in ls and its identifier must (also) not be observable in ls, and
    $\star 3$ hi″ must be the identifier of a(nother) hub in hs.

s70

14. If the command is of the form 1oldH1newH(hi,l,h) then
    $\star 1$ hi must be the identifier of a hub in hs,
    $\star 2$ l must not be in ls and its identifier must (also) not be observable in ls, and
    $\star 3$ h must not be in hs and its identifier must (also) not be observable in hs.
15. If the command is of the form 2newH(h′,l,h″) then
    $\star 1$ h′ — left to the reader as an exercise (see formalisation !),
    $\star 2$ l — left to the reader as an exercise (see formalisation !), and
    $\star 3$ h″ — left to the reader as an exercise (see formalisation !).

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

s71

value
  11 int_Insert: Insert $\rightarrow$ N $\xrightarrow{\sim}$ N
  12′ pre_int_Insert: Ins $\rightarrow$ N $\rightarrow$ **Bool**
  12″ pre_int_Insert(Ins(op))(hs,ls) $\equiv$
$\star 2$ s_l(op)$\notin$ ls $\wedge$ obs_LI(s_l(op)) $\notin$ iols(ls) $\wedge$
    case op **of**
  13)   2oldH(hi′,l,hi″) $\rightarrow$ {hi′,hi″}$\in$ iohs(hs),
  14)   1oldH1newH(hi,l,h) $\rightarrow$
           hi $\in$ iohs(hs) $\wedge$ h$\notin$ hs $\wedge$ obs_HI(h)$\notin$ iohs(hs),
  15)   2newH(h′,l,h″) $\rightarrow$
           {h′,h″}$\cap$ hs={} $\wedge$ {obs_HI(h′),obs_HI(h″)}$\cap$ iohs(hs)={}
    end

RSL Explanation

- 11: The value clause **value** int_Insert: Insert $\rightarrow$ N $\xrightarrow{\sim}$ N names a value, int_Insert, and defines its type to be Insert $\rightarrow$ 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.)
- 12′: The predicate pre_int_Insert: Insert $\rightarrow$ N $\rightarrow$ **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.
- 12″: 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)$\notin$ls$\wedge$obs_LI(s_l(op))$\notin$iols(ls), namely that the new link is not an existing net link and that its identifier is not already known.
  - $\star$ 13): If the Insert command is of kind 2oldH(hi',l,hi'') then {hi$'$,hi$''$}$\in$ 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.
  - $\star$ 14): If the Insert command is of kind 1oldH1newH(hi,l,h) then ... exercise left as an exercises to the reader.
  - $\star$ 15): 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. C.4.6 on page 207 Item 23 on page 208.

End of RSL Explanation

s72

16. 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.
17. 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
16: xtr_H: HI $\rightarrow$ N $\xrightarrow{\sim}$ H
16: xtr_H(hi)(hs,_) $\equiv$ **let** h:H•h $\in$ hs $\wedge$ obs_HI(h)=hi **in** h **end**
  pre hi $\in$ iohs(hs)
17: xtr_L: HI $\rightarrow$ N $\xrightarrow{\sim}$ H
17: xtr_L(li)(_,ls) $\equiv$ **let** l:L•l $\in$ ls $\wedge$ obs_LI(l)=li **in** l **end**
  pre li $\in$ iols(ls)

RSL Explanation

Pg.215

- 16: 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.
- 16: The wild-card, _, expresses that the extraction (xtr_H) function does not need the L-**set** argument.
- 17: Left as an exercise for the reader.

End of RSL Explanation

s73

18. 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.
19. 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 $\rightarrow$ H, rLI: H $\times$ LI $\xrightarrow{\sim}$ H
18: 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$'$)
19: 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

- 18: The add link identifier function aLI:
  - $\star$ 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.
  - $\star$ 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} ∧ 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. Pg.208[25]
- 19: The remove link identifier function rLI:
  - ⋆ The function definition clause rLI(h′,li) **as** h defines the application of rLI to a pair (h′,li) to yield an update, h of h′.
  - ⋆ The pre-condition clause **pre** li ∈ obs_LIs(h′) ∧ **card** obs_LIs(h′)≥2 expresses that the link identifier li must not be observable h.
  - ⋆ post-condition clause **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except that the argument link identifier is not in the resulting hub.

End of RSL Explanation

s74

20. If the Insert command is of kind 2newH(h',l,h") then the updated net of hubs and links, has
    - the hubs hs joined, ∪, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
21. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has
    21.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.
    21.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.
    21.2 : The set of links augmented by the new link.
22. If the Insert command is of kind 2oldH(hi',l,hi") then
    22.1–.2 : the two connecting hubs are updated to reflect the new link,
    22.3 : and the resulting sets of hubs and links updated.

s75

```
int_Insert(op)(hs,ls) ≡
⋆i   case op of
20      2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
21      1oldH1newH(hi,l,h) →
21.1       let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
21.2       (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
22      2oldH(hi′,l,hi″) →
22.1       let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
22.2                  aLI(xtr_H(hi″,hs),obs_LI(l))} in
22.3       (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** $p_1 \rightarrow c_1$, $p_2 \rightarrow c_2$, $\ldots p_n \rightarrow c_n$ **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.
- $\star_i$ + 20: If op is of the form 2newH(h′,l,h″ then — the narrative explains the rest; else
- $\star_i$ + 21: If op is of the form 1oldH1newH(hi,l,h) then
  - ⋆ 21.1: h′ is the known hub (identified by hi) updated to reflect the new link being connected to that hub,
  - ⋆ 21.2: 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

- $\star_i$ + 22: 22: If op is of the form 2oldH(hi′,l,hi″) then
  - ⋆ 22.1: the first element of the set of two hubs (hsδ) reflect one of the updated hubs,
  - ⋆ 22.2: the second element of the set of two hubs (hsδ) reflect the other of the updated hubs,
  - ⋆ 22.3: 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

s76

23. The remove command is of the form Rmv(li) for some li.
24. 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.

s77

value

23 int_Remove: Rmv → N $\xrightarrow{\sim}$ N
24 int_Remove(Rmv(li))(hs,ls) ≡
24a) **let** l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) **in**
24b) **let** {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} **in**
24c) **let** hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) **in**
24d) (hs\{h′,h″} ∪ hs′,ls\{l}) **end end end**
24a) **pre** li ∈ iols(ls)

cond_rmv: LI × H × H-**set** → H-**set**
cond_rmv(li,h,hs) ≡
24(c)i) **if** obs_HIs(h)={li} **then** {}
24(c)ii) **else** {sLI(li,h)} **end**
pre li ∈ obs_HIs(h)

RSL Explanation

- 23: 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.
- 24: To Remove a link identifier by li from the net (hs,ls) can be formalised as follows:
  - ⋆ 24a): obtain the link l from its identifier li and the set of links ls, and
  - ⋆ 24a): obtain the identifiers, {hi′,hi″}, of the two distinct hubs to which link l is connected;
  - ⋆ 24b): then obtain the hubs {h′,h″} with these identifiers;
  - ⋆ 24c): now examine cond_rmv each of these hubs (see Lines 24(c)i)–24(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).
    - ◦ 24c) 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

- ⋆ 24(c)i): either yields the empty set (of no hubs) if li is the only link identifier inh,
- ⋆ 24(c)ii): or yields a modification of h in which the link identifier li is no longer observable.

End of RSL Explanation

This ends Example 10 ■

2.4 Preliminary Summary s78

acm-wad

So domain descriptions are both informal and formal descriptions: narratives and formalisations of the domain as it is; no references are to be made to requirements let alone to software (being required).

Domain descriptions can never be normative. s79

We should be able to foresee a time, say 10 years from now, ideally, where there are a number of text- and reference-book like domain descriptions for a large variety of domains: air traffic, airports, financial services institutions (banks, brokers, stock and other commodities [metal, crops, oil, etc.] exchanges, portfolio management, credit cards, insurance, etc.), transportation (container lines, airlines, railways, commuter bus transports, etc.), assembly manufacturing, gas and oil pipelines, health care, etc. s80

But although these domain descriptions should represent quite extensive and detailed models they are only indicative. Any one software house which specialises in software (or, in general IT systems) within one (or more) of these domains will, when doing their requirements engineering tasks do so most likely on instantiated, i.e., modified, such domain descriptions.

We will never be able to describe a domain completely.

2.5 Structure of Book s81

In Sect. 3 we motivate why creating and analysing domain models is a necessity for requirements development — as well as, in general, for that of just plainly understanding the man-made world around us.

Key issues of abstraction and modelling need be identified and briefly discussed. This is done in Sect. 4.

In describing domains, in prescribing requirements and in specifying software we must somehow structure our narratives and our formalisations. One way of doing so is around the dual concepts of entities and behaviours and of functions and events. This specification ontology is outlined in Sect. 6.

Sections 7 and 8 constitute the two major sections of this paper.

In Sect. 7 we outline and exemplify domain modelling terms of six facets of, i.e., ways of viewing a domain. These facets are (i) the intrinsics, (ii) the support technologies, (iii) the management and organisation, (iv) the rules and regulations, (v) the scripts and (vi) the human behaviour — as these facets cast different, revealing light on the domain. We consider our emphasis on domain engineering and especially the identification of these facets and the principles and techniques for their description to be a contribution.

In Sect. 8 we show how major aspects of requirements can be systematically “derived” from the domain description. We put double quotes around ‘derived’ so that we can emphasise that the derivation cannot be automated. But major assists can be made in formally relating the resulting domain- and interface requirements prescriptions to the domain description. From Sect. 8 it therefore transpires that we structure our requirements prescriptions into three parts: (i) domain requirements prescriptions, (ii) interface requirements prescriptions and (iii) machine requirements prescriptions. The former, (i), are those requirements which can be expressed sôlely using terms from the domain. The latter, (iii), are those requirements which, in principle, can be expressed sôlely using terms from the machine, that is, the hardware and the software being required. Interface requirements are those which are concerned which phenomena and concepts that are shared between the domain, the environment of the machine, and the machine. Thus interface requirements are expressed using terms from both the domain and the machine. For domain requirements we identify a number of "derivation" operators: projection, instantiation, determination, extension and fitting. And for interface requirements we structure these around requirements for the initialisation and refreshment of shared entities, requirements for interactive computations of shared functions, and so forth.

We consider our emphasis on the "derivation" of requirements and especially the identification of these principles and techniques of both domain and interface requirements to be a contribution.

In Sect. 9 we conclude. Our conclusion includes an outline of future research.

MORE TO COME

• • •

Appendix C presents an ultra-short introduction to the RAISE formal Specification Language RSL [15–17, 58, 59, 61].

2.6 Exercises

See Items 1–2 (of Appendix B, starting Page 195).

3

Motivation for Domain Descriptions

s82 acm-mfdd

There are two basic reasons for creating domain descriptions. One is general and is related to the understanding of the world around and, to some extent, within us; the other is in relation to the development of IT systems, notably software.

3.1 Domain Descriptions of Infrastructure Components s83

We use here a term, ‘infrastructure’, that we ought first define.

according to the World Bank, *‘infrastructure’*[1]*. is an umbrella term for many activities referred to as ‘social overhead capital’ by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users).*[2] s84

We take a more technical view, and see infrastructures as concerned with supporting other systems or activities.

A first reason for pursuing the research and experimental engineering of domain descriptions — both informal narratives and formal specifications — is to achieve understanding, insight and, eventually, theories of domains being thus described. s85

A second reason for domain engineering is to create, not necessarily normative models, but models which can be instantiated to fit a current constellation of a number of these institutions with the aim of studying possible business process re-engineering proposals, yes even to generate such proposals, see Sect. 8.4, or with the aim of software development, see Sect. 3.2. s86

A third reason for domain engineering is to create (again not necessarily normative) descriptions whose narrative parts can be used in company training and in school education,

3.2 Domain Descriptions for Software Development s87

The reasons given in Sect. 3.1 are independent of whether one aims at developing software for a segment of the described domain or not.

But, a reason for pursuing the research and experimental engineering of domain descriptions can, nevertheless be that one wishes to develop software support for entities, operations, events and behaviours and in the supporting technologies, management and organisation, rules and regulations, scripts and the possible human behaviours (that is, the business processes) s88

So here is the dogma that guides us:

[1] Winston Churchill is quoted as having said, in the House of Commons, in 1946: *... the young Labourite speaker, that we just heard, obviously wishes to impress his constituency with the fact that he has attended Eton and Oxford when he uses such modern terms as ‘infrastructure’ ...*

[2] I thank Jan Goossenarts for bringing the text of this paragraph to my attention.

Dogma 1 – **The $\mathcal{D},\mathcal{S} \models \mathcal{R}$ Dogma:** *Before $\mathcal{S}$oftware can be designed we must understand the $\mathcal{R}$equirements, and before $\mathcal{R}$equirements can be expressed we must understand the $\mathcal{D}$omain.* ∎

s89 This dogma entails that we decompose software development into three phases and their attendant *stages:*

Domain Engineering: *identification of domain stake-holders*, *domain acquisition*, *rough sketch of business processes*, *domain analysis and concept formation*, *domain 'terminologisation'*, *the main stages of domain modelling (intertwined with domain model verification)*, *domain*
s90 *validation* and *domain theory formation*.

Requirements Engineering: *identification of requirements stake-holders*, *requirements acquisition*, *rough sketch of business process re-engineering*, *requirements analysis and concept formation*, *requirements 'terminologisation'*, *the main stages of requirements modelling (intertwined with requirements model verification)*, *requirements validation*, *requirements*
s91 *feasibility and satisfiability* and *requirements theory formation*.

Software Design: etcetera.

We shall later show how the requirements acquisition stage is basically a rough sketch version of the the main stages of requirements modelling.

3.3 Discussion s92

The dogma as enunciated above is not "dogmatic".

Engineers of the classical engineering disciplines are all rather deeply educated and trained in the domains of their subject: electronic chip designers are well-versed in plasma physics; aeronautical engineers are well-versed in aerodynamics, and celestial mechanics; mobile phone antenna designers, whether emitters or receivers, are well-versed in applying ("massaging" and calculating
s93 over) Maxwell's equations; et cetera.

No pharmaceutical company would hire a person into their research and development of new medical drugs unless that person had a serious, professional education and training in the scientific, i.e., in the domain disciplines of pharmaceutics. Likewise for structural engineers hired to design suspension or other forms of road and rail bridges: certainly they must be well-versed in structural
s94 engineering.

For a software engineer — to be deployed in the development of software for transportation, or for financial service institutions, or for health care, etc. — to be well-versed in the theories of automata and formal languages, semantics of programming and specification languages, operating systems, compilers, database management systems, etc., is accepted — but what is also needed is an ability to either read existing or to develop new domain descriptions for the fields of respectively transportation, financial service institutions, health care, or similarly.

Part III

Proper Conceptualisation

4

Abstraction & Modelling

s95 acm-aam

4.1 Abstraction

Abstraction relates to conquering complexity of description through the judicious use of abstraction, where abstraction, briefly, is the act and result of omitting consideration of (what would then be called) details while, instead, focusing on (what would therefore be called) important aspects.

4.1.1 From Phenomena to Concepts s96

Phenomena are "things" that we can point to. They are often referred to as 'individuals' since what is pointed to is a single specimen of possibly many "similar" instances of phenomena. We can then, when "figuratively pointing to" an individual (a phenomenon), either keep "talking about" just that one individual, or we can 'abstract' to the class of all 'similar' phenomena. When we do the latter then we have abstracted from a phenomenon, that is, a specific value, to a concept, i.e., to the type of all such values.

4.1.2 From Narratives to Formalisations s97

We describe domains both informally, in terms of concise natural language narratives, and formally, using one or more formal specification languages. The terms of the natural language narrative designate concepts nouns typically denoting types and values of simple entities; verbs typically denoting operations over entities; etcetera. These terms are chosen carefully to correspond, as far as is reasonable in order to achieve a readable natural language text, to names of types, values, operations, etc., of the formal specification.Thus there is, in fact, a "two-way relation" between the s98
choice of mathematical abstractions of the formal specification and the terms of the narrative; the objective is to bring "an as close as possible" relation between the narrative and the formalisation.

4.1.3 Examples of Abstraction s99

Example 10 illustrated two forms of abstraction: (i) model-oriented abstraction and (ii) property-oriented abstraction.

The model-oriented abstraction of Example 10 is illustrated by the modelling of nets as pairs of sets of hubs and links, cf. Item 2 on page 9: $\mathsf{N} = \mathsf{H}\textbf{-set} \times \mathsf{L}\textbf{-set}$, as well as by the concrete type syntax types of link insertion and remove commands and their semantics, Items 9–24d (Pages 11–16).

The property-oriented abstraction of Example 10 is illustrated by the sorts and observers relating to hubs and links, cf. Items 1 on page 9, 3–8 (Pages 10–11).

In this section we shall give some small examples of abstractions. s100

Example 11 – **Model-oriented Directory:**

25. Terminal directory entries are files and files are further undefined.
26. A directory consists of a finite set of uniquely (directory identifier) distinguished entries.
27. A directory is either a file or is a directory.

type
25. FILE, DId
26. DIR = Did $\overrightarrow{m}$ Entry
27. Entry = FILE | DIR

Directories are modelled as maps. The specification abstracts from representation of directory identifiers and files. ∎

s101

Example 12 – **Networked Social Structures:** People live in communities. People of communities may network with people of distinct other communities. And people of such network may network with
s102 people of distinct other networks. We formulate this in a narrative and we formalise the narrative.

28. People are at the heart of any social structure.
29. A region consists of a finite set of one or more communities and a finite set of zero, one or more social networks.
30. A community consists of a non-empty, finite set of people.
31. A social network consists of a non-empty, finite set of two or more people, such that
    a) all people of a network belong to distinct communities of the region, (i.e., no two people of a net belong to the same community),
    b) and, if they also are members of other networks, then they all belong to distinct other networks (i.e., no two people of a network belong to the same other networks),

s103

type
28. P
29. $\mathsf{R}' = \mathsf{C}\text{-}\mathbf{set} \times \mathsf{N}\text{-}\mathbf{set}$, $\mathsf{R} = \{|(\mathsf{cs},\mathsf{ns}):\mathsf{R}'\bullet\mathsf{cs}\neq\{\}|\}$
30. $\mathsf{C}' = \mathsf{P}\text{-}\mathbf{set}$, $\mathsf{C} = \{|\mathsf{c}:\mathsf{C}'\bullet\mathsf{c}\neq\{\}|\}$
31. $\mathsf{N}' == \mathsf{mkN}(\mathsf{sn}:\mathsf{P}\text{-}\mathbf{set})$, $\mathsf{N} = \{|\mathsf{mkN}(\mathsf{ps}):\mathsf{N}'\bullet\mathbf{card}\ \mathsf{ps}\geq 2|\}$

axiom

$\forall\ (\mathsf{cs},\mathsf{ns}):\mathsf{R}\ \bullet$

31a. $\forall\ \mathsf{n}:\mathsf{N}\bullet \mathsf{n}\in\mathsf{ns}\Rightarrow$
$\quad\quad \mathbf{card}\ \mathsf{n} = \mathbf{card}\{\mathsf{c}|\mathsf{c}:\mathsf{C}\bullet\mathsf{c}\in\mathsf{cs}\wedge\mathsf{n}\cap\mathsf{c}\neq\{\}\}\ \wedge$
31b. $\quad \exists\ \mathsf{p}:\mathsf{P}\bullet\mathsf{p}\in\mathsf{n}\ \wedge$
$\quad\quad\quad \exists\ \mathsf{n}':\mathsf{N}\bullet\mathsf{n}'\in\mathsf{ns}\wedge\mathsf{n}\neq\mathsf{n}'\wedge\mathsf{p}\in\mathsf{n}'\Rightarrow$
$\quad\quad\quad\quad \forall\ \mathsf{p}:\mathsf{P}\bullet\mathsf{p}\in\mathsf{n}\Rightarrow$
$\quad\quad\quad\quad\quad \exists\ \mathsf{n}'':\mathsf{N}\bullet\mathsf{n}''\in\mathsf{ns}\wedge\mathsf{n}\neq\mathsf{n}''\wedge\mathsf{p}\in\mathsf{n}''\ \wedge$
$\quad\quad\quad\quad\quad\quad \mathbf{card}\ \mathsf{n}' = \mathbf{card}\{\mathsf{n}''|\mathsf{n}'':\mathsf{N}\bullet\mathsf{n}''\in\mathsf{ns}\wedge\mathsf{n}''\cap\mathsf{n}'\neq\{\}\}$

s104

Formula line $\mathbf{card}\mathsf{n}=\mathbf{card}\{\mathsf{c}|\mathsf{c}:\mathsf{C}\bullet\mathsf{c}\in\mathsf{cs}\wedge\mathsf{n}\cap\mathsf{c}\neq\{\}\}$, the first of the two lines starting with **card**, expresses that the number of persons in the network is the same as the number of the communities to which these persons belong. The fact that $\mathsf{n}\cap\mathsf{c}\neq\{\}$ can be proven to be the same as $\mathbf{card}(\mathsf{n}\cap\mathsf{c})=1$ is left as an exercise.

Formula line $\mathbf{card}\mathsf{n}'=\mathbf{card}\{\mathsf{n}''|\mathsf{n}'':\mathsf{N}\bullet\mathsf{n}''\in\mathsf{ns}\wedge\mathsf{n}''\cap\ \mathsf{n}'\neq\{\}\}$, the second of the two lines starting with **card**, expresses that the number of persons in the network for the case that at least one of the persons in the network is a member of some other network, is the same as the number of the networks to which all other persons of the n must belong. The fact that $\mathsf{n}''\cap\mathsf{n}'\neq\{\}$ can be proven to be the same as $\mathbf{card}(\mathsf{n}''\cap\mathsf{n}')=1$ is left as an exercise. ∎

s105

Example 13 – **Railway Nets:** We bring a variant of Example 10.

32. A railway net consists of one or more lines and two or more stations.

type
  32. RN, LI, ST
value
  32. obs_LIs: RN $\rightarrow$ LI-**set**
  32. obs_STs: RN $\rightarrow$ ST-**set**
axiom
  32. $\forall$ n:RN • **card** obs_LIs(n) $\geq$ 1 $\wedge$ **card** obs_STs(n) $\geq$ 2

33. A railway net consists of rail units. s106

type
  33. U
value
  33. obs_Us: RN $\rightarrow$ U-**set**

34. A line is a linear sequence of one or more linear rail units.

axiom
  34. $\forall$ n:RN, l:LI • l $\in$ obs_LIs(n) $\Rightarrow$ lin_seq(l)

35. The rail units of a line must be rail units of the railway net of the line. s107

value
  34. obs_Us: LI $\rightarrow$ U-**set**
axiom
  35. $\forall$ n:RN, l:LI • l $\in$ obs_LIs(n) $\Rightarrow$ obs_Us(l) $\subseteq$ obs_Us(n)

36. A station is a set of one or more rail units.

value
  36. obs_Us: ST $\rightarrow$ U-**set**
axiom
  36. $\forall$ n:RN, s:ST • s $\in$ obs_STs(n) $\Rightarrow$ **card** obs_Us(s) $\geq$ 1

37. The rail units of a station must be rail units of the railway net of the station. s108

axiom
  37. $\forall$ n:RN, s:ST • s $\in$ obs_STs(n) $\Rightarrow$ obs_Us(s) $\subseteq$ obs_Us(n)

38. No two distinct lines and/or stations of a railway net share rail units.

axiom
  38. $\forall$ n:RN,l,l$'$:LI•{l,l$'$}$\subseteq$obs_LIs(n)$\wedge$l$\neq$l$'$$\Rightarrow$obs_Us(l)$\cap$ obs_Us(l$'$)={}
  38. $\forall$ n:RN,l:LI,s:ST•l $\in$ obs_LIs(n)$\wedge$s $\in$ obs_STs(n)$\Rightarrow$obs_Us(l)$\cap$ obs_Us(s)={}
  38. $\forall$ n:RN,s,s$'$:ST•{s,s$'$}$\subseteq$obs_STs(n)$\wedge$s$\neq$s$'$$\Rightarrow$obs_Us(s)$\cap$ obs_Us(s$'$)={}

39. A station consists of one or more tracks. s109

type
  39. Tr
value
  39. obs_Trs: ST $\rightarrow$ Tr-**set**
axiom
  39. $\forall$ s:ST•**card** obs_Trs(s)$\geq$1

40. A track is a linear sequence of one or more linear rail units.

axiom
40. $\forall$ n:RN,s:ST,t:Tr•s $\in$ obs_STs(n)$\wedge$t $\in$ obs_Trs(s)$\Rightarrow$lin_seq(t)

41. No two distinct tracks share rail units. s110

axiom
41. $\forall$ n:RN,s:ST,t,t$'$:Tr•s $\in$ obs_STs(n)$\wedge$\{t,t$'$\}$\subseteq$obs_Trs(s)$\wedge$t$\neq$t$'$$\Rightarrow$obs_Us(t) $\cap$ obs_Us(t$'$)=\{\}

42. The rail units of a track must be rail units of the station (of that track).

value
42. obs_Us: Tr $\rightarrow$ U-**set**
axiom
42. $\forall$ rn:RN,st:ST,tr:TR •
st $\in$ obs_STs(rn)$\wedge$tr $\in$ obs_Trs(st)$\Rightarrow$obs_Us(tr)$\subseteq$obs_Us(st)

s111 43. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.

value
43. is_Linear: U $\rightarrow$ **Bool**
43. is_Switch: U $\rightarrow$ **Bool**
43. is_Simple_Crossover: U $\rightarrow$ **Bool**
43. is_Switchable_Crossover: U $\rightarrow$ **Bool**

44. A rail unit has one or more connectors.

type
44. K
value
44. obs_Ks: U $\rightarrow$ K-**set**

s112 45. A linear rail unit has two distinct connectors. A switch (a point) rail unit has three distinct connectors. Crossover rail units have four distinct connectors (whether simple or switchable), etc.

axiom
$\forall$ u:U •
is_Linear(u) $\Rightarrow$ **card** obs_Ks(u)=2$\wedge$
is_Switch(u) $\Rightarrow$ **card** obs_Ks(u)=3$\wedge$
is_Simple_Crossover(u) $\Rightarrow$ **card** obs_Ks(u)=4$\wedge$
is_Switchable_Crossover(u) $\Rightarrow$ **card** obs_Ks(u)=4

46. For every connector there are at most two rail units which have that connector in common.

axiom
46. $\forall$ n:RN • $\forall$ k:K • k $\in$ $\cup$\{obs_Ks(u)|u:U•u $\in$ obs_Us(n)\}
$\Rightarrow$**card**\{u|u:U•u $\in$ obs_Us(n)$\wedge$k $\in$ obs_Ks(u)\}$\leq$2

s113 47. Every line of a railway net is connected to exactly two distinct stations of that railway net.

axiom
47. $\forall$ n:RN,l:LI • l $\in$ obs_LIs(n) $\Rightarrow$
$\exists$ s,s$'$:ST • \{s,s$'$\} $\subseteq$ obs_STs(n) $\wedge$ s$\neq$s$'$ $\Rightarrow$
let sus=obs_Us(s),sus$'$=obs_Us(s$'$),lus=obs_Us(l) **in**
$\exists$ u,u$'$,u$''$,u$'''$:U • u $\in$ sus $\wedge$
u$'$ $\in$ sus$'$ $\wedge$ \{u$''$,u$'''$\} $\subseteq$ lus $\Rightarrow$
let sks = obs_Ks(u), sks$'$ = obs_Ks(u$'$),

```
        lks = obs_Ks(u′′), lks′ = obs_Ks(u′′′) in
      ∃!k,k′:K•k≠k′∧sks ∩ lks={k}∧sks′ ∩ lks′={k′}
   end end
```

48. A linear sequence of (linear) rail units is an acyclic sequence of linear units such that neighbouring units share connectors. s114

```
value
  48.  lin_seq: U-set → Bool
       lin_seq(us) ≡
         ∀ u:U • u ∈ us ⇒ is_Linear(u) ∧
           ∃ q:U* • len q = card us ∧ elems q = us ∧
             ∀ i:Nat • {i,i+1} ⊆ inds q ⇒ ∃ k:K • obs_Ks(q(i)) ∩ obs_Ks(q(i+1)) = {k} ∧
             len q > 1 ⇒ obs_Ks(q(i)) ∩ obs_Ks(q(len q)) = {}
```

This ends Example 13 ∎

s115

Example 14 – **A Telephone Exchange:**
The example is “borrowed” — in edited form — from J.C.P. Woodcock and M. Loomes’ book *Software Engineering Mathematics* [143][1]

We start the informal description by presenting a *synopsis* and its immediate *analysis:* s116

- **Synopsis:** The simple telephone exchange system serves to efficiently honour requests for conference calls amongst any number of subscribers, whether immediately connectable, whereby they become actual, or being queued, i.e., deferred (or pending) for later connection. s117
- **Analysis:** The concepts of subscribers and calls are central: In this example we do not further analyse the concept of subscribers. A call is either an actual call, involving two or more subscribers not involved in any other actual calls, or a call is a deferred call, i.e., a requested call that is not actual, because one or more of the subscribers of the deferred call is already involved in actual calls. We shall presently pursue the concepts of requested, respectively actual calls, and only indirectly with deferred calls.

s118

The structure of the types of interest are first described. We informally describe first the basis types, then their composition. (i) Subscribers: There is a class (S) of further undefined subscribers. (ii) Connections: There is a class (C) of connections. A connection involves one subscriber, the ‘caller’, and any number of one or more other subscribers, the ‘called’. (iii) Exchange: At any time an exchange s119
reflects (i.e., is in a state which records) a number of requested connections and a number of actual connections (a) such that no two actual connections share any subscribers, (b) such that all actual connections are also requested connections, and (c) such that there are no requested calls that are not actual and share no subscribers in common with any other actual connection. (That is: The actual connections are all that can be made actual out of the requested connections. This part addresses the efficiency issue referred to above.) (iv) Requested connections: The set of all requested connections for s120
a given exchange forms a set of connections. (v) Actual connections: The set of all actual connections, for a given exchange, forms a subset of its requested connections such that no two actual connections share subscribers.

In this example we shall also be able to refer to the exchange, later to be named X, as ‘the state’ (of the telephone exchange system). We shall later have a great deal more to say about the concept of state.

[1] Permission to “lift” this example (converting it, however, from Z into RAISE, and providing it with a property-oriented solution) has been granted by Prof., Dr J.C.P. Woodcock (February 28, 2001).

Formalisation of Property-oriented State s121

```
type
   S, C, X
value
   obs_Caller: C → S
   obs_Called: C → S-set
   obs_Requests: X → C-set
   obs_Actual: X → C-set

   subs: C → S-set
   subs(c) ≡ obs_Caller(c) ∪ obs_Called(c)

   subs: C-set → S-set
   subs(cs) ≡ ∪ { subs(c) | c:C • c ∈ cs }
```

s122 The overloaded function name subs stands for two different functions. One observes ("extracts") the set of all subscribers said to be engaged in a connection. The other likewise observes the set of all subscribers engaged in any set of connections. We shall often find it useful to introduce such *auxiliary*

s123 *functions*.

```
axiom
[1]  ∀ c:C, ∃ s:S •
[2]     s = obs_Caller(c) ⇒ s ∉ obs_Called(c),

[3]  ∀ x:X •
[4]     let rcs = obs_Requests(x),
[5]        acs = obs_Actual(x) in
[6]     acs ⊆ rcs ∧
[7]     ∀ c,c′:C • c ≠ c′ ∧ {c,c′} ⊆ acs ⇒
[8]        obs_Caller(c) ≠ obs_Caller(c′) ∧
[9]        obs_Called(c) ∩ obs_Called(c′) = {} ∧
[10]      ∼∃ c:C • c ∈ rcs \ acs •
[11]           subs(c) ∩ subs(acs) = {} end
```

Let us annotate the above specification. [1] For all connections there exists a subscriber such that [2] the subscriber is a caller, but not a called subscriber. [3] For all telephone exchanges (i.e., telephone exchange states), [4–5] let us observe the requested and the actual connections. [6] The actual ones must also be requested connections, and [7] for any two different actual connections, [8] their callers must be different, [9] the callers and the ones called cannot share subscribers, and [10] there must not be a requested, but not actual connection [11] which could be an actual connection. That is all such

s124 connections must have some subscriber in common with some actual connection.

The last two lines above express the efficiency criterion mentioned earlier.

We can express a law that holds about the kind of exchanges that we are describing:

```
theorem
   ∀ x:X •
      obs_Actual(x)={} ≡ obs_Requests(x)={}
```

s125 The *law* expresses that there cannot be a non-empty set of deferred calls if there are no actual calls. That is, at least one deferred call can be established should a situation arise in which a last actual call is terminated and there is at least one deferred call.

The *law* is a *theorem* that can be *proved* on the basis of the telephone exchange system *axioms* and a *proof system* for *sets.*

Operation Signatures *s126*

The following operations, involving telephone exchanges, can be performed: (i) Request: A caller indicates, to the exchange, the set of one or more other subscribers with which a connection (i.e., a call) is requested. If the connection can be effected then it is immediately made actual, else it is deferred and (the connection) will be made actual once all called subscribers are not engaged in any actual call. (ii) Caller_Hang: A caller, engaged in a requested call, whether actual or not, can hang up, i.e., terminate, if actual, and then on behalf of all called subscribers also, or can cancel the requested (but not yet actual) call. (iii) Called_Hang: Any called subscriber engaged in some actual call can leave s127
that call individually. If that called subscriber is the only called subscriber ("left in the call"), then the call is terminated, also on behalf of the caller. (iv) is_Busy: Any subscriber can inquire as to whether any other subscriber is already engaged in an actual call. (v) is_Called: Any subscriber can inquire as to the identities of all those (zero, one or more) callers who has requested a call with the inquiring subscriber. s128
First the signatures:

value
  newX: **Unit** $\rightarrow$ X
  request: S $\times$ S-**set** $\rightarrow$ X $\rightarrow$ X
  caller_hang: S $\rightarrow$ X $\xrightarrow{\sim}$ X
  called_hang: S $\rightarrow$ X $\xrightarrow{\sim}$ X
  is_busy: S $\rightarrow$ X $\rightarrow$ **Bool**
  is_called: S $\rightarrow$ X $\rightarrow$ **Bool**

s129
The *generator function* newX is an *auxiliary function.* It is needed only to make the *axioms* cover all *states* of the telephone exchange system. In a sense it *generates* an *empty*, that is, an *initial state.* Usually such *empty state generator functions* are "paired" with a similar *test for empty state observer function.* s130
Then we get the axioms:

axiom
  $\forall$ x:X • obs_Requests(x)={} $\equiv$ x=newX(),
  $\forall$ x:X,s,s′:S,ss:S-**set** •
    $\sim$is_busy(s,newX()) $\wedge$
    s$\neq$s′ $\Rightarrow$
      s $\in$ ss $\Rightarrow$ is_busy(s)(request(s′,ss)(x)) $\wedge$
      s $\notin$ ss $\Rightarrow$ is_busy(s)(request(s′,ss)(x)) $\equiv$ is_busy(s)(x),
      ... etcetera ...

s131
We leave the axiom incomplete. Our job was to illustrate the informal and formal parts of a property-oriented specification, not to do it completely.

Model-oriented State *s132*

type
  S
  C = {| ss | ss:S-**set** • **card** ss $\geq$ 2 |}
  R = C-**set**
  A = C-**set**
  X = {| (r,a) | (r,a):R$\times$A • a $\subseteq$ r $\wedge$ $\bigcap$ a = {} |}

Efficient States s133

There is a notion of telephone exchange system efficiency, a constraint that governs its operation, hence the state, at any one time. The efficiency criterion says that all requested calls that can actually be connected are indeed connected:

value
  eff_X: X $\xrightarrow{\sim}$ **Bool**
  eff_X(r,a) $\equiv$ $\sim\exists$ a′:A • a $\subset$ a′ $\wedge$ (r,a′) $\in$ X

Formalisation of Action Types s134

type
  Cmd = Call | Hang | Busy
  Call′ == mk_Call(p:S,cs:C)
  Call = {| c:Call′ • **card** cs(c) $\geq$ 1 }
  Hang == mk_Hang(s:S)
  Busy == mk_Busy(s:S)

Pre/Post and Direct Operation Definitions s135

We shall, for each operation, define its meaning both in terms of pre/post conditions and in terms of a direct "abstract data type algorithm".

Multi-party Call s136

A multi-party call involves a (primary, s) caller and one or more (secondary, ss) callees. Enacting such a call makes the desired connection a requested connection. If none of the callers are already engaged in an actual connection then the call can be actualised. A multi-party call cannot be made by a caller
s137 who has already requested other calls.

value
  int_Call: Call $\xrightarrow{\sim}$ X $\xrightarrow{\sim}$ X
  int_Call(mk_Call(p,cs))(r,) **as** (r′,a′)
  pre p $\notin \bigcup$ r
  post r′ = r $\cup$ {{p} $\cup$ cs} $\wedge$ eff_X(r′,a′)

  int_Call(mk_Call(p,cs))(r,a) $\equiv$
    let r′ = r $\cup$ {{p} $\cup$ cs},
      a′ = a $\cup$ **if** ({{p} $\cup$ cs} $\cap \bigcup$ a) = {}
          then {{p} $\cup$ cs} **else** {} **end in**
    (r′,a′) **end**
    pre p $\notin \bigcup$ r

The above **pre/post**-definition (of int_Call) illustrates the power of this style of definition. No algorithm is specified, instead all the work is expressed by appealing to the invariant!

Call Termination s138

It takes one person, one subscriber, to terminate a call.

value
int_Hang: Hang → X $\stackrel{\sim}{\to}$ X
int_Hang(mk_Hang(p))(r,a) **as** (r′,a′)
pre existS c:C • c ∈ a ∧ p ∈ a
post r′ = r \ {c|c:C • c ∈ r ∧ p ∈ c} ∧ eff_X(r′,a′)

int_Hang(mk_Hang(p))(r,a) ≡
let r′ = r \ { c | c:C • c ∈ a ∧ p ∈ c },
a′ = a \ { c | c:C • c ∈ r ∧ p ∈ c } **in**
let a′′ = a′ ∪ { c | c:C • c ∈ r′ ∧ c ⋂ a′ = {} } **in**
(r′,a′′) **end end**
pre existS c:C • c ∈ a ∧ p ∈ a

The two ways of defining the above int_Hang function again demonstrate the strong abstractional feature of defining by means of **pre**/**post**-conditions.

Subscriber Busy s139

A line (that is, a subscriber) is only ‘busy’ if it (the person) is engaged in an actual call.

value
int_Busy: S → X $\stackrel{\sim}{\to}$ **Bool**
int_Busy(mk_Busy(p))(_,a) **as** b
pre true
post if b **then** p ∈ ⋃ a **else** p ∉ ⋃ a **end**

int_Busy(mk_Busy(p))(_,a) ≡ p ∈ ⋃ a

Here, perhaps not so surprisingly, we find that the explicit function definition is the most straightforward.
This ends Example 14 ■

4.1.4 Mathematics and Formal Specification Languages s140

Using mathematical concepts has shown to be the most powerful way of expressing abstractions. The discrete mathematical concepts of sets, Cartesians, sequences, maps, that is, enumerable functions and functions, as well as mathematical logic and algebras has served mathematicians well for quite some time and will serve professional software engineers well. s141

Formal specification languages, like Alloy [87], Event B [1,31], RSL [15–17,58,59,61], VDM [24,25,53,54], Z [74,75,134,135,142] and others, embody the above-mentioned mathematical concepts in quite readable forms. The current book favours the RAISE specification language RSL. The first use of the RSL notation was in Example 10 starting Page 9. That first use of RSL will be extensively annotated and with margin references to sections and pages of Appendix C.

4.2 Modelling s142

Definition 7 ***Model:*** *A model is the mathematical meaning of a description of a domain, or a prescription of requirements, or a specification of software, i.e., is the meaning of a specification of some universe of discourse.* ■

s143

Definition 8 ***Modelling:*** *Modelling is the act (or process) of identifying appropriate phenomena and concepts and of choosing appropriate abstractions in order to construct a model (or a set of models) which reflects appropriately on the universe of discourse being modelled.* ■

4.2.1 Property-oriented Modelling s144

Definition 9 ***Property-oriented Modelling:*** *By property-oriented modelling we shall understand a modelling which emphasises the properties of what is being modelled, through suitable use of abstract types, that is, sorts, of postulated observer (**obs_**), generator (**mk_**) and type checking (**is_**) functions, and axioms over these.* ▪

4.2.2 Model-oriented Modelling s145

Definition 10 ***Model-oriented Modelling:*** *By abstract, but model-oriented modelling we shall understand a modelling which expresses the properties of what is being modelled, through suitable use of mathematical concepts such as sets, Cartesians, sequences, maps (finite domain, enumerable functions), and functions (in the sense of λ-Calculus functions).* ▪

4.3 Model Attributes s146

Specifications achieve their intended purpose by emphasising one or more attributes. Either: (i.1) analogic, (i.2) analytic and/or (i.3) iconic; and then either: (ii.1) descriptive or (ii.2) prescriptive;
s147 and finally either: (iii.1) extensional or (iii.2) intensional. That is, a model may, at the same time (although time has nothing to do with this aspect of models), be one or more of analogic, analytic and iconic; expressed either only descriptive, or mostly descriptive (with some prescriptive aspects), or only prescriptive, or mostly prescriptive (etc.); and expressed either only extensional, or mostly extensional (with some aspects), or only intensional, or mostly intensional (etc.). We may claim that a good model blends the above consciously and judiciously — including featuring exactly (or primarily) one attribute from each of the three categorisations. We next take a look at these model attributes.

4.3.1 Analogic, Analytics and Iconic Models s148

Definition 11 ***Analogic Model:*** *An analogic model resembles some other universe than the universe of discourse purported to be modelled.* ▪

Definition 12 ***Analytic Model:*** *An analytic model is a mathematical specification: It allows analysis of the universe of discourse being modelled.* ▪

Definition 13 ***Iconic Model:*** *An iconic model is an "image" of the universe of discourse that is the target of our attention.* ▪

s149

Example 15 – **Analogic, Analytic and Iconic Models:** We lump three kinds of examples into one larger example:

- *Analogic models:* (1) The symbol, on the visual display screen of your computer, of a `trash can`, denoting an ability to delete files.[2] (2) A four-pole, electric circuit network of resistors, inductances, capacitors and current or voltage supplies can be used to analogically model some aspects of the behaviour of certain mechanical vibration and/or spring dampening aggregations. (3) A tomographic image of, say the brain, with its colour-enhanced "blots" is an analogic model of a cross section of
s150 that brain!

[2]Thus `Macintosh` systems, although undoubtedly so-called "user-friendly", got the concept name wrong: The `Macintosh` 'trash can' symbol is not an icon! It is an analogic!

- *Analytic models:* (4) The differential equations whose variables model spatial x, y, z coordinates and the temporal t dimension, and whose constant, m, model the mass of a stone, may be an analytic model of the dynamics of the throwing of such a stone in a vacuum. (5) A description, in `RSL`, involving quantities that purport to model bank accounts, their balance, time, etc., may be an analytic model of a banking system — in the real world — provided the model reflects at least "some of the things that can go wrong" in actual life. (6) A graph with labelled nodes and weighted arcs may be used as a model of a road net with cities and distances between these, and can be used for the computation of shortest distances, etc. s151
- *Iconic models:* Typical iconic models are certain advisory or judicially binding traffic signs: (7) The roadside sign showing, typically in Sweden, an `Elk`, denoting that elks may be crossing the road ahead at any time; (8) the roadside sign showing an `automobile` (from behind) `"underlined" with two crossing S curves`, denoting that the road surface ahead may be slippery and hence that automobiles may spin out of control; and (9) the roadside sign showing a `crossed-out horn`, denoting that use of the automobile horn is not allowed.

Observe that a model may possess characteristics of more than one of the above attributes. ■

4.3.2 Descriptive and Prescriptive Models s152

Definition 14 ***Descriptive Model:*** *A descriptive model*[3] *describes something already existing.* ■

Definition 15 ***Prescriptive Model:*** *A prescriptive model*[4] *models something as yet to be implemented.* ■

s153

Thus domain specifications are descriptive, while requirements specifications are prescriptive. A requirements specification prescribes properties that the intended software (cum computing system) shall satisfy. A software specification prescribes certain kinds of computations. s154

We remind the reader that we use the terms model and specification near synonymously. A specification defines a set of zero, one or more, possibly even an infinity, of models. But we use the term the model in connection with a given specification to stand for the general member of the set of models. Hence when we use the term model below, please read specification. s155

Example 16 – **Descriptive and Prescriptive Models:** *A descriptive model:* A railway net *consists* of two or more distinct stations and one or more distinct railway lines. A railway line *consists* of a linear sequence of one or more linear rail units. Any railway line *connects* exactly two distinct stations. A route *is* a sequence of one or more, and if more, then connected railway lines. Two railway lines *are* connected if they have the connecting station in common. s156

A prescriptive model: The train timetable *shall,* for each train journey, list all station visits. A train timetable station visit *shall* list the name of the station visited, the time of arrival of the train, the time of departure of the train. No train timetable train journey entry lists the same station twice. Times of train departures and train arrivals *shall* be compatible with reasonable stops at stations and with the distance between stations visited. Two immediately time-consecutive train timetable station visits *must* be compatible with the railway net: It shall be possible to route a train between such consecutive stations. ■

s157

Notice in the descriptive model the unhedged use of the verbs *consists*, *connects*, *is* and *are*. A description is *indicative:*[5] It tells *what there is.* Likewise notice in the prescriptive model the use of the (compelling) verbs *shall* and *must*. A prescription is *putative:*[6] It tells *what there will be.*

---

[3] *Descriptive:* factually grounded or informative rather than normative, prescriptive, or emotive [136].

[4] *Prescriptive:* to lay down a rule [136].

[5] *Indicative:* of, relating to, or constituting a verb form or set of verb forms that represents the denoted act or state as an objective fact [136].

[6] *Putative:* of "putare" to think, assumed to exist [136].

4.3.3 Extensional and Intensional Models s158

Definition 16 ***Extensional Model:*** *An extensional model*[7] *(black, opaque box) presentation models something as if observed by someone external to the universe of discourse.* ▪

Definition 17 ***Intensional Model:*** *An intensional model*[8] *in logic, correlative words that indicate the reference of a term or concept. Intension indicates the internal content of a term or concept that constitutes its formal definition. .* ▪

s159

Intensional versus extensional meaning: (i) intensional meaning: consists of the qualities or attributes the term *connotes* (the attributes of class membership); (ii) extensional meaning: consists of the qualities or attributes the term denotes (the class members themselves).

Connotation: the suggesting of a meaning by a word apart from the thing it explicitly names or describes.

Denotation: a direct specific meaning as distinct from an implied or associated idea. (glass (or
s160 white), transparent box) presentation models the internal structure of the *universe of discourse*

An *extensional model* presents, i.e., reflects, the behaviour as seen from an outside. In that sense one may claim, but the claim cannot be justified from extensionality alone, that an extensional model focuses on properties, on what the thing that is being modelled offers an outside world, i.e., users of that thing. If a model is expressed in a property-oriented style, then we can claim the converse: that the model is extensional!

An *intensional model* presents the internal mechanisms of what is being modelled in a way
s161 that may explain why it has the extension that it might have.

The subject of intension and extension, in mathematical logic as well as in philosophy, is not a closed book. It is still very much subject to analysis, redefinition, rethinking. In this paper we shall restrict ourselves to the views expressed above. Extension is the "black box" view of observing only externally perceivable properties of what is being specified. Intension is the "glass box" view of observing some, most or all of the inner workings of what is being specified.

Example 17 – **Extensional Model Presentations:** (1) To explain the square root function, $\sqrt{n} = r$, by explaining that $r \times r = n \wedge\ r \geq 0$ is to give an extensional definition, hence model.

(2) To explain a stack extensionally we may define (a) the stack sorts for elements and stacks, (b) the signatures of the empty, pop, top and push functions, and (c) the axioms which relate sorts and operations. ▪

s162

Example 18 – **Intensional Model Presentations:** (1) To explain the square root function, $\sqrt{n}$, by presenting, e.g., the Newton–Raphson algorithm ([118] pp. 230, 347, 355 and 360), is to give an intensional definition, hence model.

(2) An intensional model of stacks may model stacks as lists of (extensionally modelled) elements, and define the (i) empty, (ii) pop, (iii) top, and (iv) push functions in terms of (i) constructing the empty list, of (ii) yielding the tail of a list, of (iii) yielding the head element of a list, and (iv) of concatenating a supplied element to the front of the list. ▪

4.4 Rôles of Models s163

We pursue modelling for one or more reasons:

(i) *To gain understanding:* in the process of modelling we are forced to come to grips with many issues of the universe of discourse.

(ii) *To get inspiration and to inspire:* abstraction often invites such generalisations that induce, in the writer, or in the reader, desires of change.

(iii) *To present, educate and train:* a model can serve as the basis for presentations to others
s164 for the purposes of awareness, education or training.

[7] *Extensional:* concerned with objective reality.
[8] *Intensional: TBD*

(iv) *To assert and predict:* a mathematical, including a formal model, usually allows abstract interpretation — in the "vernacular": calculations, computations — that simulates, estimates or otherwise expresses potential properties of the universe of discourse.

(v) *To implement:* two kinds of implementations can be suggested: in business process re-engineering we propose the re-engineeringof some domain on the basis of a model and in *computing systems design* we base the development of requirements on a domain specification and we base software design on requirements.

4.5 Exercises

See Items 3–4 (of Appendix B, starting Page 195).

5

Semiotics

s165 acm-semiotics

5.1 An Overview

The following decomposition of language concepts and their explication is taken from [5]. The syntax-semantics-pragmatics sub-structuring is believed to be due to Morris [32–34, 105, 106, 144]. In fact, the term ‘semiotics’, as we use it, is really to be taken in its widest meaning [35].

Definition 18 – **Semiotics:** *Semiotics is the study of and knowledge about the structure of all ‘sign systems’. We divide this study (and our knowledge) into three parts: syntax, semantics and pragmatics.* ■

Conventional natural language as spoken (written) by people is but one ‘sign’ system. Examples of sign systems are sound (audio), sight (visual), touch (tactile), smell and taste — and in all contexts: dance, film, politics, eating and clothing [35]. s166

Definition 19 – **Syntax:** *Syntax is the study of and knowledge about how signs (words) can be put together to form correct sentences and of how sentence-signs relate to one another.* ■

We shall understand signs (words) and sentences in a wide sense. Programs in a programing languages and specifications in a formal specification language will here be considered to be sentences. and variable and function identifiers (a, ab, id, fct, etc.); constants (0, 1, 2, ..., **true, false, chaos**, etc.); expressions and statements; statement and expression symbols (such as value operators s167
($+, -, /, *, \times, \mapsto$, etc.), and **dom, rng, elems, len, card**) etc.; type operators (`Boolean`, `integer`, `real`, `char`, `string`, etc., and **-set**, **-infset**, *, $^\omega$, $\rightarrow$, $\stackrel{\sim}{\rightarrow}$, $\underset{m}{\rightarrow}$, $\times$); parentheses ((,), {, }, [,] etc.); comma (,); semicolon (;); assignment symbols ($:=$, $=$, $\leftarrow$); definition symbols ($\equiv$, $::=$) etc.) and literals (such as **begin, end, let, in, cases, of, while, do, type, value, axiom**, etc.) will here be considered words. s168

But also diagrams, say technical drawings and actual layout of, for example, buildings and railway tracks, will be considered sentences, and the boxes and lines of diagrams, and the various visual (proper) sub-components of actual physical phenomena will be considered words. That is, we s169
consider phenomena such as geographical and geodetic maps; buildings, and their “accompanying” architectural and engineering drawings; railway tracks (lines and stations) and their “accompanying” engineering drawings; cities and city plans; etc., as languages [4]. GIS and CAD/CAM systems thus translate descriptions of such phenomena into database structures and GIS and CAD/CAM system user commands are compiled and executed as language programs in the context of these databases. s170

We define syntaxes in terms of either BNF grammars or `RSL` types. We distinguish between syntactic types and semantic types. The syntactic types designate sentences and words. The semantic types designate meanings (of sentences and words).

Definition 20 – **Semantics:** *Semantics is the study of and knowledge about the meaning of words, sentences, and structures of sentences.* ∎

s171 Let

type
  SynType, SemType

designate syntactic, respectively semantic types. Then

value
  M: SynType → SemType

presents the signature of a semantic function M. It assumes all syntactic inputs to be well-formed. If the syntax of SynType is such as to allow ill-formed sentences, then we must define a well-formedness function:

value
  Wf_SynType: SynType → **Bool**

and the signature of M must be sharpened:

value
  M: SynType $\xrightarrow{\sim}$ SemType

s172

Definition 21 – **Pragmatics:** *Pragmatics is the study of and knowledge about the use of words, sentences and structures of sentences, and of how contexts affect the meanings of words, sentences, etc.* ∎

5.2 Syntax s173

We recall our definition: syntax is the study of and knowledge about how signs (words) can be put together to form correct sentences and of how sentence-signs relate to one another.

We shall divide our presentation of syntax into three parts: (i) BNF grammars, (ii) concrete type syntax and (iii) abstract type syntax. These three are just three increasingly more abstract
s174 ways of dealing with syntax.

The subject of syntax goes well beyond our software engineering treatment. The computer science topic of *formal languages and automata theory* studies far wider consequences of grammars and syntax than we cover. Seminal books are [80][1] The computing science topic of *regular expression recognizers* and *context free language parsers* likewise goes well beyond our coverage — and their study is important for the software engineer to implement efficient software for language
s175 handling. Seminal books are [3].[2]

BNF grammars define sets of strings of characters; concrete type syntaxes define sets of mathematical structures (numbers, Booleans, sets, Cartesians, maps and functions over concrete type values); and abstract type syntaxes define properties of simple phenomena and concept entities. We say that BNF grammars and concrete type syntaxes define simple phenomena and concept entities in a *model-oriented* fashion whereas abstract type syntaxes define simple phenomena and concept entities in a *property-oriented* fashion.

---

[1] Dines: Find more recent references.
[2] Dines: Find more recent references.

5.2.1 BNF Grammars s176

BNF stands for `Backus Naur Form`. BNF grammars, as we shall see, stand for sets of finite length strings of characters, including blanks and punctuation marks.

Definition 22 – **Character:** *A character is a symbol that can be displayed (on paper, on a computer screen, or otherwise).* ∎

Example 19 – **Characters:** Our example is the conventional example of characters from an English/American computer keyboard: a, A, b, B, c, C, ..., z, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, !, @, #, $, %, &, *, ˜, (,), {, }, [,], -, +, ', ", <, >, ., ,, :, ;, ?, /, |, [blank], etc. ∎ s177

Definition 23 – **Alphabet:** *An alphabet is a finite set of characters.* ∎

Example 20 – **Alphabet:** Three examples, $\mathcal{A}_i, \mathcal{A}_j, \mathcal{A}_k$, of subsets of the above characters:

- $\mathcal{A}_i$: {a,b,[blank],O,|}
- $\mathcal{A}_j$: {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,[blank]}
- $\mathcal{A}_k$: {0,1,2,3,4,5,6,7,8,9,[blank],a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}

∎ s178

Definition 24 – **Terminal:** *By a terminal we understand a sequence of one or more characters of some given alphabet.* ∎

Example 21 – **Terminals:**

- a, b, c, 0, 1;
- a, aa, aaa, abc, 0, 1, 00, 01, 001;
- open, deposit, withdraw, close, account, client, number.

∎ s179

Definition 25 – **Non-terminal:** *By a non-terminal we understand a specially highlighted sequence of one or more characters, not necessarily from the alphabet of a given set of terminals.* ∎

Example 22 – **Non-terminals:**

- ⟨Command⟩,
- ⟨Open⟩, ⟨Deposit⟩, ⟨Withdraw⟩, ⟨Close⟩
- ⟨ClientName⟩, ⟨AccountNumber⟩,
- ⟨Cash⟩, ⟨Amount⟩

∎ s180

Definition 26 – **BNF Rules:** *By a* BNF *rule we understand a triple: ($\mathcal{L}$, ::=, $\mathcal{A}s$) where $\mathcal{L}$ is a non-terminal and $\mathcal{A}s$ is a finite set of zero, one or more alternatives where an alternative is a finite sequence of zero, one or more non-terminals and/or terminals. ::= is the definition symbol.* ∎

Example 23 – **BNF Rules:**

- ⟨Command⟩ ::= ⟨Open⟩ | ⟨Close⟩
- ⟨Open⟩ ::= client ⟨ClientName⟩ opens account
- ⟨Withdraw⟩ ::= client ⟨ClientName⟩ withdraws ⟨Amount⟩
  from account ⟨AccountNumber⟩

∎ s181

Definition 27 – **BNF Grammar:** *By a* BNF *grammar we understand a quadruple*

$$(\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{G})$$

- *$\mathcal{N}$ is an alphabet of non-terminals,*
- *$\mathcal{T}$ is an alphabet of terminals,*
- *$\mathcal{R}$ is a set of rules,*
- *$\mathcal{G}$ is a non-terminal,*

such that $\mathcal{G}$ is in $\mathcal{N}$; all the left hand sides of rules in $\mathcal{R}$ are in $\mathcal{N}$; all the non-terminals of right hand sides of rules in $\mathcal{R}$ are distinct and together form $\mathcal{N}$; and all the terminals of right hand sides of rules in $\mathcal{R}$ are in $\mathcal{T}$. ∎

s182

Example 24 – **BNF Grammar: Banks:** The ... (*Identifiers,Alphanumerics,Numerals*) are not part of the syntax.

$\mathcal{N}$: {⟨Command⟩, ⟨Open⟩, ⟨Deposit⟩, ⟨Withdraw⟩, ⟨Close⟩, ⟨ClientName⟩,
⟨AccountNumber⟩, ⟨Amount⟩}
$\mathcal{T}$: {client,opens account,deposits,into account,withdraws,from account,closes account} ...
... ∪ *Identifiers* ∪ *Alphanumerics* ∪ *Numerals*
$\mathcal{R}$: ⟨Command⟩ ::= ⟨Open⟩ | ⟨Deposit⟩ | ⟨Withdraw⟩ | ⟨Close⟩
⟨Open⟩ ::= client ⟨ClientName⟩ opens account
⟨Deposit⟩ ::= client ⟨ClientName⟩ deposits ⟨Amount⟩ into account ⟨AccountNumber⟩
⟨Withdraw⟩ ::= client ⟨ClientName⟩ withdraws ⟨Amount⟩ from account ⟨AccountNumber⟩
⟨Close⟩ ::= client ⟨ClientName⟩ closes account ⟨AccountNumber⟩
⟨ClientName⟩ ::= ... *Identifiers*
⟨AccountNumber⟩ ::= ... *Alphanumerics*
⟨Amount⟩ ::= ... *Numerals*
$\mathcal{G}$: ⟨Command⟩

∎

s183

The "... ∪ *Identifiers* ∪ *Alphanumerics* ∪ *Numerals*" which is not part of the syntax, ought be fully defined by a somewhat longer BNF grammar.

The example showed one form of BNF grammars. In the below definition of the meaning of BNF grammars we abstract from the above forms of rules for BNF grammars.

Examples 26 on the facing page and 28 on page 45 follow up on this example of a BNF grammar by presenting a concrete type syntax, respectively an abstract type syntax for "supposedly" the
s184 same command language.

Definition 28 – **Meaning of a BNF Grammar:** *The meaning of a BNF grammar is a language, that is, a possibly infinite set of finite length strings over the terminal alphabet of the BNF grammar. To properly define this language, for any BNF grammar we shall proceed, formally, as follows:*

Let N and T denote the alphabets of non-terminals and terminals.

Let $r : (n, \{\ell_1, \ell_2, \ldots, \ell_m\})$ designate a rule, r, that is: $n : N$ and $\ell_i : (N|T)^*$, for $m \geq 0, 1 \leq i \leq m$ where $(N|T)^*$ denotes the possibly infinite set of finite length strings over non-terminal and terminal characters.

Let $G : (N, T, R, n_0)$ where G names the grammar, N the alphabet of non-terminals, T the alphabet of terminals, R the finite set of rules (over N and T), and n_0 a distinguished non-terminal of N.

*G is constrained as follows: no two distinct rules, (n, ls) and (n', ls') in G have $n = n'$, that is: all left hand side non-terminals are distinct and together they form N, and there is a rule
s185 $r : (n, \{\ell_1, \ell_2, \ldots, \ell_m\})$ in R such that $n = n_0$.*

Let $s_i \widehat{\ } s_j$ denote the concatenation of strings s_i and s_j.

Let $s \widehat{\ } U \widehat{\ } s'$ be a string over $(N|T)^*$.

Let $(U, \{\ell_1, \ell_2, \ldots, \ell_i, \ldots, \ell_m\})$ be a rule in R.

Then $s \widehat{\ } U \widehat{\ } s' \ \rightarrow_G \ s \widehat{\ } \ell_i \widehat{\ } s'$ means: from $s \widehat{\ } U \widehat{\ } s'$, by means of rule $(U, \{\ell_1, \ell_2, \ldots, \ell_m\})$ of G, we derive, $\rightarrow_G$, $s \widehat{\ } \ell_i \widehat{\ } s'$.

If, in some rule $(U, \{\ell_1, \ell_2, \ldots, \ell_i, \ldots, \ell_m\}), m = 0$, that is, the rule is $(U, \{\})$, then $s \widehat{\ } U \widehat{\ } s' \ \rightarrow_G \ s \widehat{\ } s'$.

If $s_p \rightarrow_G s_q$, $s_q \rightarrow_G s_r$, ..., and $s_v \rightarrow_G s_w$, then $s_p \rightarrow_G{}^* s_w$ (and thus $s_p \rightarrow_G{}^* s_r$, $s_p \rightarrow_G{}^* s_w$, etc. — assuming $s_p \rightarrow_G{}^* s_v$).

The meaning of $\rightarrow_G$ is specific to the given Grammar.

Now the meaning, $\mathcal{L}(G)$ is defined as follows:

$$\mathcal{L}_G = \{s \mid \ n_0 \rightarrow_G{}^* s \wedge \ s \in T^*\}$$

■ s186

Some BNF grammars are such that $\mathcal{L}_G$ is empty: no derivation, $\rightarrow_G$, and hence $\rightarrow_G{}^*$, results in terminal strings. s187

Example 25 – **Meaning of a BNF Grammar:** First we show a form of BNF grammar which is more in line with the above definition.

$\mathcal{N} = \{E, C, V, P, I, B\}$,
$\mathcal{T} = \{0, 1, 2, 3, 4, 5, \ldots, a, b, c, \ldots, z, +, -, *, /,$
$\mathcal{R} =$
0 $\{\ \ E = C \mid V \mid P \mid I \mid B,$
1 $\quad C = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \ldots,$
2 $\quad V = a \mid b \mid c \mid \ldots \mid z,$
3 $\quad P = -E,$
4 $\quad I = E\ O\ E,$
5 $\quad O = + \mid - \mid * \mid /,$
6 $\quad B = (\ E\ )\ \}$
$n_0 = E$ s188

Then we show a derivation of the expression $5 + (a/3) + -c$ from E:

$E \ \rightarrow$	0	$5 + (\ E\ O\ E\ )\ O\ E \ \rightarrow$	0
$I \ \rightarrow$	4	$5 + (\ V\ O\ E\ )\ O\ E \ \rightarrow$	2
$E\ O\ E \ \rightarrow$	0	$5 + (\ a\ O\ E\ )\ O\ E \ \rightarrow$	5
$C\ O\ E \ \rightarrow$	1	$5 + (\ a\ /\ E\ )\ O\ E \ \rightarrow$	0
$5\ O\ E \ \rightarrow$	5	$5 + (\ a\ /\ C\ )\ O\ E \ \rightarrow$	1
$5 + E \ \rightarrow$	0	$5 + (\ a\ /\ 3\ )\ O\ E \ \rightarrow$	5
$5 + I \ \rightarrow$	4	$5 + (\ a\ /\ 3\ ) + E \ \rightarrow$	0
$5 + E\ O\ E \ \rightarrow$	6	$5 + (\ a\ /\ 3\ ) + P \ \rightarrow$	3
$5 + B\ O\ E \ \rightarrow$	4	$5 + (\ a\ /\ 3\ ) + -E \ \rightarrow$	0
$5 + (\ E\ )\ O\ E \ \rightarrow$	0	$5 + (\ a\ /\ 3\ ) + -\ V \ \rightarrow$	2
$5 + (\ I\ )\ O\ E \ \rightarrow$	4	$5 + (\ a\ /\ 3\ ) + -\ c$	

s189

Please disregard that we have, in the above derivation, always replaced the leftmost non-terminal. That is of no consequence. The fact that the BNF grammar is ambiguous, that is, allows entirely distinct derivation sequences to lead to the same final string also should be disregarded. It is, at most, perhaps, an unfortunate choice of grammar ! ■

5.2.2 Concrete Type Syntax s190

Definition 29 – **Concrete Type Syntax:** *By a concrete type syntax we shall understand the definition of a set of mathematical structures such as sets, Cartesians, lists, maps and functions.* ■ s191

Example 26 – **A Concrete Type Syntax: Banks:**

49. There are clients, c:C, account numbers a:A and money, m:M.
50. A bank record client accounts and account balances.
51. Client accounts map client names to a finite number of zero, one or more account numbers.

52. Account balances map account numbers into money balances.
53. All client accounts are recorded by the account balances, and the account balances record only accounts listed by one or more clients.

type
49. C, A, Money
50. Bank = Clients $\times$ Accounts
51. Clients = C $\overrightarrow{m}$ A-**set**
52. Accounts = A $\overrightarrow{m}$ Money
axiom
53. $\forall$ (cs,acs):Bank • $\cup$ **rng** cs = **dom** acs

s192 The two sentences of Item 53 (53.1) All client accounts are recorded by the account balances, and (53.2) the account balances record all accounts listed by clients. correspond to:

axiom
53. $\forall$ (cs,acs):Bank •
53.1 $\cup$ **rng** cs $\subseteq$ **dom** acs
53.2 **dom** acs $\subseteq$ $\cup$ **rng** cs

s193 Hence formula line 53.

We then give a concrete type syntax for a bank/client comand language first hinted at in Example 24 on page 40.

54. To the syntactic types we include client identifications, account numbers, money (i.e., cash) and amounts of such.
55. There are open, deposit, withdraw and close commands.
56. Open commands identify the client.
57. Deposit commands identify the client, the account number and the monies to be deposited.
58. Withdraw commands identify the client, the account number and the amount of monies to be withdrawn.
59. Close commands identify the client and the account to be closed.

s194

54. C, A, Money, Amount
55. Command = Open | Deposit | Withdraw | Close
56. Open == mkO(c:C)
57. Deposit == mkD(c:C,a:A,m:Money)
58. Withdraw == mkW(c:C,a:A,amount:Amount)
59. Close == mkC(c:C,a:A)

This example will be followed up by Examples 28 on page 45 and 29 on page 46. ∎

s195

Definition 30 – **Meaning of Concrete Type Syntax:** *We explain both the syntactic* RSL *type definitions, expressions and their meaning. First the syntax.*

60. There are two kinds of type definitions:
    a) simple type definitions which have a left hand side type name and a right hand side type expression (separated by an equal sign: '='),
    b) record type definitions which have a left hand side type name and a right hand side pair of a record constructor name and a parenthesized list of pairs of distinct selector and not necessarily distinct type names (where the left and the right is separated by a double equal sign: '==').
61. Type, record constructor and selector names are identifiers. s196
62. A type expression is either

a) a Boolean (**Bool**) or
b) an integer number (**Intg**) or
c) a natural number (**Nat**) or
d) a real number (**Real**) type name, or is
e) a set (A-**set**) or
f) a Cartesian (A×B×...×C) or
g) a list (A*) or
h) a map (A $\overrightarrow{m}$ B) or
i) a partial (A$\xrightarrow{\sim}$B) or
j) a total function (A→B) type expression, or is
k) a set of (alternative, |) type expressions.

The below only shows how such type definitions and expressions may look like when we (otherwise) write them. That is, the below type definitions and expressions are not type definitions and proper type expressions. s197

Type Definition Examples:
60a. TN = TE
60b. TN == RN(s1:TN1,s2:TN2,...,sn:TNm)
Type Expression Examples:
62. TE =
62a. **Bool**
62b. **Int**
62c. **Nat**
62d. **Real**
62e. TE-**set**
62f. TE1×TE2×...×TEm
62g. TE*
62h. TEi $\overrightarrow{m}$ TEj
62i. TEi $\xrightarrow{\sim}$ TEj
62j. TEi → TEj
62k. TE1 | TE2 | ... | TEm
where: m≥2

In an concrete type syntax of two or more type definitions all left hand side type names are distinct, all type names occurring in right hand side record constructor and type expressions are defined by an abstract or concrete type syntax, and no set (62e.) or function (62h.,62i.) type is defined recursively. Now to the meaning of a concrete type syntax. s198

- The meaning of a type name is the meaning of the right hand side
  - ⋆ type expression TE
  - ⋆ or record constructor expression RN(s1:TN1,s2:TN2,...,sn:TNm).

We shall use a concept of the meanings being "sets" of values. The practicing software engineer may consider these "sets" just as normal set. But, for reasons not explained here, but based in a proper definition of a mathematical semantics for RSL, they are not sets in the usual sense of mathematics. s199

- The meaning of a type expression depends on its form:
  - ⋆ **Bool**: the "set" {**false,true,chaos**};
  - ⋆ **Intg**: the "set" of all integers: {...,-3,-2,-1,0,1,2,3,...};
  - ⋆ **Nat**: the "set" of all natural numbers {0,1,2,3,...};
  - ⋆ **Real**: the "set" of all real number $\{m/n \mid m, n : \mathbf{Nat}, n \neq 0\}$;
  - ⋆ TE-**set**: the "set" of all finite sets of zero, one or more elements, e_i, of TE: $\{\ldots\{e_i, e_j, \ldots, e_k\}\ldots\}$;
  - ⋆ TE1×TE2×...×TEm: the "set" of all Cartesians $\{\ \ldots, (e_{TE1_i}, e_{TE2_i}, \ldots, e_{TEm_n}), \ldots\}$;
  - ⋆ TE*: the "set" of all finite length sequences (or lists) of zero, one or more elements, e_i, of TE: $\{\ldots,\langle e_i, e_j, \ldots, e_k\rangle,\ldots\}$;
  - ⋆ TEi $\overrightarrow{m}$ TEj: the "set" of all finite maps (that is, finite definition set discrete functions) from elements, e_{i_k}, of TEi to elements, e_{j_ℓ}, of TEj: $\{\ldots,[\ldots,e_{i_k}\mapsto e_{j_\ell},\ldots]\ldots\}$; s200
  - ⋆ TEi$\xrightarrow{\sim}$TEj: the "set" of all partial functions from some, but not all elements, e_{i_k}, of TEi, to elements, e_{j_ℓ}, of TEj: $\lambda e_i : \mathsf{TEi} \bullet \mathcal{E}_{\mathsf{TEj}}$;[3]
  - ⋆ TEi→TEj: the "set" of all total functions from elements e_{i_k}, of type TEi, to elements, e_{j_ℓ}, of TEj: $\lambda e_i \bullet \mathcal{E}_j$; and
  - ⋆ TE1|TE2|...|TEm: the "set" which is a "union" of the "sets" denoted by the TEi for i=1,2,...,m.
- The meaning of a record constructor type definition, Tn == RN(s1:TN1, s2:TN2, ..., sn:TNm), is the "set" of all records, { ..., RN(te1,te2,...,tem), ...}, where tei is any value of type TEi for all i.

■ s201

Where the meaning of a BNF grammar is a possibly infinite set of strings over a terminal alphabet, the meaning of a concrete type syntax is a possibly infinite "set" of mathematical values: Booleans, numbers, sets, Cartesians, lists, maps, partial and total functions, where the elements of sets, Cartesians, lists and maps, and where the function argument and results values are any of the values of any of these mathematical values.

The RSL type constructs also allow infinite sets, TE-**infset**, and infinite length lists, TE$^\omega$.

[3] The expression $\lambda e : T_i \bullet \mathcal{E}_{T_j}$ denotes the function which when applied to elements v, of type T_i, yields a value (of type T_j) of expression $\mathcal{E}_{T_j}$ where all free occurrences of e are replaced by v.

5.2.3 Abstract Type Syntax s202

The first abstract syntax proposal was put forward by John McCarthy in [100] where an *analytic abstract syntax* was given for arithmetic expressions the latter in what McCarthy calls a *synthetic* manner. In an *analytic abstract syntax* we postulate, as sorts, a class of terms as a subset of all the "things" that can be analysed. And we associate a number of observer functions with these.

Definition 31 – **Abstract Type Syntax Definition:** *By an abstract type syntax definition we mean a set of one or more sorts, that is, type names, a set of one or more observer, of zero, one or more selector and zero, and one or more constructor ('make') function signatures (function names and argument and result types over these sorts) and a set of axioms which which range over the sorts and defines the observer, selector and constructor ('make') functions.* ∎

s203

Definition 32 – **Abstract Type Syntax:** *By an abstract type syntax we mean a set of sort values of named type with observer, selector and constructor ('make') functions where the sort values and the functions satisfy the axioms.* ∎

s204

Example 27 – **An Abstract Type Syntax: Arithmetic Expressions:** First we treat the notion of analytic grammar, then that of synthetic grammar.

Analytic Grammars: Observers and Selectors For a "small" language of arithmetic expressions we focus just on constants, variables, and infix sum and product terms:

```
type
   A, Term
value
   is_term: A → Bool
   is_const: Term → Bool
   is_var: Term → Bool
   is_sum: Term → Bool
   is_prod: Term → Bool
   s_addend: Term → Term
   s_augend: Term → Term
   s_mplier: Term → Term
   s_mpcand: Term → Term
```

s205

```
axiom
   ∀ t:Term •
      (is_const(t) ∧ ∼ (is_var(t) ∨ is_sum(t) ∨ is_prod(t))) ∧
      (is_var(t) ∧ ∼ (is_const(t) ∨ is_sum(t) ∨ is_prod(t))) ∧
      (is_sum(t) ∧  ∼ (is_const(t) ∨ is_var(t) ∨ is_prod(t))) ∧
      (is_prod(t) ∧ ∼ (isc_const(t) ∨ isv_ar(t) ∨ is_sum(t))),
   ∀ t:A • is_term(t) ≡
      (is_var(t) ∨ is_const(t) ∨ is_sum(t) ∨ is_prod(t)) ∧
      (is_sum(t) ≡ is_term(s_addend(t)) ∧ is_term(s_augend(t))) ∧
      (is_prod(t) ≡ is_term(s_mplier(t)) ∧ is_term(s_mpcand(t)))
```

s206

A is a universe of "things": some are terms, some are not! The terms are restricted, in this example, to constants, variables, two argument sums and two argument products. How a sum is represented one way or another is immaterial to the above. Thus one could think of the following external, written
s207 representations: $a + b$, $+ab$, $(\mathrm{PLUS}\ \ A\ \ B)$, or $7^a \times 11^b$.

Synthetic Grammars: Generators A synthetic abstract syntax introduces generators of sort values, i.e., as here, of terms:

```
value
   mk_sum: Term × Term → Term
   mk_prod: Term × Term → Term
axiom
   ∀ u,v:Term •
      is_sum(mk_sum(u,v)) ∧ is_prod(mk_prod(u,v)) ∧
      s_addend(mk_sum(u,v)) ≡ u ∧  s_augend(mk_sum(u,v)) ≡ v ∧
```

s_mplier(mk_prod(u,v)) ≡ u ∧ s_apcand(mk_prod(u,v)) ≡ v ∧
is_sum(t) ⇒ mk_sum(s_addend(t),s_augend(t)) ≡ t ∧
is_prod(t) ⇒ mk_prod(s_mplier(t),s_mpcand(t)) ≡ t

McCarthy's notion of abstract syntax, both the analytic and the synthetic aspects, are found in most abstraction languages, thus are also in RSL. ■

s208

The previous example illustrated the expression of an abstract type syntax for a *syntactic type* of arithmetic expressions. The next example illustrates the expression of an abstract type syntax for a *semantic type* of banks as well as expression of an abstract type syntax for a *syntactic type* of client commands. The example "pairs" with Example 26 on page 41. s209

Example 28 – **An Abstract Type Syntax: Banks:**
We refer (back) to Example 26 on page 41.

Abstract Syntax of Semantic Types

63. There are banks (BANK) and clients (C), and client have accounts (A) with amounts (Amount) of money (M).
64. From a bank one can observe its set of clients (by their client identifications, C),
65. and its set of accounts (by their account numbers, A).
66. From a bank one can observe the account numbers of a client.
67. For every bank client there is at least one account.
68. From a bank one can observe the money of an account of a client.

s210

type
63 BANK, C, A, Amount, M
value
64 obs_Cs: BANK → C-**set**
65 obs_As: BANK → A-**set**
66 obs_As: BANK × C → A-**set**
  pre obs_As(bank,c): c ∈ obs_Cs(bank)
axiom
  ∀ bank:BANK •
67 ∀ c:C • c ∈ obs_Cs(bank) ⇒ obs_As(bank,c)⊆obs_As(bank)
type
68 obs_M: BANK × C × A → M
  pre obs_M(bank,c,a): c ∈ obs_Cs(bank)∧a ∈ obs_As(bank,c)

s211

Abstract Syntax of Syntactic Types

69. There are bank transaction commands (Command) and these are either open (Open), deposit (Deposit), withdraw (Withdraw) or close (Close) commands.
70. One can observe whether a command is an open, or a deposit, or a withdraw, or a close command.
71. From any command one can observe the identity of the client issuing the command.
72. From other that open commands one can observe the number of the account "against" which the client is directing the transaction.
73. From a deposit command one can observe the cash money being deposited.
74. From a withdraw command one can observe the amount of cash money to be withdrawn.

s212

type
69 cmd:Command, Open, Deposit, Withdraw, Amount, Close
value
70 is_Open, is_Deposit, is_Withdraw, is_Close: Command → **Bool**
71 obs_C: Command → C
72 obs_A: Command $\xrightarrow{\sim}$ A

pre obs_A(cmd): ∼is_Open(cmd)
73 obs_M: Command $\xrightarrow{\sim}$ M
pre obs_M(cmd): is_Deposit(cmd)
74 obs_Amount: Command $\xrightarrow{\sim}$ Amount
pre obs_Amount(cmd): is_Withdraw(cmd)

■

5.2.4 Abstract Versus Concrete Type Syntax s213

Example 29 – **Comparison: Abstract and Concrete Banks:** We refer (back) to Examples 26 on page 41 and 28 on the preceding page. The former presented a concrete type syntax of both semantic and syntactic types related to banking. The latter presented an abstract type syntax of both semantic and syntactic types related to banking. Supposedly the two notion of banks are the same ! We formulate this as follows: The meaning of the model-oriented definition of Example 26 is a model of the meaning of the property-oriented definition of Example 28. The properties expressed by Example 28 are satisfied by the meaning of Example 26.

s214 Usually a property-oriented definition has many, usually an infinite set of models. We now show three "other" model-oriented definitions of the semantic types of banks.

type
BANK_1=(C $\overrightarrow{m}$ A-**set**)×(A $\overrightarrow{m}$ M)

type
BANK_2=A $\overrightarrow{m}$ (C-**set**×M)

type
BANK_3=(C×A×M)-**set**

The first model is that of Example 26 with its invariant as expressed in Item 53 on page 42. The second model requires the following invariant

axiom
∀ bank_2:BANK_2 • ∀ (cs,m):(C-**set**×M) • (cs,m)∈ **rng** bank_2⇒cs≠{}

s215

The third model is like a relational database-oriented model. Each Cartesian (c,a,m) in any bank_3 is like a relation tuple. But two different Cartesians with same account number, (c,a,m), (c′,a,m′) must have same cash balance:

type
BANK_3=(C×A×M)-**set**
axiom
∀ bank_3:BANK_3 •
∀ (c,a,m),(c′,a′,m′):(C×A×M) •
{(c,a,m),(c′,a′,m′)}⊆bank_3 ∧ a=a′ ⇒ m=m′.

For each of the four models: Example 26 and the three above, one can define the observer observer functions of Example 28 and prove its axioms. ■

5.3 Semantics s216

We recall our definition of semantics: semantics is the study of and knowledge about the meaning of words, sentences, and structures of sentences.

We consider two forms of semantics definition styles: denotational and behavioural. Both will be briefly characterised and both will be "amply" exemplified. There are many (other) semantics definition styles: but we shall leave it to other textbooks to fill you in on those, and even our presentation of the two "announced" styles need a deeper treatment than the present software engineering coverage. We refer to [15, Chaps. 19–21] and [16, Chaps. 7 and Chaps. 16-19] for a more thorough software engineering coverage and to [38, 64, 125, 130, 138, 141] for computer and computing science in-depth treatments.

5.3.1 Denotational Semantics s217

Definition 33 – **Denotational Semantics:** *By a denotational semantics we understand a semantics which to simple sentences ascribe a mathematical function and to composite sentences ascribe a semantics which is a homomorphic composition of the meaning of the simpler parts.* ∎ s218

Example 30 – **A Denotational Language Semantics: Banks:** We continue Example 26. We augment the simple sentences of commands with a 'command' which is a list of simple commands:

type
  Command′ = Command | CmdList
  ComdList == mkCL(cl:Command*)
  Response == ok | nok$_d$ |nok$_w$ | nok$_c$ | mkM(m:M)

The meaning of a command is a bank to bank state change and a response value.

value
  $\mathcal{M}$: Command′ → BANK $\stackrel{\sim}{\to}$ BANK × Response

s219
The nok$_d$, nok$_w$, and nok$_c$ responses "signal" the client that command arguments were erroneous.

Opening an account is always possible, but for the other simple commands the client must be known by the bank and the account must be an account of that client. For the withdraw command the amount to be withdrawn must be less than or equal to the account balance. The response value serves to record these conditions for a successful transaction as well as "containing" the "returned" monies in the case of the withdraw and close commands. s220

value
  $\mathcal{M}$: Command′ → BANK $\stackrel{\sim}{\to}$ BANK × Response
  $\mathcal{M}$(cmd)(bank) ≡
    case cmd **of**
      mkO(c) → Open(c)(bank),
      mkD(c,a) → Deposit(c,a)(bank),
      mkW(c,m) → Withdraw(c,a,am)(bank),
      mkC(c,a) → Close(c,a)(bank),
      mkCL(cl) → Compose(cl)(bank)(ok)
    end

s221
Next we define the five auxiliary semantic functions, Open, Deposit, Withdraw, Close and Compose The auxiliary functions Open, Deposit, Withdraw and Closefunction definitions show the denotational principle of ascribing simple functions, in BANK$\stackrel{\sim}{\to}$BANK, to simple commands. The latter, Compose, is defined first. It shows the denotational principle of homomorphic composition. s222

value
  Compose: Command* → BANK $\stackrel{\sim}{\to}$ BANK × Response
  Compose(cl)(bank)(r) ≡
    if cl=⟨⟩
      then (bank,r)
      else let (r′,bank′) = $\mathcal{M}$(**hd** cl)(bank) **in** Compose(**tl** cl)(bank′)(r′) **end**
    end

The homomorphic composition is that of function composition: Compose(**tl** cl) being applied to the bank part, (bank′), of the result of $\mathcal{M}$(**hd** cl)(bank).The "continuation" Response argument, r, of Compose is there to "clean" up, by "removing", the intermediate response results. s223

value
  m$_0$:M
  Open: C → BANK → BANK × Response

$\mathsf{Open(c)(bank)} \equiv$
$\quad \textbf{let } \mathsf{a{:}A} \bullet \mathsf{a} \notin \textbf{dom } \mathsf{acs} \textbf{ in}$
$\quad \textbf{let } \mathsf{cs}' = \textbf{if } \mathsf{c} \notin \textbf{dom } \mathsf{cs} \textbf{ then } [\,\mathsf{c}\mapsto\{\mathsf{a}\}\,] \textbf{ else } [\,\mathsf{c}\mapsto\mathsf{cs(c)}\cup\{\mathsf{a}\}\,] \textbf{ end},$
$\qquad \mathsf{acs}' = \mathsf{acs} \cup [\,\mathsf{a}\mapsto\mathsf{m}_0\,] \textbf{ in}$
$\quad ((\mathsf{cs}',\mathsf{acs}'),\mathsf{ok}) \textbf{ end end}$

We annotate this function definition:

- $\textbf{let } \mathsf{a{:}A} \bullet \mathsf{a} \notin \textbf{dom } \mathsf{acs}$ expresses that a “fresh, hitherto unused” account number is “fetched”;
- $\textbf{let } \mathsf{cs}' = \textbf{if } \mathsf{c} \notin \textbf{dom } \mathsf{cs} \textbf{ then } [\,\mathsf{c}\mapsto\{\mathsf{a}\}\,] \textbf{ else } [\,\mathsf{c}\mapsto\mathsf{cs(c)}\cup\{\mathsf{a}\}\,] \textbf{ end}$, expresses that the new Clients components updates the “input” component cs as follows: if the client is not (yet) known by the bank then the new cs' is the old cs with a new mapping ($\mapsto$) from c to the singleton set $\{\mathsf{a}\}$;
- $\mathsf{acs}' = \mathsf{acs} \cup [\,\mathsf{a}\mapsto\mathsf{m}_0\,]$ expresses that the new Accounts component joins ($\cup$) the mapping ($\mapsto$) from a to a zero valued Money value; and
- $(\mathsf{cs}',\mathsf{acs}')$ expresses the “new” bank.

s224

value
$\quad \mathsf{Deposit{:}\ C \times A \times M \to BANK} \xrightarrow{\sim} \mathsf{BANK \times Response}$
$\quad \mathsf{Deposit(c,a,m)(cs,acs)} \equiv$
$\qquad \textbf{if } \mathsf{c} \in \textbf{dom } \mathsf{cs} \wedge \mathsf{a} \in \mathsf{cs(c)}$
$\qquad\quad \textbf{then } ((\mathsf{cs},\mathsf{acs}\dagger[\,\mathsf{a}\mapsto\mathsf{AddM(acs(a),m)}\,]),\mathsf{ok})$
$\qquad\quad \textbf{else } ((\mathsf{cs},\mathsf{acs}),\mathsf{nok}_d)$
$\qquad \textbf{end}$
$\quad \mathsf{AddM{:}\ M \times M \to M}$

We annotate this function definition:

- The **if** argument $\mathsf{a} \in \mathsf{cs(c)}$ expresses that account a should be an account of client c;
- $(\mathsf{cs},\mathsf{acs}\dagger[\,\mathsf{a}\mapsto\mathsf{AddM(acs(a),m)}\,])$ expresses the “updated” bank:
  - ⋆ the Clients component cs is unchanged;
  - ⋆ the Accounts component acs is updated to reflect that the cash money m has been added, AddM, to the previous balance acs(a) of the a account;
- $((\mathsf{cs},\mathsf{acs}),\mathsf{nok}_d)$ expresses the “not ok” result of an unchanged bank.

s225

value
$\quad \mathsf{Withdraw{:}\ C \times A \times Amount \to BANK} \xrightarrow{\sim} \mathsf{BANK \times Response}$
$\quad \mathsf{Withdraw(c,a,am)(cs,acs)} \equiv$
$\qquad \textbf{if } \mathsf{c} \in \textbf{dom } \mathsf{cs} \wedge \mathsf{a} \in \mathsf{cs(c)} \wedge \mathsf{LessEqM(am,ConvM(acs(a)))}$
$\qquad\quad \textbf{then } ((\mathsf{cs},\mathsf{acs}\dagger[\,\mathsf{a}\mapsto\mathsf{SubM(am,acs(a))}\,]),\mathsf{mkM(ConvM(am))})$
$\qquad\quad \textbf{else } ((\mathsf{cs},\mathsf{acs}),\mathsf{nok}_w)$
$\qquad \textbf{end}$
$\quad \mathsf{SubM{:}\ M \times M} \xrightarrow{\sim} \mathsf{M}$
$\quad \mathsf{ConvM{:}\ (M \to Amount)|(Amount \to M)}$
$\quad \mathsf{LessEqM{:}\ Amount \times Amount} \to \textbf{Bool}$

We annotate this function definition:

- The **if** argument: $\textbf{if } \mathsf{c} \in \textbf{dom } \mathsf{cs} \wedge \mathsf{a} \in \mathsf{cs(c)} \wedge \mathsf{LessEqM(am,ConvM(acs(a)))}$ expresses that the client must be known, that the account must be one of that client and that the amount to be withdrawn must be less than or equal to the in-going account balance;
- the Clients component, cs, of the new bank is unchanged;
- the Accounts component, $\mathsf{acs}\dagger[\,\mathsf{a}\mapsto\mathsf{SubM(am,acs(a))}\,]$ is an update of the previous acs component where account a is updated ($\dagger$) to a balance which is the previous balance, acs(a), minus the withdrawn amount of money (am);

- the ((cs,acs),nok_w) expresses the “not ok” result of an unchanged bank.
- The SubM function subtracts monies.
- The ConvM function converts money into their face value (and vice versa).
- The LessEqM compares two money face values.

s226

value

Close: C $\times$ A $\to$ BANK $\xrightarrow{\sim}$ BANK $\times$ (nok|M)
Close(c,a)(cs,acs) $\equiv$
  if c $\in$ **dom** cs $\wedge$ a $\in$ cs(c)
    then let cs$'$ = cs†[c$'\mapsto$cs(c$'$)\{a}|c$'$:C•c$'$ $\in$ **dom** cs$\wedge$a $\in$ cs(c$'$)],
        acs$'$ = acs\{a} **in**
        ((cs$'$,acs$'$),acs(a)) **end**
    else ((cs,acs),nok_c)
  end

We annotate this function definition:

- The **if** argument c $\in$ **dom** cs $\wedge$ a $\in$ cs(c) expresses that the client must be known (by the bank) and that the account must be one of that client;
- **let** cs$'$ = cs†[c$\mapsto$cs(c)\{a}|c:C•c $\in$ **dom** cs$\wedge$a $\in$ cs(a)] expresses that the update Clients part, cs$'$, of the new bank reflect that all clients, not just the command-issuing client c, that share this account (a) will have their association with that account removed.
- acs$'$ = acs\{a} expresses that the update Accounts part, acs$'$, of the new bank reflect that account a has been removed.
- ((cs$'$,acs$'$),acs(a)) expresses the new bank “value” and that any monies of the closed account are being return to client c;
- ((cs,acs),nok_c) expresses the “not ok” result of an unchanged bank.
- It is only in this function definition that we reveal that accounts may be shared. How such accounts got share we do not reveal — such sharing could be effected by a ‘Share’ command:

  type
    Command$''$ == Command$'$ | mkS{c1:C,a:A,c2:C}.

  The fact that we have not dealt with the issue of who “owns” (and can close an account) and who “co-shares” (and cannot close an account) is of no consequence for our main purpose of this example, namely showing semantics of a client/banking command language.

• • •

s227

The nok_d, nok_w, and nok_c responses could, in a requirements prescriptions be detailed, for example as follows:

- nok_d: `"client or account deposit arguments were wrong"`,
- nok_w: `"client or account deposit arguments were wrong or amount to be withdrawn was too large"`, and
- nok_c: `"client or account deposit arguments were wrong"`;

And, of course, even these more informative “diagnostics” can be sharpened to reflect the conjunction of the **if** predicates. ■

5.3.2 Behavioural Semantics s228

Definition 34 – **Behavioural Semantics:** *By a behavioural semantics we shall here understand a semantics which emphasises concurrency properties of the language being modelled.* ■

s229

Example 31 – **A Behavioural Semantics:** We continue Example 30.

75. There are a number of clients, each is considered a distinct cyclic behaviour
76. indexed by a (well: the) unique Client index.

value
75. client: C ... → **Unit**

s230 The **Unit** designates a "never ending" client behaviour. The . . . will now be "filled in".

77. Each client communicates with one bank (with communication modelled in terms of **channel** input/output (c?, c!cmd)).
78. There is one cyclic bank behaviour.

77. **channel** {ch[c]|c:C} Command|Response
75. client: c:C × CΣ → **out**,**in** {cb[c′]|c′:C\\{c}} **Unit**
78. bank: BANK → **in**,**out** {ch[c]|c:C} **Unit**

s231

79. Client behaviours (over some internal state, cσ [not explained]) alternate
    a) between doing nothing, **skip**, in relation to the bank, and
    b) arbitrarily issuing, based on some property of its local state,
    c) a client/banking command to the bank
    d) and waiting for a response from the bank —
    e) based on which the client updates its local state and continues.
80. We do not detail the predicate over choice of commands and the local client state nor the local client state update.

s232

type
79. CΣ
value
79. client(c,cσ) ≡
79a. (**skip** ; client(c,cσ))
$\sqcap$
79b. (**let** cmd:Command′ • $\mathcal{P}$(cmd,cσ) **in**
79c. ch[c]!cmd;
79d. **let** r = ch[c]? **in**
79e. client(c,client_state_update(cmd,r,cσ)) **end end**)

80. $\mathcal{P}$: Command′ × CΣ → **Bool**
80. client_state_update: Command′ × Response × CΣ → CΣ

s233

81. The bank alternates between serving any of its customers.
82. Sooner or later, if ever a client, c, issues a command, cmd, that command and its origin is received.
83. The command interpretation results in a possibly new bank and a response.
84. The response is communicated to the issuing client.
85. And the bank continues in the possibly new bank state.

value
81. bank(β) ≡
82. **let** (c,cmd) = $\lceil\rceil$ {ch[c]?|c:C} **in**
83. **let** (β',r) = $\mathcal{M}$(cmd)(β) **in**
84. ch[c]!r;
85. bank(β') **end end**

■

5.3.3 Axiomatic Semantics s234

Definition 35 – **Axiomatic Semantics:** *By an axiomatic semantics we understand a pair of abstract type presentations of syntactic and semantic types and a set of axioms which express the meaning of some syntactic values in terms of semantic values.* ■

s235

Example 32 – **An Axiomatic Semantics: Banks:** We continue Example 28. We now give an axiomatics semantics of the simple commands of Example 28. We start by recalling semantic and syntactic types and observer functions. First the semantic types:

```
type
63  BANK, C, A, Amount, M
value
64  obs_Cs: BANK → C-set
65  obs_As: BANK → A-set
66  obs_As: BANK × C → A-set
                    pre obs_As(bank,c): c ∈ obs_Cs(bank)
axiom
    ∀ bank:BANK •
67  ∀ c:C • c ∈ obs_Cs(bank) ⇒ obs_As(bank,c)⊆obs_As(bank)
type
68  obs_M: BANK × C × A → M
                    pre obs_M(bank,c,a): c ∈ obs_Cs(bank)∧a ∈ obs_As(bank,c)
```

s236

Then the syntactic types:

```
type
69  cmd:Command, Open, Deposit, Withdraw, Amount, Close
value
70  is_Open, is_Deposit, is_Withdraw, is_Close: Command → Bool
71  obs_C: Command → C
72  obs_A: Command ~→ A
                    pre obs_A(cmd): ∼is_Open(cmd)
73  obs_M: Command ~→ M
                    pre obs_M(cmd): is_Deposit(cmd)
74  obs_Amount: Command ~→ Amount
                    pre obs_Amount(cmd): is_Withdraw(cmd)
```

s237

The semantic function signatures are:

```
value
   open: Open → BANK → BANK × A
   deposit: Deposit → BANK → BANK × (ok|nok)
   withdraw: Withdraw → BANK → BANK × (mkM(m:M)|nok)
   close: Close → BANK → BANK × (mkM(m:M)|nok)
```

s238

We shall illustrate an axiomatic semantics of just Open commands.

```
value
   m₀:M
axiom
   ∀ bank:Bank
      ∀ op:Open •
         let c = obs_C(op),
             (bank′,a) = open(op)(bank),
             cs = obs_Cs(bank),      cs′ = obs_Cs(bank′),
```

$$
\begin{array}{l}
\quad\quad \mathsf{acs} = \mathsf{obs\_As(bank)}, \quad \mathsf{acs}' = \mathsf{obs\_As(bank')}, \\
\quad\quad \mathsf{cacs} = \textbf{if } \mathsf{c} \in \mathsf{obs\_Cs(bank)} \textbf{ then } \mathsf{obs\_As(bank,c)} \textbf{ else } \{\} \textbf{ end}, \\
\quad\quad \mathsf{cacs}' = \mathsf{obs\_As(bank',c)} \textbf{ in} \\
\mathsf{cs}\backslash\{\mathsf{c}\} = \mathsf{cs}'\backslash\{\mathsf{c}\} \wedge \mathsf{c} \in \mathsf{cs}' \wedge \mathsf{a} \notin \mathsf{acs} \wedge \mathsf{a} \notin \mathsf{cacs}' \wedge \\
\mathsf{acs}' = \mathsf{acs} \cup \{\mathsf{a}\} \wedge \mathsf{cacs}' = \mathsf{cacs} \cup \{\mathsf{a}\} \wedge \mathsf{m}_0 = \mathsf{obs\_M(bank',c,a)} \wedge \\
\forall\, \mathsf{c'}:\mathsf{C} \bullet \mathsf{c}' \in \mathsf{cs}\backslash\{\mathsf{c}\} \Rightarrow \mathsf{obs\_As(bank,c')} = \mathsf{obs\_As(bank',c')} \wedge \\
\quad \forall\, \mathsf{a}:\mathsf{A} \bullet \mathsf{a} \in \mathsf{obs\_As(bank,c')} \Rightarrow \mathsf{obs\_M(bank,c',a)} = \mathsf{obs\_M(bank',c',a)} \\
\textbf{end}
\end{array}
$$

s239 The reader is encouraged to formulate the axiomatic semantics for the Deposit, Withdraw and Close commands. This ends Example 32 ■

5.4 Pragmatics s240

We recall our definition of pragmatics: pragmatics is the study of and knowledge about the use of words, sentences and structures of sentences, and of how contexts affect the meanings of words, sentences, etc.

Recall that we "extended" the notion of sentences and words to include building drawings, city plans, machine drawings, production floor machinery, radio circuit diagrams, railway track layouts, enterprise organisation charts, et cetera, We think of these two or three dimensional
s241 artefacts as designating systems.

Rather than dwelling on how, for example bank clients may use the client/banking language of command, we shall, in our example we therefore emphasise

- mostly the pragmatics of both **what** and **how**
  - ⋆ we choose to domain model (describe) and
  - ⋆ requirements prescribe

  and
- to some extent also the pragmatics of **why** these systems are endowed with certain structurings.

We shall emphasise *"the use of words, sentences and structures of sentences,"* and not say much
s242 about *"how contexts affect the meanings of words, sentences, etc."*

Example 33 – **Pragmatics: Banks:** The pragmatics of **what** we describe of banks is determined by the pedagogics of giving as simple, yet as "convincing" examples of syntactic and semantic types and both denotational (albeit a rather "simplistic example of that) without embellishing the example with too many kinds of banking services (for example, intra-bank account transfers, mortgages, statement
s243 requests, etc.).

The pragmatics of **how** we describe banks is determined by the didactics of covering both concrete type syntaxes and abstract type syntaxes of syntactic types, and covering both denotational and
s244 behavioural semantics definitions.

The pragmatics of **why** we describe banks is determined by our wish to convince the reader that it is not a difficult software engineering task to give easy and realistic domain descriptions of important,
s245 seemingly "large" infrastructure components (such as banks).

The pragmatics related to *"how contexts affect the meanings"* includes that we do not, in Examples 26, 28, 29, and 30–31, describe other financial institutions such as portfolio (wealth and investment) management, insurance companies, credit card companies, brokers, trader, commodity and stock exchanges, let alone include the modelling of several banks. These other institutions and banks form one possible context of our model and hence our model limits the meaning of client/banking commands. Another possible context is provided by the personal diligent or casual or delinquent or sloppy, etc., behaviour of client. The human behaviours are not modelled, but must eventually be modelled (cf. Sect. 7.8's Examples 68 on page 125 and 69 on page 125). ■

s246
Pragmatics is not about empirical aspects of software engineering. The pragmatics that we refer to, in the above definition, is that of staff and users of banks. The pragmatics that we covered in the example is that of the pedagogics and didactics of presenting a methodology for software engineering. s247

The aspects of software engineering that we cover, namely that of domain and requirements engineering, are not empirical sciences, or, more precisely the methodologies of domain and requirements engineering are not based on studies of the behaviour of neither domain nor requirements engineers. The aspects of software engineering that we put forward in this book are based on s248
computing science[4] and computing science, like mathematics, upon which it is based, is not an empirical science.[5] s249

The pragmatics of the kind of domains in the context of the way in which we wish to describe these domains and prescribe requirements for computing systems to serve in these domains is far from studied.

We, but this is only a personal remark and not a scientific conjecture, venture to claim that perhaps one cannot formalise pragmatics, that is, that pragmatics is what cannot be formalised. But this is just a "hunch" !

5.5 Discussion s250

We summarise this chapter on semiotics by first recalling our definition: Semiotics is the study of and knowledge about the structure of all 'sign systems'. In accordance with some practice we have divided our presentation into three parts: syntax, semantics and pragmatics. s251

BNF grammars were first[6] made known (in the late 1950s) in connection with the work on defining the first block structured programming language, `Algol 60` [8]. So BNF grammars were for defining the one-dimensional, i.e., textual layout of programming languages. In Sect. 5.1 we enlarge the scope of syntax to also embody the definition of the structure of 'systems' (that is, domains) such as mentioned there (Page 37). The "language" of systems is the possibly infinite set of utterings that staff and users, i.e., system stake holders express when working with (or in) the system. We exemplified this only briefly and in terms of client/banking commands. We make, s252
in this chapter and in this book, a distinction between using syntax definitions to define syntactic types versus using syntax definitions to define semantic types.

MORE TO COME

We encourage readers to embark on studies of the (albeit informal) pragmatics of domains.

5.6 Exercises

See Items 5–6 (of Appendix B, starting Page 196).

[4] Software engineering is applied computing science.

[5] Although some may reasonably claim that Mathematics is what Mathematicians do, that is not, in our opinion, the same as saying: let us therefore study how all those people who claim they are mathematicians are doing call what they mathematics and let the result of such an empirical study determine what mathematics is!

[6] Dines: Find reference to Don Knuth's "paper" on ancient Indian's knowing of "BNF".

6

A Specification Ontology

s253 acm-aso

The point of philosophy is to start with something so simple
as not to seem worth stating,
and to end with something so paradoxical
that no one will believe it.

`Bertrand Russell`
`The Philosophy of Logical Atomism`
`The Monist`[1]`, The Open Court Publ.Co., Chicago, USA`
`Vol. XXVIII 1918: pp 495-527`
`Vol. XXIX 1919: pp 32-63, 190-222, 345-380`

s254

The basic approach to description (prescription and specification) is to describe algebras. We take a somewhat "novel" approach to this: We describe simple entities and functions (i.e., operations) over these, as for any algebra; and then we focus on behaviours of simple entities as sequences of function invocations and events, where events are the results of usually external function invocations. In addition we describe (prescribe and specify) both informally, by means of precise narratives and formally — here in the `RAISE` specification language `RSL` [15–17, 58, 59, 61]. s255

Example 34 – **Transport Net (II):** In Example 10 nets, hubs and links are examples of simple entities of (Pages 9–11). (Hub and link identifiers are not simple entities, they are entity attributes.) The link insert and delete operations (Pages 11–17) of that example are examples of operations. The situation that a link suddenly "disappears" (a road segment is covered by a mudslide, or a bridge collapses) are examples of events (that can be "mimicked" by the remove link operation). The sequence of many insert, some remove and a few link disappearances form a behaviour. ∎

6.1 Russel's Logical Atomism s256

6.1.1 Metaphysics and Methodology

Russell's **metaphysical** view can be expressed as follows: *the world consists of a plurality of independent existing particulars (phenomena, things, entities, individuals*[2]*) exhibiting qualities and standing in relations.*

Russell's **methodology** for doing philosophy was *follow a process of analysis, whereby one attempts to define or construct more complex notions or vocabularies in terms of simpler ones.* s257

---

[1] See http://www.archive.org/search.php?query=title%3A(the monist) AND creator%3A(Hegeler Institute)

[2] We consider the terms 'particulars', 'phenomena', 'things', 'entities' and 'individuals' to be synonymous.

Russell's idea of **logical atomism** can be expressed as consisting of both the *metaphysics* and the *methodology* as basically outlined above.

We shall later in this chapter take up Russell's line of inquiry.

6.1.2 The Particulars [Phenomena - Things - Entities - Individuals] s258

So which are the particulars, that is, the phenomena that we are to describe? Well, they are the particulars of the domain. How do we describe them? Well, we shall now introduce our description ontology. By 'ontology' is meant

> *the philosophical study of the nature of being, existence or reality in general, as well as of the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology deals with questions concerning what entities exist or can be said to exist, and how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences.*[3]

s259

We can speak both of a domain ontology and a description ontology. First, in this section, we shall cover the notion of description ontology, or, as we shall generalise it, specification ontology. The purpose of having a firm understanding of, hopefully a good specification ontology is to be better able to produce good domain ontologies. Later in following chapters we shall outline how
s260 to construct pleasing domain ontologies.

Our specification (description, prescription) ontology emphasises, as also mentioned in the above indented and *slanted* quote, *the basic categories of being and their relations* and *how such entities can be grouped, related within a hierarchy, and subdivided according to similarities and differences*.

The *basic* description *categories*, that is, the *grouping, hierarchy, subdivision* of means of description are these: (i) simple entities[4], (ii) operations (over entities), (iii) events (involving entities)
s261 and (iv) behaviours.

We should here bring a reasoned argument, of philosophical nature, in order to motivate this subdivision of specification means. Instead we postulate this subdivision and hope that the reader, after having read this chapter, will accept the subdivision.

6.2 Entities s262

We have used and we shall be using the term 'entity' extensively in this book. Other, synonymous terms are 'particular' and 'individual'.

Definition 36 – **Entity:** *By an entity we shall understand a phenomenon or a concept which is either inert (in which case we shall call it a 'simple entity'), or "like" a function, or an event, or a behaviour.* ▪

6.3 Simple Entities and Behaviours s263

We lump two of the description categories: simple entities and behaviours in this section. The reason is that we wish to highlight a duality: simple entities as exhibiting behaviours, and behaviours are evolving around simple entities. In the vernacular one often refers to a phenomenon using a name that both covers that phenomenon as a simple entity and as a behaviour.

[3] http://en.wikipedia.org/wiki/Ontology

[4] We shall consider all four categories of description items as entities, but single out simple entities as a category of its own.

Example: A bank as a simple, in this case composite entity with demand/deposit accounts, mortgage accounts, and clients; and a bank as a behaviour with clients opening and closing accounts, depositing into and withdrawing from accounts, etc., and with events such a interest rate change, attempts at withdrawing below the credit limits, etc. □

Rather than further motivating this duality, let us cover the two notions and show how they relate.

6.3.1 Simple Entities s264

Definition 37 – **Simple Entity:** *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).* ■

s265

Example 35 – **Simple Entities:** (i) From air traffic we illustrate aircraft, terminal control towers, ground control towers, regional control centers and continental control centers as simple entities. (ii) From the financial service industry we illustrate money (cash), securities instruments (like stocks, bonds, a transacted credit card slip, etc.), banks, brokers, traders, stock exchanges, commodities exchanges, bank and mortgage accounts, etc. as simple entities. (iii) From health care we illustrate citizens and potential patients, medical staff, wards, beds, medicine and operating theatres as simple entities. (iv) From railway systems we illustrate train stations and rail tracks, their constituent (linear, switch, crossover, etc.) rail units, trains, train wagons, tickets, passengers, timetables, station and train staff as simple entities. ■

• • • s266

We model simple entities by stating their types.

type
  A, B, C, D, E, F, G. H, J
  aT
  cT = A × B-**set** × C^* × (D $\overrightarrow{m}$ E) × (F→G) × (H$\stackrel{\sim}{\rightarrow}$J)
value
  obs_A: aT → A
  obs_Bs: aT → B-**set**
  obs_Dl: aT → C^*
  obs_mEF: aT → D $\overrightarrow{m}$ E
  obs−tfGH: aT → F→G
  obs_pfJK: aT → H$\stackrel{\sim}{\rightarrow}$J

A, B, C, D, E, F, G, H, J and aT are sorts, that is, abstract types. cT is a (concrete) type definition. It defines values ct:cT to be Cartesians (groupings, records, structures) consisting of six components: an A value, a value consisting of a set of zero, one or more B values, a value consisting of a list of zero, one or more C values, a value which is a finite definition set function, that is, a map from D values to E values, a value which is a total function from F values to G values, and a value which is a partial function from H values to J values. Let us assume that values of type aT contain (at least) six distinguishable components, like those of aT then we can define corresponding observer functions. So for sort, i.e., abstract type, values, one can always define observer functions as appropriate. s267

Our specification language, here RSL, allows us to define Cartesian, set, list, map, partial function and total function types. It also allows us to define sorts and observer functions. These possibilities permit us to model composite entities as follows: unordered collections of sub-entities as sets, ordered collections of sub-entities as lists, finite sets of uniquely "marked" sub-entities as maps, and infinite or indefinite sets of uniquely "marked" sub-entities as functions, partial or total.

• • •

s268

A simple entity has properties[5]. A simple entity is either continuous or is discrete, and then it is
s269 either atomic or composite.

By an attribute we mean a 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 properties.

s270

Example 36 – **Attributes:** A continuous simple entity, like 'oil', may have the following attributes: class: *mineral*, kind: *Brent-crude*, amount: *6 barrels*, price: *45 US $/barrel*.

```
type
   Oil, Barrel, Price
   Class == mineral|organic
   Kind == brent_crude|brent_sweet_light_crude|oseberg|ecofisk|forties
value
   obs_Class: Oil → Class
   obs_Kind: Oil → Kind
   obs_No_of_Barrels: Oil → Nat
   obs_Price: Oil → Price
```

s271

An atomic simple entity, like a 'person', may have the following attributes: gender : *male*, name: *Dines Bjørner*, age: *("oh well, too old anyway")*, height: *178cm*, weight: *("oh well, too much anyway")*.

```
type
   Person, Age, Height
   Gender == female|male
value
   obs_Gender: Person → Gender
   obs_Age: Person → Age
   obs_Height: Person → Height
```

s272

A composite simple 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*.

```
type
   RS, Owner, Name, Owner,
   Country == denmark!norway|sweden|...
   Electrified == no|partly|yes
value
   obs_Country: RS → Country
   obs_Name: RS → Name
   obs_Electrified: RS → Electrified
   obs_Owner: RS → Owner
```

The above informal and formal descriptions are just rough sketches. ■

s273

A simple entity is said to be continuous if it can be arbitrarily decomposed into smaller parts each of which still remain simple continuous entities of the same simple entity kind.

Example 37 – **Continuous Entities:** Examples of continuous entities are: oil, i.e., any fluid, air, i.e., any gas, time period and a measure of fabric. ■

s274
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.

Example 38 – **Discrete Entities:** Examples of discrete entities are: persons, rail units, oil pipes, a group of persons, a railway line (of one or more rail units) and an oil pipeline (of one or more oil pipes, pumps and valves). ■ s275

A simple entity is said to be atomic if it cannot be meaningfully decomposed into parts where these parts have a useful "value" in the context in which the simple entity is viewed and while still remaining an instantiation of that entity. s276

Example 39 – **Atomic Entities:** 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 is its attributes.

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. s277

Example 40 – **Composite Entities (1):** A *railway net* (of a railway system) 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 entities.

```
type
   RS, RN, Line, Station
value
   obs_RN: RS → RN
   obs_Lines: RN → Line-set
   obs_Stations: RN → Station-set
axiom
   ∀ rs:RS,rn:RN • let rn = obs_RN(rs) in
      card obs_Lines(rn)≥2 ∧ card obs_Stations(rn)≥1 end
```

■ s278

Example 41 – **Composite 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.

```
type
   Oil_Industry, Oil_Field, Pipeline_System, Refinery, Distrib_System
value
   obs_Oil_Field: Oil_Industry → Oil_Field-set
   obs_Pipeline_System: Oil_Industry → Pipeline_System-set
   obs_Refineries: Oil_Industry → Refinery-set
   obs_Distrib_Systems: Oil_Industry → Distrib_System-set
axiom
   [ all observed sets are non−empty ]
```

■ s279

Composite simple entities are thus characterisable by their **attributes**, their **sub-entities**, and the **mereology** of how these sub-entities are put together.

[5] We shall refrain from a deeper, more ontological discussion of what is meant by properties [55, 101]. Suffice it here to state that properties are what we can model in terms of types, values (including functions) and axioms.

Definition 38 – **Mereology:** *Mereology is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole.* ■

s280 We shall exemplify the above in the following, abstract example.

Example 42 – **Mereology: Parts and Wholes (1):** We speak of systems as assemblies. From an assembly we can immediately observe a set of parts. Parts are either assemblies or units. For the time being we do not further define what units are.

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

s281 Parts observed from an assembly are said to be immediately embedded in that assembly.

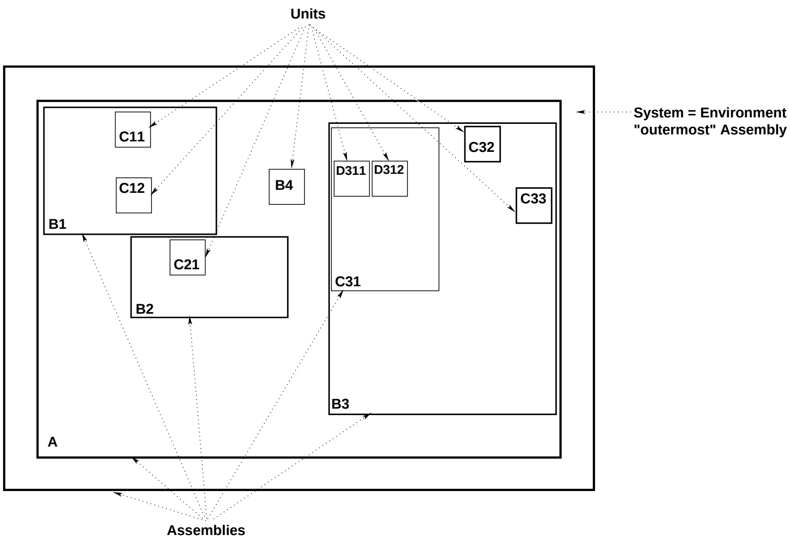


Fig. 6.1. Assemblies and Units "embedded" in an Environment

Figure 6.1 is not our way of modelling a composite simple entity. The formulas above and below is our way of modelling simple entities. Figure 6.1 is just an illustration, a diagrammatic interpretation,
s282 of what the formulas describe. (The same remarks apply also to that of Fig. 6.2 on the facing page.)

For the time being we omit any reference to an environment.

Embeddedness generalises to a transitive relation. All parts thus observable from a system are distinct.

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

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

Parts have unique identifiers. s283

```
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**

We shall now add to this a rather general notion of parts being otherwise related. That notion is one s284
of connectors.

Connectors may, and usually do 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 two parts. Connections "cut across" the "classical" *parts being part of the (or a) whole* and *parts being related by embeddedness or adjacency*. s285

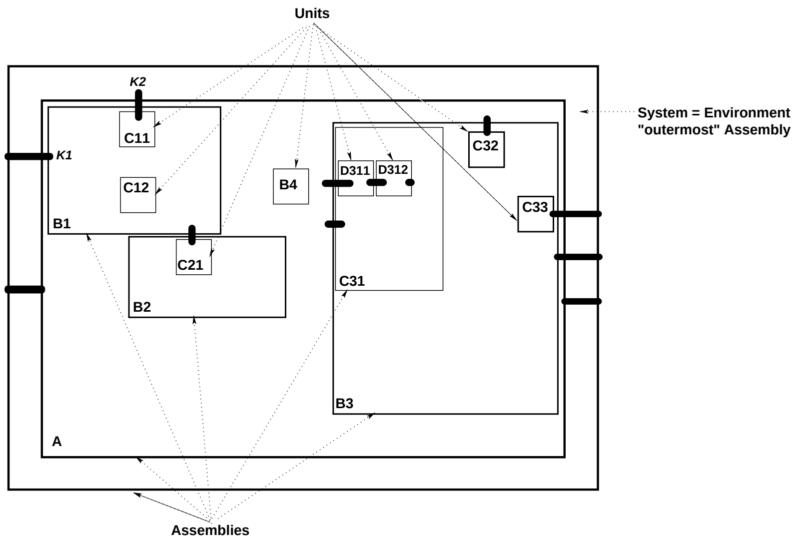


Fig. 6.2. Assembly and Unit Connectors: Internal and External

s286
Figure 6.2 "repeats" Fig. 6.1 on the facing page but "adds" connectors. 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. 6.2 assembly A is connected, by *K2*, (without, as we shall later see, interfering with assembly B1), to part C11; the "external world" is connected, by K1 to B1, etcetera. Thus we make, to begin with, a distinction between internal connectors that connect two identified parts, and external connectors that connect an identified part with an external world. Later we shall discuss more general forms of connectors. s287

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, two if it is an internal connectors, one if it is an external connector. All part identifiers of system connectors identify parts of the system. All observable connector identifiers of parts identify connectors of the system. s288

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 • 1≤**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)

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"
s289 from an outside to any embedded part.

For Example 44 on page 64 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)}

s290

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 and adjacency, and on the other hand, those expressed by connectors: relations between arbitrary parts and between arbitrary
s291 parts and the exterior.

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.

s292

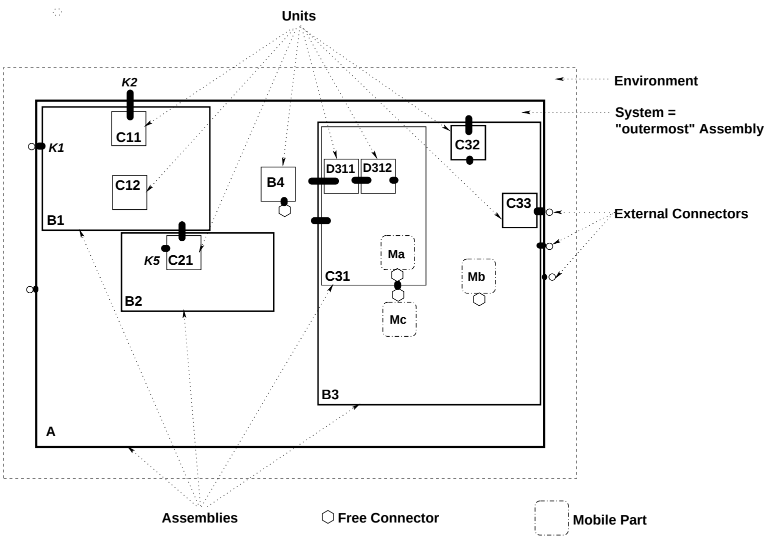


Fig. 6.3. Mobile Parts and Free Connectors

s293

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.

This ends Example 42 ■

s294
Summarising we find, for discrete simple entities, that atomic entities are characterisable by their attributes (as well as by the operations that apply to entity arguments); and that composite entities are characterisable by their attributes, the sub-entities from which they are made up and the mereology, i.e., the part-whole relations between these sub-entities (as well as by the operations that apply to entity arguments). Continuous entities we treat almost as we treat atomic entities except that we can speak of, i.e., define, functions that decompose a continuous entity of kind $\mathcal{C}$ into an arbitrary number of continuous entities of the same kind $\mathcal{C}$, and vice-versa: compose a continuous entity of kind $\mathcal{C}$ from an arbitrary number of continuous entities of the same kind $\mathcal{C}$.

6.3.2 Behaviours s295

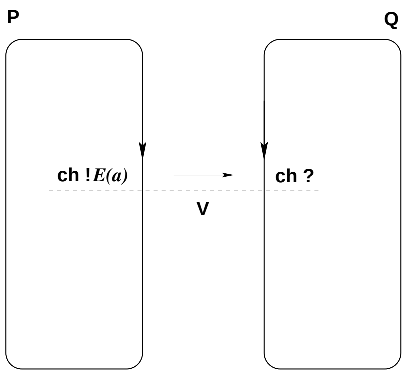
Behaviours can be simple, sequential, and behaviours can be highly composite. To define behaviours we need notions of states and actions: By a domain state we mean any collection of simple entities — so designated by the domain engineer. By a domain action we mean the invocation of an operation which "changes" the state.[6] s296

Definition 39 – **Behaviours:** *(i) By a* behaviour *we shall understand either a simple, sequential behaviour, or a simple parallel (or concurrent) behaviour, or a communicating behaviour. (ii) By a* simple, sequential behaviour *we shall understand a sequence of actions and events (the latter to be defined shortly). (iii) By a* simple parallel (or concurrent) behaviour *we shall understand a set of simple, sequential behaviours. (iv) By a* communicating behaviour *we shall understand a set of simple parallel behaviours which in addition (to being simple parallel behaviours) communicate messages between one-another.* ■

s297
We shall base our formal specification of behaviours on the use of CSP (Hoare's Communicating Sequential Processes [78,79,127,131]). Other concurrency formalisms can, of course, be used: Petri nets [88,112,120–122], Message Sequence Charts (MSCs) [83–85], Statecharts [68–71,73], or other.

Communication, between two behaviours (CSP processes), P and Q is in CSP expressed by the CSP output and input clauses: ch ! e, respectively ch ? where ch designates a CSP `channel`. s298

type
  A, B, M
channel
  ch:M
value
  av:A, bv:B
  $S = P(av) \parallel Q(bv)$
  P: A $\rightarrow$ **out** ch **Unit**
  $P(a) \equiv ...\ ch!\mathcal{E}(a)\ ...\ P(a')$
  Q: B $\rightarrow$ **in** ch **Unit**
  $Q(b) \equiv ...$ **let** $v = ch?$ **in** $...\ Q(b')$ **end**



s299

where a, a′ and b, b′ designate process P, respectively process Q state values provided to process invocations (with a′ and b′ resulting from "calculations" not shown in the bodies of the definitions of P and Q. av and bv are initial entities. S is the overall system behaviour of two communicating behaviours operating in parallel (‖). M designates the type of the messages sent over channel ch.

We shall now show a duality between entities and behaviours. The background for this duality is the following: In everyday parlance we speak of some domain phenomena both as being entities and as embodying behaviours. s300

Example 43 – **Entities and Behaviours:** A train is the composite entity of one or more engines (i.e., locomotives) and one or more passenger and/or freight cars.

[6] Since we shall be expressing our formalisations in a pure, functional language, state changes are expressed by functions whose signature include state entity types both as arguments and as results.

```
type
   Train, Engine, PassCar, FreightCar
   Car = PassCar|FreightCar
value
   obs_Engine: Train → Engine
   obs_CarI: Train → Car*
axiom
   ∀ tr:Train • card obs_CarI(tr)>0
```

s301 A train is also the behaviour whose state include the time dependent train location and the states of these engines and cars and whose sequence of actions comprise the arrival and stop of the train at stations, the unloading and loading of passengers and freight at stations, the start-up and departure of trains from stations and the continuous movement, initially at accelerated speeds, then constant speed, finally at decelerating speeds along the rail track between stations — occasionally allowing for stops at track segment blocking signals. A train behaviour event could be that a cow presence of the
s302 track causes interrupt of scheduled train behaviour. Et cetera.

```
type
   T, Loc,
   TrainBehaviour = T →ₘ Train
channel
   net_channel (is_at_station|Bool)
value
   obs_Loc: Train → Loc
   train: Train → T → out,in net_channel → Unit
   train(tr)(t) ≡
      if is_at_Station(obs_Loc(tr))
         then
            let (tr′,t′) = stop_train(tr)(t);
            let (tr″,t″) = load_unload_passengers_and_freight(tr′)(t′) in
            let (tr‴,t‴) = move_train(tr″)(t″) in
                  [ assert: ∼is_at_Station(obs_Loc(tr‴)) ]
            train(eσ″,(el′,pfl′),loc′)(t′) end end end
         else
            let (tr′,t′) = move_train(tr)(t) in train(tr′)(t′) end
      end
```

s303 Here we leave undefined a number of auxiliary functions:

```
value
   is_at_Station: Loc → out,in net_channel  Bool
   stop_train: Train → T → Train × T
   load_unload_passengers_and_freight: Train → T → Train × T
   move_train: Train → T → Train × T
```

The above “model” of a train behaviour is really not of the kind (of models) that we shall eventually seek. The predicate is_at_Station communicates with the net behaviour (not shown). The function load_unload_passengers_and_freight communicates with the net behaviour (platforms, marshalling yards, etc., not shown). It is a rough sketch meant only to illustrate the process behaviour. ■

s304 Example 42 illustrated a very general class of mereologies. The next example, Example 44 will show how the duality between entities and behaviours can be “drawn” to an ultimate conclusion !

s305

Example 44 – **Mereology: Parts and Wholes (2):** The model of mereology presented in Example 42 on page 60 (Pages 60–62) focused on the following simple entities

- the assemblies,
- the units and
- the connectors.

To assemblies and units we associate CSP processes, and to connectors we associate CSP channels, one-by-one [78, 79, 127, 131]. s306

The connectors form the mereological attributes of the model.

To each connection we associate a CSP channel, it is “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 one unit or one assembly process. s307

From a system which is an assembly, we can extract all the connector identifiers as well as all the internal connector identifiers. They become indexes into an array of channels. Each of the external connector channels is mentioned in exactly one unit or one assembly process; and each of these internal connection channels is a mentioned in exactly two unit or assembly processes. The xtr∀KIs(s) below was defined in Example 42 (Page 62). s308

value
  s:S
  kis:KI-**set** = xtr∀KIs(s)
type
  ChMap = AUI $\underset{m}{\rightarrow}$ 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

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

s309

value
  assembly: a:A→**in**,**out** {ch[cm(i)]|i:KI•i ∈ cm(obs_AUI(a))} **process**
  assembly(a) ≡
    $\mathcal{M}_{\mathcal{A}}$(a)(obs_A$\Sigma$(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→A$\Sigma$→**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

s310

The unit process is defined in terms of the recursive meaning function $\mathcal{M}_{\mathcal{U}}$ function which requires access to all the same channels as the unit process.

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

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

$\mathcal{M}_\mathcal{U}$(u)(uσ) $\equiv$ $\mathcal{M}_\mathcal{U}$(u)($U\mathcal{F}$(u)(uσ))

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

s311 The meaning processes $\mathcal{M}_\mathcal{U}$ and $\mathcal{M}_\mathcal{A}$ are generic. Their sôle purpose is to provide a never ending recursions. "In-between" they "makes use" of assembly, respectively unit specific functions here symbolised s312 by $A\mathcal{F}$ and $U\mathcal{F}$.

The assembly function "first" "functions" as a compiler. The 'compiler' translates an assembly structure into three process expressions: the $\mathcal{M}_\mathcal{A}(a)(a\sigma, a\rho)$ 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 $\mathcal{M}_\mathcal{A}(a)(a\sigma, a\rho)$ and $\mathcal{M}_\mathcal{U}(u)(u\sigma, u\rho)$ process expressions s313 are invoked.

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. This ends Example 44 ∎

s314 The duality between simple entities and behaviours has the attributes of atomic as well as of composite entities become the state in which the entity behaviours evolve.

Whereas — in principle — the mereology of how sub-entities compose into entities are modelled as in Example 42, namely in terms of sorts, observer functions and axioms over unique identifiers of simple entities, their attributes are usually modelled in a more model-oriented way, in terms of mathematical sets, Cartesians, sequences and maps.

6.4 Functions and Events s315

We shall consider events to be special cases of function invocations.

6.4.1 Functions

Definition 40 – **Function:** *By a* function *we shall understand something (a* functional entity*) which when* applied *to an entity, which we shall call an* argument *of the function,* yields *a* result *which is also an entity.* ∎

We shall refer to the application of a function to an argument as an invocation. Functions are characterised by the function signature or just signature and by their function definition. We show s316 a number of function (and process) signatures:

type
  A, B, M1, M2

channel
  chi:MI, cho:MO

The above type definitions and channel declarations are used below:

value
  f0: A $\rightarrow$ B
  f1: **Unit** $\rightarrow$ B
  f2: **Unit** $\rightarrow$ **Unit**
  f3: A $\rightarrow$ **Unit**
  f4: A $\rightarrow$ **in** chi B
  f5: A $\rightarrow$ **in** chi **out** cho B
  f6: A $\rightarrow$ **out** cho B
  f7: A $\rightarrow$ **in** chi **Unit**
  f8: A $\rightarrow$ **in** chi **out** cho **Unit**
  f9: A $\rightarrow$ **out** cho **Unit**

In Example 44 we used the literal **process** where we now, and in future, shall use the more proper RSL term **Unit**. The reason for using **process** in the example was that the term 'unit' was used to name a sort of simple entities. s317

- f0 designates a ("pure, applicative") function from A into B.
- f1 designates a constant function: takes no argument, i.e., invocation is expressed by f1(), but yields a (constant) value in B.
- f2 designates a process (i.e., a process function, one that never terminates). Invocation is expressed by f2(),
- f3 designates a process, accepting arguments in A and otherwise never terminating.
- f4 designates a function, accepting arguments in A and inputs, of type MI, on channel chi and otherwise terminating yielding a value of type B.
- f5 designates a function, accepting arguments in A, and inputs, of type MI, on channel chi, offers outputs, of type MO, on channel cho, and otherwise terminating yielding a value of type B. s318
- f6 designates a function, accepting arguments in A, offers outputs, of type MO, on channel cho, and otherwise terminating yielding a value of type B.
- f7 designates a function, accepting arguments in A, and inputs, of type MI, on channel chi and otherwise never terminating.
- f8 f7 designates a function, accepting arguments in A, inputs, of type MI, on channel chi, offers outputs, of type MO, on channel cho, and otherwise never terminating.
- f9 designates a function, accepting arguments in A, offers outputs, of type MO, on channel cho, and otherwise never terminating.

s319

For functions f4–f9 you may replace A by **Unit** to obtain further signatures. Thus the literal **Unit**, to the left of the → designates that no input is to be provided, and to the right of the → designates a never ending process. s320

We show a number of function signatures exemplifying "Currying":

type
  A, B, C, D
value
  f': A × B × C → D
  f'': A × B → C → D
  f''': A → B → C → D

invocation examples:
  $f'(a,b,c)$
  $f''(a,b)(c)$
  $f'''(a)(b)(c)$

s321

We show two forms of function definitions:

type
  A, B
value
  f: A $\xrightarrow{\sim}$ B
  f(a) **as** b
  pre $\mathcal{P}(a)$
  post $\mathcal{Q}(a,b)$

type
  A, B
value
  g: A → B [A $\xrightarrow{\sim}$ B]
  g(a) ≡ $\mathcal{E}(a)$
  [**pre** $\mathcal{P}(a)$]

s322

f is defined by a pair of **pre/post** conditions expressed by the predicates $\mathcal{P}$(a) and $\mathcal{Q}$(a,b) respectively. The clause '**as** b' expresses that the result is named b g and allows $\mathcal{Q}$ to refer to the result.

g is defined by an explicit ("abstract algorithmic") expression $\mathcal{E}$(a). To avoid cluttering $\mathcal{E}$(a) with basically a test on $\mathcal{P}$(a) (should g not be total on A, that test is brought as a **pre** condition.

6.4.2 Events s323

Definition 41 – **Event:** *By an event we shall generally understand a state change that satisfies a given predicate:*

type
  Σ
value
  $event_i$: $\Sigma \times \Sigma \rightarrow$ **Bool**

Given two states: σ, σ', if $\mathsf{event}_i(\sigma,\sigma')$ *holds then we say that event* event_i *has occurred.* ∎

s324
This definition of an event is much too general. Of course, the domain (or requirements or software design) engineer is the one who decides which events to describe. But we shall accept it on formal grounds. More pragmatically we shall introduces the notions of internal event and external event. Most actions cause events — and they are all internal events. And most of these internal events are (usually) "uninteresting".

s325 A few internal events are interesting, that is, cause state changes "over-and-above" those primarily intended by the action.

Example 45 – **Interesting Internal Events:** Examples of what we would term interesting internal events are: *Banking:* A bank changes its interest rates; *Train Traffic:* a train is cancelled, etc.; *Oil Pipeline:* a pipeline runs dry of oil (due, for example, to valve and pump settings); and *Health Care:* a patient is give a wrong medicine (a form of medical malpractice). ∎

s326
External events are events cause by "functions" beyond "our" control. That is, we postulate that some, maybe we could call it "demonic" function caused an event.

Example 46 – **External Events:** Examples of external events are: *Banking:* a major debitor defaults on a loan; *Train Traffic:* a train runes off the tracks; *Oil Pipeline:* a pipeline bursts; and *Health Care:* a patient dies. ∎

s327
Internal events are typically modelled by providing the usual function, that is, action definitions with a suitable case distinction based on the event_i predicate.

value
  function: A $\rightarrow \Sigma \rightarrow \Sigma \times$ B
  function(a)(σ) $\equiv$
    let (σ',b) = action(a)(σ) **in**
    if event$_i$(σ,σ')
      then cope_with_internal_event(a)(σ,σ')
      else (σ',b) **end end**
  action: A $\rightarrow \Sigma \rightarrow \Sigma \times$ B
  cope_with_internal_event: A $\rightarrow (\Sigma\times\Sigma) \rightarrow \Sigma \times$ B

s328
We may model external events as inputs on channels that can thus be said to "originate" in the environments — but for which functions definitions set aside an alternative choice of accepting such inputs from the environment.

type
  A, B, Σ, Event
channel
  x_ch:Event
value
  function: A $\rightarrow \Sigma \rightarrow$ **in** x_ch Σ B
  function(a)(σ) $\equiv$
    action(a)(σ)

$\lceil\rceil$
let event = x_ch ? **in** cope_with_external_event(a)(event)(σ) **end**
action: A $\rightarrow$ Σ $\rightarrow$ Σ $\times$ B
cope_with_external_event: A $\rightarrow$ Event $\rightarrow$ Σ $\rightarrow$ Σ $\times$ B

6.5 On Descriptions s329

We refer to the subsections of Sect. 2.1, Pages 5–7.

The discussion of this section amounts to establishing a meta-theory of domains, that is, a theory of the abstract, conceptual laws of describing domains in contrast to a theory of any one specific domain, that is, a theory of the concrete, physical and human laws of the described domain.

In our discussion we will rely on understanding specifically referenced examples. This understanding might very well be improved as a result of understanding the message of this section.

6.5.1 What Is It that We Describe ? s330

What is it that our descriptions denote ?[7] The answer is: the "things" that the nouns of our description "points to" are the actual things, "out there", in the domain. The denoted individuals are not "figs of our imagination"[8]. They are 'real', 'actual', can be pointed to, seen, heard, touched, smelled, or otherwise measured by physical, including chemical/physical apparatus(es) !

Therefore there can be no "identical" copies. If two sensed or measured phenomena are "equal" then they are the same phenomenon.

6.5.2 Phenomena Identification s331

We have in various examples, as from Example 10 on page 9 introduced the abstract concept of unique identifiers: of hubs and links (Example 10 on page 9), of parts (assemblies and units) (Example 42 on page 60), and in many later examples. These unique identifications are, in a sense, a mere technicality. We need the unique identifications when we wish to express mereological properties such a *"part of"*, *"next to"*, *"connected to"*, etcetera. Therefore, if two sensed or s332
measured and described phenomena are "equal", except for their postulated unique identification, then they are still the same phenomenon, and there is a problem of description. We next turn to such 'problems of description'.

6.5.3 Problems of Description s333

We can illustrate a number of 'problems of description'. (i) Unique identifications: two (or more) hubs that are claimed to have distinct hub identifiers, cf. Example 10 on page 9, but otherwise have identical values for all conceivable attributes, including spatial location must be the same hub, i.e., have identical hub identifier; (ii) Observability: if from a hub, cf. Example 10 on page 9, we can observe a link, hence all its connected links, and, vice versa, from a link we can observe its connected hubs, and not merely their identifiers, but the 'real' phenomena, then we can argue that we can observe, from any hub the entire net: all hubs and all links, and that is counter to our intuition, we claim, of how we observe.

[7]This question is just another way of expressing the question of the title of this subsection (i.e., Sect. 6.5.1).

[8]I apologize to more philosophically inclined readers: ours is not a discourse on ontology in the philosophical sense: *What may exists ?* etcetera. Our setting is computing science and software engineering — so we have no qualms about postulating that what I can sense, every person in full control of all her senses can sense and in an identical way !

6.5.4 Observability s334

That is: we reject the dogma: *"The Universe in a Single Atom"*[9], that is, that all can be observed from a single "position".

But this rejection begs the issue *"What Do We Mean by Observability ?"*

In the following we shall treat observability of simple entities and their attributes on par, that is, not make a distinction. Later we shall make a distinction between observing simple entities and observing attributes.

Simple Observability s335

The simple part of an answer to this question, and, mind you, it is an answer that is based on our computing science and software engineering viewpoint, concerns that which can be physically observed of any domain phenomenon "itself", that is, of the phenomenon observed in isolation. That part of the answer goes like this: of a physically manifested phenomenon we can observe all that can be physically sensed: seen, heard, smelled, tasted, and touched; as well as measured by physical/chemical apparatus(es).

Not-so-Simple, Simple Entity Observability s336

The not-so-simple part of an answer focuses on simple entities and concerns that which can be physically observed of the "immediate" mereology of the simple entity "itself", that is, of the
s337 parts to which that simple entity is connected. Here the answer is: One can observe immediate phenomenological and conceptual connections, that is, the simple entity parts that are connected to the entity under review, and by reference to their identity — hence the need for the identity concept; and similarly for operations, event and behaviours: which operations directly invoke other operations, directly cause events, and, in general, directly participate in behaviours; which events "trigger" operations, other events, and, in general, directly participate in behaviours; and which behaviours synchronise and/or communicate with other behaviours.

6.5.5 On Denoting s338

Yes, we do know that Bertrand Russel wrote a famous paper with this title [128]. But our intention here is less 'lofty', and, perhaps not ! When, above, we write: *the denoted individuals are not "figs of our imagination"* and *they are 'real', 'actual', can be pointed to, seen, heard, touched, smelled, or otherwise measured by physical, including chemical/physical apparatus(es)*, then it is our intention that we express. We can make that claim as far as the informal narrative description is concerned. But when our description is formalised, then what ? Our formal description language has a semantics. That semantics ascribes to our formalisation some mathematical values, structures. That is, our narrative is of the 'real thing', and our formalisation is a model of
s339 the real thing. So there are two notions of 'denoting' at play here. an informal one: the relation between the narrative description and the physical (incl. human) phenomena and a formal one: the relation between the syntax of our formal description and its semantics, i.e., 'the model'. The two notions relate, but only informally: enumerated lines of the narrative has been "syntactically", that is informally, related to "identically" numbered formula lines, with the informal claim that a numbered narrative line "means" the same as the same-numbered formula line !

[9] Also the title of a book by HH The 14$^{\text{th}}$Dalai Lama: 'The Universe in a Single Atom': Reason and Faith

6.5.6 A Dichotomy s340

Example 42 (Pages 60–63), we now claim, has the narrative denote classes of real phenomena and has the formalisation model the syntax and syntactic well-formedness of a large class of such real phenomena. Later, in Example 44, (Pages 64–66), we now claim, has the (the same) narrative (as in Example 42) indicate conceptual semantic models of with the formalisation explicitly designating classes of such semantics models. So the transition between the two examples, Example 42 s341 and Example 44, signal a reasonably profound "shift" from (informally) designating actual phenomena and (formally) denoting their algebraic structures, to, informally and formally, referring to semantic models in terms of behaviours and states. There is no dichotomy here, just a shift of abstraction.

6.5.7 Suppression of Unique Identification s342

When comparing, for example, two simple entities one is comparing not only their attributes but also, when the entities are composite, their sub-entities. Concerning unique identifiers of simple entities we have this to say: We can decide to either include unique identifiers as an entity attribute, or we can decide that such identifiers form a third kind of observable property of a simple entity the two others being ("other") attributes — as we see fit to define and the possible sub-entities of composite entities. Either way, we need to introduce a meta-linguistic operator[10], say s343

$\mathcal{S}_I$: Simple_observable_entity_value $\rightarrow$ Anonymous_simple_entity_value

The concept of an anonymous value is also meta-linguistic. The anonymous value is basically "the same, i.e., "identical" value as is the simple entity value (from which, through $\mathcal{S}_I$[11], it derives) with the single exception that the simple entity value "possesses" the unique identifier of the observable entity value and the anonymous entity value does not.

6.5.8 Laws of Domain Descriptions s344

Preliminaries

When we wish to distinguish one simple entity phenomenon from another then we say that the two ("the one and the other") are distinct. To be distinct to us means that the two phenomena have distinct, that is, unique identifiers. Being simple entity phenomena, separately observable in the domain, means that their spatial (positional) properties are distinct. That is their anonymous values are distinct. Meta-linguistically, that is, going outside the RSL framework[12], we can "formalise" this: s345

type
  A [A models a type of simple entity phenomena]
  I[13] [I models the type of unique A identifiers]
value
  obs_I: A $\rightarrow$ I
axiom
  $\forall$ a,a$'$:A • obs_I(a)$\neq$obs_I(a$'$) $\Rightarrow$ $\mathcal{S}_I$(a)$\neq\mathcal{S}_I$(a$'$)

s346

[10]The operator $\mathcal{S}_I$ is meta-linguistic with respect to RSL: it is not part of RSL, but applies to RSL values.
[11]The $\mathcal{S}$ stands for "suppress" and the $\mathcal{I}$ for the suppressed unique identifier.
[12]but staying within a proper mathematical framework — once we have understood the mathematical properties of $\mathcal{S}_I$ and proper RSL values and 'anonymous' values (which, by the way, are also RSL values)
[13]We have here emphasized I, the type name of the type of unique A identifiers. Elsewhere in this book we treat types of unique identifiers of different types of observable simple entities as "ordinary" RSL types. Perhaps we should have "singled" such unique identifier type names out with a special font ? Well, we'll leave it as is !

The above applies to any kind of observable simple entity *phenomenon* A. It does not necessarily apply to simple entity *concepts*.

Example: *Two uniquely identified timetables may have their anonymous values be the exact*
s347 *same value.* □

Simple entity phenomena, in our ontology, are closely tied to space/time "co-ordinates" — with no two simple entity phenomena sharing overlapping space. Concepts are, in our ontology, not so constrained, that is, we allow "copies" although uniquely named ! That is, two seemingly distinct concepts may be the same when "stripped" of their unique names !

Some Domain Description Laws s348

We shall just bring a few domain description laws here. Enough, we hope, to spur further research
s349 into 'laws of description'.

Domain Description Law 1 – **Unique Identifiers:** If two observable simple entities have the same unique identifier then they are the same simple entity. ■

Any domain description must satisfy this law. The domain describer must, typically through axioms, secure that the domain description satisfy this law. Thus there is a *proof obligation* to be
s350 *dispensed*, namely that the unique identifier law holds of a domain description.

Domain Description Law 2 – **Unique Phenomena:** If two observable simple entities have different unique identifiers then their values, "stripped" of their unique identifiers are different. ■

Any domain description must satisfy this law. The domain describer must, typically through axioms, secure that the domain description satisfy this law. Thus there is a *proof obligation* to be
s351 *dispensed*, namely that the unique phenomena law holds of a domain description.

Domain Description Law 3 – **Space Phenomena Consistency:** Two otherwise unique, and hence distinctly observable phenomena can, spatially, not overlap. ■

s352

We can express the *Space/Time Phenomena Consistency Law* meta-linguistically, yet in a proper mathematical manner:

type
  E [E is the type name of a class of observable simple entity phenomena]
  $\mathcal{I}$ [$\mathcal{I}$ is the type name of unique E identifiers]
  $\mathcal{L}$ [$\mathcal{L}$ is the type name of E locations]
value
  obs_$\mathcal{I}$: E $\to$ $\mathcal{I}$
  obs_$\mathcal{L}$: E $\to$ $\mathcal{L}$
axiom
  $\forall$ e,e$'$:E • obs_$\mathcal{I}$(e)$\neq$obs_$\mathcal{I}$(e$'$) $\Rightarrow$ obs_$\mathcal{L}$(e) $\sqcap$ obs_$\mathcal{L}$(e$'$) $= \emptyset$

We can assume that this law always holds for otherwise unique, and hence distinctly observable
s353 phenomena.

Domain Description Law 4 – **Space/Time Phenomena Consistency:** If a simple entity (that has the location property), at time t is at location ℓ, and at time t' (larger than t) is at location ℓ' (different from ℓ), then it moves **monotonically** from ℓ to ℓ' during the interval (t, t'). ■

Specialisations of this law are, for example, that if the movement is of two simple entities, like two trains, along a single rail track and in the same direction, then where train s_i is in front of train s_j at time t, train s_j cannot be in front of train s_i at time t' (where $t' - t$ is some small time interval).

Discussion

s354

There are more domain description laws. And there are most likely laws that have yet to be "discovered" ! Any set of laws must be proven consistent. And any domain description must be proven to adhere to these (and "the" other) laws. We decided to bring this selection of laws because they are a part of the emerging 'domain science'. Laws 3 on the preceding page and 4 on the facing page are also mentioned, in some other form, in [129].

Are these domain description laws laws of the domain or of their descriptions, that is, are they domain laws ? We leave the reader to ponder on this !

6.6 Exercises

See Items 7–10 (of Appendix B, starting Page 196).

Part IV

Domains

7

Domain Engineering

s355 acm-tsode-1

7.1 The Core Stages of Domain Engineering

The core stages of domain engineering are those of modelling the following domain facets: intrinsics (Sect. 7.3), support technologies (Sect. 7.4), management and organisation (Sect. 7.5),rules and regulations (Sect. 7.6), scripts (contracts and licenses) (Sect. 7.7) and human behaviour (Sect. 7.8) of the domain. s356

An important stage of domain engineering is that of rough sketching the business processes. This stage is "sandwiched" in-between the opening stages of domain acquisition and domain analysis. s357

The decomposition of these core stages into exactly these facet description stages is one of pragmatics. Experience has shown that this decomposition into modelling stages leads to a suitable base for a final model. That is, the domain engineers may follow, more-or-less strictly the facet stage sequence hinted at above but the domain engineers may, very well, in the end, present the final domain description without clear delineations, in the description, between these facets. In other words, the decomposition and the principles of each individual facet stage, we think, provides a good set of guidelines for the domain engineers on how to proceed. s358

These core stages are preceded by a number of opening stages and succeeded by a number of closing stages.

The opening and closing stages, cf. Sects. 7.9.1 on page 128 and 7.9.2 on page 128, except for the business process sketching stage, are here considered less germane to the proper understanding of the domain concept.

7.2 Business Processes s359

The rough-sketching of business processes shall serve as an "easiest", informal way of starting the more systematic domain acquisition process.

Definition 42 – **Business Process:** *By a business process we understand the procedurally describable aspects, of one or more of the ways in which a business, an enterprise, a factory, etc., conducts its yearly, quarterly, monthly, weekly and daily processes, that is, regularly occurring chores. The business processes may include strategic, tactical and operational management and* s360 *work-flow planning and decision activities; and the administrative, and where applicable, the marketing, the research and development, the production planning and execution, the sales and the service (work-flow) activities — to name some.* ■

7.2.1 General Remarks s361

A domain is often known to its stakeholders by the various actions they play in that domain. That is, the domain is known by the various sequences of entities, functions and events the stakeholders are exposed to, are performing and are influenced by. Such sequences are what we shall here
s362 understand as business processes.

In our ongoing example, that of railway systems, informal examples of business processes are: for a potential passenger to plan, buy tickets for, and undergo a journey. For the driver of the locomotive the sequence of undergoing a briefing of the train journey plan, taking possession of the train, checking some basic properties of that train, negotiating its start, driving it down the line, obeying signals and the plan, and, finally entering the next station, stopping at a platform, and concluding a trip of the train journey — all that constitutes a business process. For a train dispatcher, the monitoring and control of trains and signals during a work shift constitutes a
s363 business process.

Describing domain intrinsics focuses on the very essentials of a domain. It can sometimes be a bit hard for a domain engineer, in collaboration with stakeholders, to decide which are the domain intrinsics. It can often help (the process of identifying the domain intrinsics) if one alternatively, or hand in hand analyses and describes what is known as the business processes. From a description of business processes one can then analyse which parts of such a description designate, i.e., are about or relate to, which facets.

7.2.2 Rough Sketching s364

Initially the domain engineer proceeds by sketching. We use the term *rough sketching*[1] to emphasise that a rough sketch is just a preparatory document. A roughly sketched business process appears easier to make, that is, gets one started more easily. A rough sketch business process does not have to conform to specific principles about what to describe first, whether to first describe phenomena or concepts; whether to first describe discrete facts or continuous; whether to first
s365 describe atomic facts or composite; whether to first describe informally or formally; etcetera.

Principle 1 – **Describing Domain Business Process Facets:** *As part of understanding any (at least human-made) domain it is important to delineate and describe its business processes. Initially that should preferably be done in the form of rough sketches. These rough sketches should — again initially — focus on identifiable simple entities, functions, events and behaviours. Naturally, being business processes, identification of behaviours comes first. Then be prepared to rework these descriptions as the modelling of domain facets starts in earnest.* ■

s366

Roughly sketched business processes help the domain engineer in the more general domain acquisition effort. Domain stakeholders can be asked to sketch the business processes they are part of. The domain engineer, interacting with the domain stakeholder can clarify open points about a sketched business process. And the domain engineer can elicit facts about the domain as inspired by someone else's sketch.

7.2.3 Examples (I) s367

Example 47 – **A Business Plan Business Process:** The board of any company instructs its chief executive officer (CEO) to formulate revised business plans.[2] Briefly, a business plan is a plan for how the company — strategically, tactically and, to some extent, operationally — wishes to conduct its business: what it strives for, product-wise, image-wise, market-share-wise, financially, etc. The CEO develops a business plan in consultation with executive layers of (i.e., with strategic) management. Strategic management (in-between) discusses the plan (which the CEO wishes to submit to the Board) with tactical management, etc. Once generally agreed upon, the CEO submits the plan to the Board. ■

s368

[1] To both say 'rough' and 'sketching' may, perhaps be saying the same thing twice: sketches usually are rough.

[2] A business plan is not the same as a description of the business' processes.

Example 48 – **A Purchase Regulation Business Process:** In our “example company”, purchase of equipment must adhere to the following — roughly sketched — process: Once the need for acquisition of one or more units of a certain equipment, or a related set of equipment, has been identified, the staff most relevant to take responsibility for the use of this equipment issues a purchase inquiry request. The purchase inquiry request is sent to the purchasing department. The purchasing department investigates the market and reports back to the person who issued the request with a purchase inquiry report containing facts about zero, one or more possible equipment choices, their prices, and their purchase (i.e., payment), delivery, service and guarantee conditions. The person who issued the purchase inquiry s369
request may now proceed to issue a purchase request order, attach the purchase inquiry report and send this to the relevant budget controlling manager for acceptance. If purchase is approved then the purchasing department is instructed to issue, to the chosen supplier, a purchase request order. Once the supplier delivers the ordered equipment, the purchasing department inspects the delivery and issues an equipment inspection report. An invoice from the supplier for the above-mentioned equipment is only paid if the equipment inspection report recommends to do so. Otherwise the delivered equipment is returned to the supplier. ■

s370

Example 49 – **A Comprehensive Set of Administrative Business Processes:** The University of California at Irvine (UCI), had their Administrative and Business Services department suggest, as a learning example, the description of a number of business processes. The “learning” had to do, actually, with business process re-engineering (BPR). So we really should bring the below example into Sect. 8.4! We quote from their home Web page [139]: s371

1. *Human Resources:* “Examine the hiring business process of the University, including the applicant process. Special emphasis should be given to simplifying the process, identifying those parts where there is no value added — i.e., where those parts of the process which one considers *simplifying “away”* add no value. Increase speed of response to applicant and units, and reduce process costs while achieving high quality.” s372
2. *Renovation:* “Review the campus’ remodelling and alterations business process, and develop recommendations to improve Facilities Management services to UCI departments for small projects (under $50,000) and minor capital projects (up to $250,000). Special emphasis should be given to simplifying the process; identifying those parts where there is no value added to the customer’s product; to increase speed and flexibility of response; and to reduce process costs while achieving high quality.” s373
3. *Procurement:* “Review the campus procurement business process and develop recommendations/-solutions for process improvement. The redesigned process should provide “hassle-free” purchasing, give a quick response time to the purchaser, be economical in terms of all costs, be reasonably error-free and be compliant with (US) Federal procurement standards.” s374
4. *Travel:* “Study the travel business process from the stage when a staff member identifies the need to travel to the time when reimbursement is received. Analyze and redesign the process through a six step program based on the following business process improvement (BPI) principles: (i) simplify the process, (ii) identify those parts where there is no value added to the customer, increase (iii) speed and (iv) flexibility of response, (v) improve clarity for responsibilities and (vi) reduce process costs while meeting customer expectations from travel services. The redesign should reflectcustomer needs, service, economy of operation and be in compliance with applicable regulations.” s375
5. *Accounts payable:* “Redesign the accounts payable business process to meet the following functional objectives (in addition to BPI measures): Payment for goods and services must assure that vendors receive remittance in a timely manner for all goods and services provided to the University. Significantly improve the operation’s ability to serve campus customers while maintaining financial solvency and adequate internal controls.” s376
6. *Parking:* “Review how parking permits are sold to students, faculty and staff with the intent of omitting unnecessary steps and redundant data collection. The redesigned process should achieve a dramatic reduction in time spent by people standing in line to purchase a permit, and reduce administrative time (and cost) in recording and tracking permit sales.”

Please observe that the above examples illustrate requests for possible business process re-engineering — but that they also give rough-sketch glimpses of underlying business processes. ▪

7.2.4 Methodology s377

Definition 43 – **Business Process Engineering:** *By business process engineering we understand the identification of which business processes should be subject to precise description, describing these and securing their general adoption (acceptance) in the business, and enacting these business process descriptions* ▪

s378

Principle 2 – **Business Processes:** *Human-made universes of discourse*[3] *entail the concept of business processes. The principle of business processes states that the description of business processes is indispensable in any description of a human-made universe of discourse. The principle of business processes also states that describing these is not sufficient: all facets must be described* ▪

s379

Principle 3 – **Describing Domain Business Process Facets:** *As part of understanding any (at least human-made) domain it is important to delineate and describe its business processes. Initially that should preferably be done in the form of rough sketches. These rough sketches should — again initially — focus on identifiable entities, functions, events and behaviours. Naturally, being business processes, identification of behaviours comes first. Then be prepared to rework these descriptions as other facets are being described in depth* ▪

s380

Technique 1 – **Business Processes:** *The basic technique of describing a human-made universe of discourse involves: (i) identification and description of a suitably comprehensive set of behaviours: the behaviours of interest and the environment; (ii) identification and description, for each behaviour, of the entities characteristic of this behaviour; (iii) identification and description, for each entity, of the functions that apply to entities, or from which entities are yielded; (iv) identification and description, for each behaviour, of the events that it shares — either with other specifically identified behaviours of interest, or with a further, abstract, environment* ▪

s381

Tool 1 – **Business Processes:** *Further techniques and the basic tools for describing business processes include: (1)* `RSL/CSP` *definition of processes, where one suitably defines their input/output*
s382 *signatures, associated channel names and types, and their process definition bodies;*[4] *(2) Petri nets;*[5] *(3) message and live sequence charts for the definition of interaction between behaviours;*[6] *(4) statecharts for the definition of highly complex, typically interwoven behaviours;*[7] *and (5) the usual, full complement of* `RSL`*'s type, function value, and axiom constructs and their abstract techniques for modelling entities and functions* ▪

7.2.5 Examples (II) s383

We rough-sketch a number of examples. In each example we start, according to the principles and techniques enunciated above, with identifying behaviours, events, and hence channels and the type of entities communicated over channels, i.e. participating in events. Hence we shall emphasise, in these examples, the behaviour, or process diagrams. We leave it to other examples to present other aspects, so that their totality yields the principles, the techniques and the tools of domain
s384 description.

---

[3] Examples of human-made universes of discourse are: public administration, manufacturing industries (mechanical, chemical, medical, woodworking, etc.), transportation, the financial service industry (banks, insurance companies, securities instrument brokers, traders and exchanges, portfolio management, etc.), agriculture, fisheries, mining, etc.

[4] `RSL/CSP` [78, 79, 127, 131] was covered in detail in Vol. 1, Chap. 21.

[5] Petri Nets [88, 112, 120–122] were covered in detail in Vol. 2, Chap. 12.

[6] Message [83–85] and live sequence charts [37, 72, 92] were covered in detail in Vol. 2, Chap. 13.

[7] Statecharts [68–71, 73] were covered in detail in Vol. 2, Chap. 14.

Example 50 – **Air Traffic Business Processes:** The main business process behaviours of an air traffic system are the following: (i) the aircraft, (ii) the ground control towers, (iii) the terminal control towers, (iv) the area control centres and (v) the continental control centres (Fig. 7.1). We describe s385

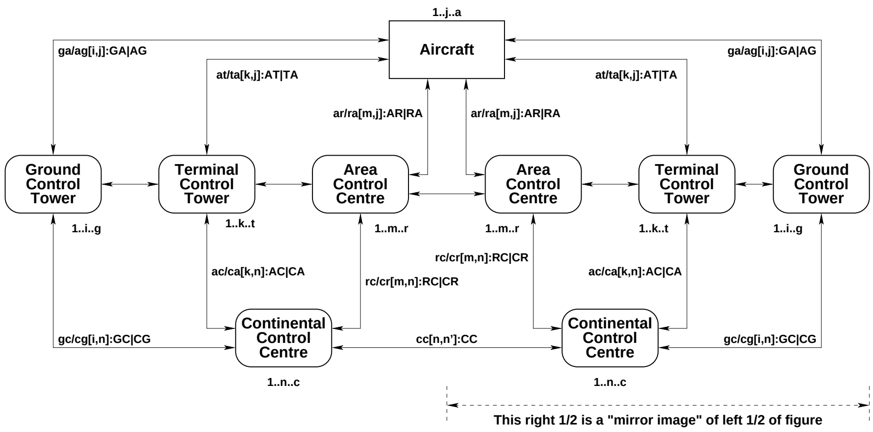


Fig. 7.1. An air traffic behavioural system abstraction

s386
each of these behaviours separately:

(i) *Aircraft* get permission from ground control towers to depart; proceed to fly according to a flight plan (an entity); keep in contact with area control centres along the route, (upon approach) contacting terminal control towers from which they, simplifying, get permission to land; and upon touchdown, changing over from terminal control tower to ground control tower guidance. s387

(ii) The ground control towers, on one hand, take over monitoring and control of landing aircraft from terminal control towers; and, on the other hand, hand over monitoring and control of departing aircraft to area control centres. Ground control towers, on behalf of a requesting aircraft, negotiate with destination ground control tower and (simplifying) with continental control centres when a departing aircraft can actually start in order to satisfy certain "slot" rules and regulations (as one business process). Ground control towers, on behalf of the associated airport, assign gates to landing aircraft, and guide them from the spot of touchdown to that gate, etc. (as another business process). s388

(iii) The terminal control towers play their major rôle in handling aircraft approaching airports with intention to land. They may direct these to temporarily wait in a holding area. They — eventually — guide the aircraft down, usually "stringing" them into an ordered landing queue. In doing this the terminal control towers take over the monitoring and control of landing aircraft from regional control centres, and pass their monitoring and control on to the ground control towers. s389

(iv) The area control centres handle aircraft flying over their territory: taking over their monitoring and control either from ground control towers, or from neighbouring area control centres. Area control centres shall help ensure smooth flight, that aircraft are allotted to appropriate air corridors, if and when needed (as one business process), and are otherwise kept informed of "neighbouring" aircraft and weather conditions en route (other business processes). Area control centres hand over aircraft either to terminal control towers (as yet another business process), or to neighbouring area control centres (as yet another business process). s390

(v) The continental control centres monitor and control, in collaboration with regional and ground control centres, overall traffic in an area comprising several regional control centres (as a major business process), and can thus monitor and control whether contracted (landing) slot allocations and schedules can be honoured, and, if not, reschedule these (landing) slots (as another major business process). s391

From the above rough sketches of behaviours the domain engineer then goes on to describe types of messages (i.e., entities) between behaviours, types of entities specific to the behaviours, and the functions that apply to or yield those entities. ∎

s392

Example 51 – **Freight Logistics Business Processes:**

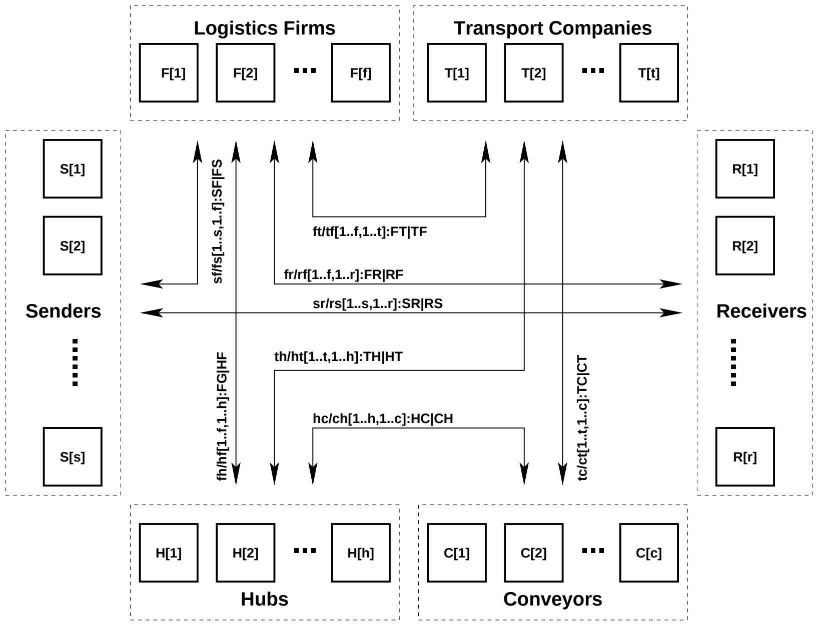


Fig. 7.2. A freight logistics behavioural system abstraction

s393 The main business process behaviours of a freight logistics system are the following: (i) the senders of freight, (ii) the logistics firms which plan and coordinate freight transport, (iii) the transport companies on whose conveyors freight is being transported, (iv) the hubs between which freight conveyors "ply their trade", (v) the conveyors themselves and (vi) the receivers of freight (Fig. 7.2).

A detailed description for each of the freight logistics business process behaviours listed above should now follow. We leave this as an exercise to the reader to complete. ■

s394 **Example 52** – **Harbour Business Processes:** The main business process behaviours of a harbour system are the following: (i) the ships who seek harbour to unload and load cargo at a harbour quay, (ii) the harbour-master who allocates and schedules ships to quays, (iii) the quays at which ships berth and unload and load cargo (to and from a container area) and (iv) the container area which temporarily
s395 stores ("houses") containers (Fig. 7.3 on the next page).

s396 There may be other parts of a harbour: a holding area for ships to wait before being allowed to properly enter the harbour and be berthed at a buoy or a quay, or for ships to rest before proceeding; as well as buoys at which ships may be anchored while unloading and loading. We shall assume that the reader can properly complete an appropriate, realistic harbour domain.

A detailed description for each of the harbour business process behaviours listed above should now follow. We leave this as an exercise to the reader to complete. ■

s397 **Example 53** – **Financial Service Industry Business Processes:** The main business process behaviours of a financial service system are the following: (i) clients, (ii) banks, (iii) securities instrument brokers and traders, (iv) portfolio managers, (v) (the, or a, or several) stock exchange(s), (vi) stock incorporated enterprises and (vii) the financial service industry "watchdog". We rough-sketch the be-
s398 haviour of a number of business processes of the financial service industry.

s399 (i) Clients engage in a number of business processes: (i.1) they open, deposit into, withdraw from, obtain statements about, transfer sums between and close demand/deposit, mortgage and other accounts; (i.2) they request brokers to buy or sell, or to withdraw buy/sell orders for securities instruments

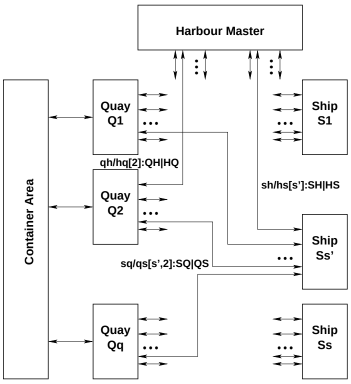


Fig. 7.3. A harbour behavioural system abstraction

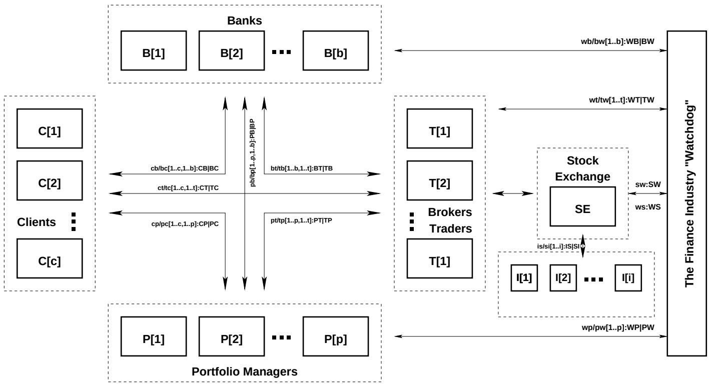


Fig. 7.4. A financial behavioural system abstraction

(bonds, stocks, futures, etc.); and (i.3) they arrange with portfolio managers to look after their bank and securities instrument assets, and occasionally they re-instruct portfolio managers in those respects. s400

(ii) Banks engage with clients, portfolio managers, and brokers and traders in exchanges related to client transactions with banks, portfolio managers, and brokers and traders, as well as with these on their own behalf, as clients.

(iii) Securities instrument brokers and traders engage with clients, portfolio managers and the stock exchange(s) in exchanges related to client transactions with brokers and traders, and, for traders, as well as with the stock exchange(s) on their own behalf, as clients. s401

(iv) Portfolio managers engage with clients, banks, and brokers and traders in exchanges related to client portfolios.

(v) Stock exchanges engage with the financial service industry watchdog, with brokers and traders, and with the stock listed enterprises, reinforcing trading practices, possibly suspending trading of stocks of enterprises, etc.

(vi) Stock incorporated enterprises engage with the stock exchange: They send reports, according to law, of possible major acquisitions, business developments, and quarterly and annual stockholder and other reports.

(vii) The financial industry watchdog engages with banks, portfolio managers, brokers and traders and with the stock exchanges. ∎

s402

Example 54 – **Railway and Train Business Processes:** This example emphasises the simple entities that enable specific business processes. The net of lines and stations, cf. Fig. 7.5 on the following page[A], made up from simple units, cf. Fig. 7.5 on the next page[B], enable train traffic.

And train traffic gives rise to a number of business processes: train journies (say, according to a timetable) [11]; the selling of train tickets including reservation of seats; the controlling of signals such that trains can move in and out of stations and along tracks between stations [13]; track and train
s403 maintenance [117]; staff rostering [137]; et cetera.

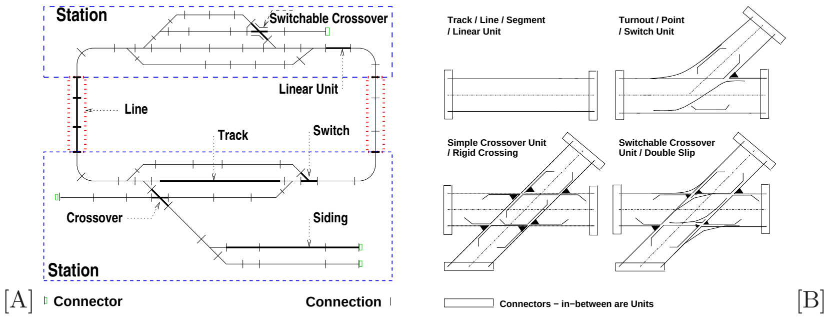


Fig. 7.5. [A] A "model" railway net. An Assembly of four Assemblies:
Two stations and two lines; Lines here consist of linear rail units;
stations of all the kinds of units shown in Fig. 7.5[B].
There were 66 connections at last count and three "dangling" connectors

■

7.2.6 Discussion s404

We shall take up the concept of business processes in Sect. 8 where, in Sect. 8.4 we introduce the important topic of 'business process re-engineering'.

7.3 Domain Intrinsics s405

Definition 44 – **Domain Intrinsics:** *By domain intrinsics we shall understand the very basics upon which a domain is based, the very essence of that domain, the simple entities, operations, events and behaviours without which none of the other facets of the domain can be described.* ■

The choice as to which simple entities, operations, events and behaviours "belong" to intrinsics is a pragmatic choice. It is taken, by the domain engineers, based on those persons' choice of abstraction and modelling techniques and tools. It is a choice that requires quite some experience, quite some years of training, including studying other persons' domain descriptions of similar or
s406 other domains.

Example 55 – **An Oil Pipeline System:**
s407 **Statics of Pipelines**

86. From an oil pipeline system, cf. Fig. 7.6 on the facing page, one can observe units and connectors.
87. Units are either pipe, or (flow, not extraction) pump, or valve, or join or fork units.
88. Units and connectors have unique identifiers.
89. From a connector one can observe the ordered pair of the identity of two (actual or pseudo) from-, respectively to-units that the connector connects.

s408

type
86 OPLS, U, K
value

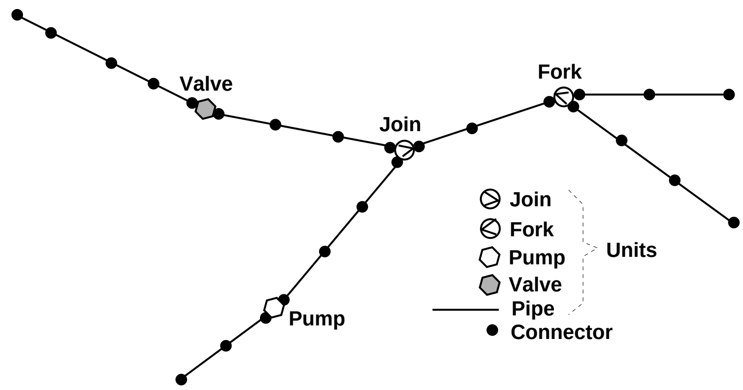


Fig. 7.6. An oil pipeline system with 23 units (19 pipes) and 26 connectors

86 obs_Us: OPLS → U-**set**, obs_Ks: OPLS → K-**set**
87 is_PiU, is_PuU, is_VaU, is_JoU, is_FoU: U → **Bool** [mutually exclusive]
type
88 UI, KI
value
88 obs_UI: U → UI, obs_KI: K → KI
axiom [uniqueness of identifiers]
88 ∀ opls:OPLS,u,u′:U,k,k′:K •
    {u,u′}⊆obs_Us(opls)∧{k,k′}⊆obs_Ks(opls)∧u≠u′∧k≠k′⇒
        obs_UI(u)≠obs_UI(u′)∧obs_KI(u)≠obs_KI(u′)
value
89 obs_UIp: K → (UI|{nil}) × (UI|{nil})

s409

A pseudo unit identity is here modelled by `nil`. `nil` shall indicate that connector from, respectively to-unit does not exist, that is, that the unit is an input, or an output, or both an input and an output of the oil pipeline system.

90. From a unit one can observe the identity of the connectors that provide input to, respectively that provide output from that unit — the two sets of identities are disjoint.
91. From a pipe, pump and valve units we can observe one input and one output connector identifier. From join units we can observe one output and two or more input connector identifiers, and from a fork unit the "reverse": one input and two or more output connector identifiers.
92. Given an oil pipeline system and a connector of that system, the observable ordered pair of actual identities of from- and to-units indeed do identify distinct units of that oil pipeline system.
93. No two connectors connect the same pair of units.

s410

value
90 obs_iKIs, obs_oKIs: U → KI-**set**
axiom
90 ∀ u:U • obs_iKIs(u) ∩ obs_oKIs(u) = {}
91 ∀ u:U •
    is_PiU(u)∨is_VaU(u)∨is_PuU(u) ⇒ **card** obs_iKIs(u)=1=**card** obs_oKIs(u) ∧
    is_JoU(u) ⇒ **card** obs_iKIs(u)≥2 ∧ **card** obs_oKIs(u)=1 ∧
    is_FoU(u) ⇒ **card** obs_oKIs(u)≥2 ∧ **card** obs_iKIs(u)=1
92 ∀ opls:OPLS,k:K: k ∈ obs_Ks(opls) ⇒
    let (fui,tui) = obs_UIp(k) **in**
    fui≠nil ⇒ exist!u:U •
        u ∈ obs_Us(opls)∧fui=obs_UI(u)∧obs_KI(k)∈ obs_oKIs(u) ∧

```
       tui≠nil ⇒ exist!u:U •
             u ∈ obs_Us(opls)∧tui=obs_UI(u)∧obs_KI(k)∈ obs_iKIs(u) end
93  ∀ ols:OPLS,k,k′:K • {k,k′}⊆obs_Ks(opls)∧k≠k′⇒
       let ((fui,tui),(fui′,tui′)) = (obs_UIp(k),obs_UIp(k′)) in
       nil≠fui∧fui=fui′⇒tui≠tui′∧ nil≠tui∧tui=tui′⇒fui≠fui′ end
```

s411

94. An oil pipeline system thus has a set of input units, a set of output units and a set of routes from input to output units.
95. It follows from the above definitions that two two sets are non-empty.

```
value
94  iUs,oUs: OPLS → U-set
94  iUs(opls) ≡
       {u|u:U•u ∈ obs_Us(opls)∧
             let ikis = obs_iKIs(u) in
             ∼∃ u′:U•u′isin obs_Us(opls)∧ikis ∩ obs_oKIs(u′)≠{} end}
94  oUs(opls) ≡
       {u|u:U•u ∈ obs_Us(opls)∧
             let okis = obs_oKIs(u) in
             ∼∃ u′:U•u′isin obs_Us(opls)∧okis ∩ obs_iKIs(u′)≠{} end}
lemma:
95   ∀ opls:OPLS • iUs(opls) ≠ {} ∧ oUs(opls) ≠ {}
```

s412

96. We introduce the concept of a route being a special sequence of units.
97. **Basis Clause:** A unit, u, provides a route, $< u >$, of the oil pipeline system.
98. **Inductive Clause:** If r and r' are routes of the oil pipeline system
    a) and the last unit, u of r, has an output connector identifier
    b) which is an output connector identifier of the first unit, u' of r',
    then their concatenation is a route of the oil pipeline system.
99. **Extremal Clause:** Only such sequences of units are routes if that follows from a finite set of applications of clauses 97 and 98.

s413

```
type
96   R′ = U*
96   R = {| r:R′ • wfR(r) |}
value
96   wfR: R′ → Bool
96   wfR(r) ≡
        case r of
97      ⟨u⟩ → true,
98      r′⁀r″ → wfR(r′) ∧ wfR(r″)
98a-98b  ∧ obs_oKIs(len r′)∩ obs_iKIS(hd r″)≠{}
        end
96  routes: U-set → R-set
96  routes(us) ≡
97    let urs = {⟨u⟩|u:U•u ∈ us} in
97    let rs = urs ∪
98       {r′⁀r″|r′,r″:R•{r′,r″}⊆rs ∧
98a-98b   obs_oKIs(len r′) =  obs_iKIs(hd r″)}
99    rs end end
```

s414

100. An oil pipeline system is well-formed, if — in addition to the earlier mentioned constraints —

a) there is a route from any input unit to some output unit,
b) there is a route leading to any output unit from some input unit and
c) the system of units and connectors "hang together", that is, there is not a partition of these such that the sum of their routes equals the routes of the whole.

s415

axiom
100 ∀ opls:OPLS •
100a ∀ iu:U • iu ∈ obs_iUs(opls) ⇒
100a ∃ ou:U • ou ∈ obs_oUs(opls) ∧
100a ∃ r:R • r ∈ routes(opls) ∧
100a **hd** r = iu ∧ r(**len** r) = ou ∧
100b ∀ ou:U • ou ∈ obs_oUs(opls) ⇒
100b ∃ iu:U • iu ∈ obs_iUs(opls) ∧
100b ∃ r:R • r ∈ routes(opls) ∧
100b **hd** r = iu ∧ r(**len** r) = ou ∧
100c ∼∃ us,us′:U-**set** • us⊂obs_Us(opls) ∧ us′⊂obs_Us(opls)
100c ∧ us ∩ us′ = {} ∧ us ∪ us′ = obs_Us(opls)
100c ⇒ routes(us)∪ rotes(us′)

s416

Dynamics of Pipelines

101. There is oil, $o : O$, and there is oil flow, $f : F$. We do not bother how oil volume is measured, but all oil is measured with the same measuring unit. Oil flow is measured by that measuring unit per some time units (for example, barrels per second).
102. One can observe the oil contained in oil pipeline units.
103. One can observe the oil flowing into and out of connectors of oil pipeline units.
104. Units leak oil.
105. The sum of the oil flowing into a unit minus its leak equals the sum of the oil flowing out of the unit.

s417

type
101 O, F
value
102 obs_O: U → O
103 obs_ioFs: U → (KI $\overrightarrow{m}$ F)×(KI $\overrightarrow{m}$ F)
104 obs_Leak: U → F
axiom
103 ∀ u:U •
  let (ikis,okis) = (obs_iKIs(u),obs_iKIs(u)), (iflow,oflow) = obs_ioFs(u) **in**
  dom iflow = ikis ∧ **dom** oflow = okis **end**
105 ∀ u:U • in_F(u) − obs_Leak(u) = out_F(u)
value
  in_F,out_F: KI $\overrightarrow{m}$ F → F
  in_F(fm),out_F(fm) ≡ **case** fm **of** [ ]→f_0,[ki↦f]∪ fm′→f⊕in_F(fm′) **end**

  ⊕: F × F → F
  f_0:F

f_0 is our way of designating the 'zero' flow, and ⊕ is our way of adding two flows. s418

106. Valve units can be in either of two states: `closed` or `open`.
107. Valves, when `closed`, also leak – in addition to "the usual" leak of units.
108. Pump units can be in either of two states: `pumping` or `not_pumping`.
109. If a valve unit is `closed` then the flows into and out from the unit are characterised by two leak flows.

110. If a pump unit is `not_pumping` then the flows into and out from the unit are characterised to be the same minus the leak of the pump unit.
111. If a pump unit is `pumping` then the flows into and out from the unit a characterised to still be the same minus the leak of the pump unit.

s419

```
value
106      is_open: U → Bool
107      obs_Valve_Leak: U → F
108      is_pumping: U → Bool
axiom
109      ∀ u:U • is_VaU(u) ∧ ∼is_open(u) ⇒
                in_F(u) = obs_Leak(u) ∧ out_F(u) = obs_Valve_Leak(u)
110-111  ∀ u:U • is_PuU(u) ⇒  in_F(u) − obs_Leak(u) = out_F(u)
```

s420

112. One can speak of the total leak of an oil pipeline system.
113. And one can speak of the total flow of oil into and the total flow of oil out from an oil pipeline system.
114. And, consequently one can conjecture a ‘law’ of oil pipeline systems: “what flows in is either lost to leaks or flows out”.

s421

```
value
112  total_Leak: U-set → F
112  total_Leak(us) ≡ case us of {}→f_0,{u}∪ us′→obs_Leak(u) ∪ total_Leak(us′) end
113  total_in_F, total_out_F: OPLS → F
113  total_in_F(opls) ≡ tot_in_F(obs_iUs(opls))
113  total_out_F(opls) ≡ tot_out_F(obs_oUs(opls))
113  tot_io_F: U-set → F
113  tot_in_F(us) ≡ case us of {}→f_0,{u}∪ us′→in_F(u) ∪ tot_in_F(us′) end
113  tot_out_F(us) ≡ case us of {}→f_0,{u}∪ us′→out_F(u) ∪ tot_out_F(us′) end
lemma:
114  ∀ opls:OPLS • total_in_F(opls) − total_Leak(obs_Us(opls) = total_in_F(opls)
```

This ends Example 55 ■

7.3.1 Principles s422

7.3.2 Discussion s423

7.4 Domain Support Technologies s424

acm-tsode-2

Definition 45 – **Domain Support Technology:** *By domain support technology we mean a human or man-made technological device for the support of entities and behaviours, operations and events of the domain — with such a support thus enabling the existence of such phenomena and concepts in the domain.* ■

s425

Example 56 – **Railway Switch Support Technology:** In “ye olde” days a railway switch (point machine [British], turn-out [US English], aguielette [French], sporskifte [Danish], weiche [German]) was operated by a human, a railroad staff member; later, when quality of steel wires and pullers improved, the switch position could be controlled from the station cabin house; further on, in time, such mechanical gear was replaced by electro-mechanical gears, and, most recently, the monitoring and control of groups of switched could be (interlock) done with electronics interfacing to the electro-mechanics. ■

s426 Usually, as hinted at in Example 56, several technologies may co-exist.

Example 57 – **Air Traffic (II):** By air traffic we mean the time and position continuous movement of aircraft in and out of airports and in airspace, that is, for every time point there is a set of aircraft in airspace each with their (not necessarily) distinct positions where an aircraft position is some triple of latitude (ϕ), longitude (λ) and (true, indicated, height, pressure, or density) altitude (above sea level, above the terrain over which aircraft flying, etc.). s427

type
  Time, Aircraft, Position
  cAirTraffic = Time $\rightarrow$ (Aircraft $\overrightarrow{m}$ Position)

How do we know the position of aircraft at any one time ? That is, can we record the continuous movement ? In the above model time is assumed to be a linear, dense point set. But can we record, measure, that ? The answer is: no we cannot ! s428

We, on the ground, can observe with our eyes, with binoculars, and with the aid of some radar (support) technology. The aircraft pilots can record altitude with a pressure altimeter (an aneroid barometer), and `LORAN` or a `Global Navigation Satellite System` (together with an aircraft chronometer) for determination of latitude and longitude.

In any case, whether human or physical instrument-aided observation, one cannot record continuously. Instead any human or instrument awareness of movement is time and position discretised.

type
  Time, Aircraft, Position
  cAirTraffic = Time $\overrightarrow{m}$ (Aircraft $\overrightarrow{m}$ Position)

s429
The difference between continuous and discretised air traffic, that is, between dAirTraffic and cAirTraffic, is the discretisation of Time.

The way we get from cAirTraffic to dAirTraffic is by applying some SupportTechnology:

value
  SupportTechnology: cAirTraffic $\rightarrow$ dAirTraffic

illustration:
  axiom
    $\forall$ cmvnt:cAirTraffic •
      $\exists$ dmvnt:dAirTraffic • dmvnt=SupportTechnology(cmvnt)

This ends Example 57 ■

Example 57 just hints at the concept of 'support technology'. Section 7.4.1 on page 92 enlarges upon the class of support technologies that enable the observation and recording of movement. s430

Example 58 – **Street Intersection Signalling:** In this example of a support technology we shall illustrate an abstraction of the kind of semaphore signalling one encounters at road intersections, that is, hubs. acm-sis

The example is indeed an abstraction: we do not model the actual "machinery" of road sensors, hub-side monitoring & control boxes, and the actuators of the green/yellow/red semaphore lamps. But, eventually, one has to, all of it, as part of domain modelling.

To model signalling we need to model hub and link states. s431

We claim that the concept of hub and link states is an intrinsics facet of transport nets. We now introduce the notions ofhub and link states and state spaces and hub and link state changing operations. A hub (link) state is the set of all traversals that the hub (link) allows.A hub traversal is a triple of identifiers: of the link from where the hub traversal starts, of the hub being traversed, and of the link to where the hub traversal ends. A link traversal is a triple of identifiers: of the hub from where the link traversal starts, of the link being traversed, and of the hub to where the link traversal ends. A hub (link) state space is the set of all states that the hub (link) may be in. A hub (link) state changing operation can be designated by the hub and a possibly new hub state (the link and a possibly new link state). s432

```
type
   LΣ′ = L_Trav-set
   L_Trav = (HI × LI × HI)
   LΣ = {| lnkσ:LΣ′ • syn_wf_LΣ{lnkσ} |}
value
   syn_wf_LΣ: LΣ′ → Bool
   syn_wf_LΣ(lnkσ) ≡
      ∀ (hi′,li,hi″),(hi‴,li′,hi⁗):L_Trav • ⇒
         ({(hi′,li,hi″),(hi‴,li′,hi⁗)} ∈ lnkσ ⇒ li = li′ ∧
         hi′≠hi″ ∧ hi‴≠hi⁗ ∧ {hi′,hi″} = {hi‴,hi⁗})
type
   HΣ′ = H_Trav-set
   H_Trav = (LI × HI × LI)
   HΣ = {| hubσ:HΣ′ • wf_HΣ{hubσ} |}
value
   syn_wf_HΣ: HΣ′ → Bool
   syn_wf_HΣ(hubσ) ≡
      ∀ (li′,hi,li″),(li‴,hi′,li⁗):H_Trav •
         {(li′,hi,li″),(li‴,hi′,li⁗)}⊆hubσ ⇒ hi = hi′
```

s433 The above well-formedness only checks syntactic well-formedness, that is well-formedness when only considering the traversal designator, not when considering the "underlying" net. Semantic well-formedness takes into account that link identifiers designate existing links and that hub identifiers designate existing hub. s434

```
value
   sem_wf_LΣ: LΣ → N → Bool
   sem_wf_HΣ: HΣ → N → Bool
   sem_wf_LΣ(lnkσ)(ls,hs) ≡ lnkσ≠{}⇒
      ∀ (hi,li,hi′): LΣ • (hi,li,hi′) ∈ lnkσ ⇒
         ∃ h,h′:H • {h,h′}⊆hs ∧ obs_HI(h)=hi ∧ obs_HI(h′)=hi′
         ∃ l:L • l ∈ ls ∧ obs_LI(l)=li
      pre syn_wf_LΣ(lnkσ)
   sem_wf_HΣ(hubσ)(ls,hs) ≡ hubσ≠{}⇒
      ∀ (li,hi,li′): HΣ • (li,hi,li′) ∈ hubσ ⇒
         ∃ l,l′:L • {l,l′}⊆ls ∧ obs_LI(l)=li ∧ obs_LI(l′)=li′
         ∃ h:H • h ∈ hs ∧ obs_HI(l)=hi
      pre syn_wf_HΣ(hubσ)
   xtr_LIs: HΣ → LI-set
   xtr_LIs(hubσ) ≡ {li,li′|(li,hi,li′):H_Trav • (li,hi,li′) ∈ hubσ}
   xtr_HI: HΣ → HI
   xtr_HI(hubσ) ≡ let (li,hi,li′):H_Trav • (li,hi,li′) ∈ hubσ in hi end
   pre: hubσ≠{}
   xtr_LI: LΣ → LI
   xtr_LIs(lnkσ) ≡ let (hi,li,hi′):L_Trav • (hi,li,hi′) ∈ hubσ in li end
   pre: lnkσ≠{}
s435
   xtr_HIs: LΣ → HI-set
   xtr_HIs(lnkσ) as his
   pre: lnkσ≠{}
   post his={hi,hi′|(hi,li,hi′):L_Trav • (hi,li,hi′) ∈ hubσ}∧ card his=2
type
   HΩ = HΣ-set, LΩ = LΣ-set
value
   obs_HΩ: H → HΩ, obs_LΩ: L → LΩ
```

axiom
  ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h) ∧ ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
value
  chg_HΣ: H × HΣ → H, chg_LΣ: L × LΣ → L
  chg_HΣ(h,hσ) **as** h′
    pre hσ ∈ obs_HΩ(h) **post** obs_HΣ(h′)=hσ
  chg_LΣ(l,lσ) **as** l′
    pre lσ ∈ obs_LΩ(h) **post** obs_HΣ(l′)=lσ

s436
Well, so far we have indicated that there is an operation that can change hub and link states. But one may debate whether those operations shown are really examples of a support technology. (That is, one could equally well claim that they remain examples of intrinsic facets.) We may accept that and then ask the question: How to effect the described state changing functions ? In a simple street crossing a semaphore does not instantaneously change from red to green in one direction while changing from green to red in the cross direction. Rather there is are intermediate sequences of, for example, not necessarily synchronised green/yellow/red and red/yellow/green states to help avoid vehicle crashes and to prepare vehicle drivers. Our "solution" is to modify the hub state notion. s437

type
  Colour == red | yellow | green
  X = LI×HI×LI×Colour [crossings **of** a hub]
  HΣ = X-**set** [hub states]
value
  obs_HΣ: H → HΣ, xtr_Xs: H → X-**set**
  xtr_Xs(h) ≡
    {(li,hi,li′,c)|li,li′:LI,hi:HI,c:Colour•{li,li′}⊆obs_LIs(h)∧hi=obs_HI(h)}
axiom
  ∀ n:N,h:H • h ∈ obs_Hs(n) ⇒ obs_HΣ(h)⊆xtr_Xs(h) ∧
    ∀ (li1,hi2,li3,c),(li4,hi5,li6,c′):X •
      {(li1,hi2,li3,c),(li4,hi5,li6,c′)}⊆obs_HΣ(h) ∧
      li1=li4 ∧ hi2=hi5 ∧ li3=li6 ⇒ c=c′

s438
We consider the colouring, or any such scheme, an aspect of a support technology facet. There remains, however, a description of how the technology that supports the intermediate sequences of colour changing hub states.

We can think of each hub being provided with a mapping from pairs of "stable" (that is non-yellow coloured) hub states (hσ_i,hσ_f) to well-ordered sequences of intermediate "un-stable' (that is yellow coloured) hub states paired with some time interval information $\langle(h\sigma', t\delta'),\ (h\sigma'', t\delta''),\ \ldots,\ (h\sigma'^{\cdots\prime}, t\delta'^{\cdots\prime})\rangle$ and so that each of these intermediate states can be set, according to the time interval information,[8] before the final hub state (hσ_f) is set. s439

type
  TI [time interval]
  Signalling = (HΣ × TI)*
  Sema = (HΣ × HΣ) $\overrightarrow{m}$ Signalling
value
  obs_Sema: H → Sema,
  chg_HΣ: H × HΣ → H,
  chg_HΣ_Seq: H × HΣ → H
  chg_HΣ(h,hσ) **as** h′
    pre hσ ∈ obs_HΩ(h) **post** obs_HΣ(h′)=hσ
  chg_HΣ_Seq(h,hσ) ≡
    let sigseq = (obs_Sema(h))(obs_Σ(h),hσ) **in** sig_seq(h)(sigseq) **end**
  sig_seq: H → Signalling → H

[8] Hub state $h\sigma''$ is set $t\delta'$ time unites after hub state $h\sigma'$ was set.

```
   sig_seq(h)(sigseq) ≡
      if sigseq=⟨⟩ then h else
      let (hσ,tδ) = hd sigseq in let h′ = chg_HΣ(h,hσ);
      wait tδ;
      sig_seq(h′)(tl sigseq) end end end
```

This ends Example 58 ∎

Example 58 hinted at another class of support technologies, a class whose members illustrate how abstract concepts (of phenomena) are 'implemented'.

7.4.1 A Formal Characterisation of a Class of Support Technologies s440

acm-tsode-2

We have presented an abstraction of the physical phenomenon of a road intersection semaphore. That abstraction has to be further concretised. The electronic, electro-mechanical or other and the data communication monitoring of incoming street traffic and the semaphore control box control
s441 of when to start and end semaphore switching, etcetera, must all be detailed.

Schema 1 – **A Support Technology Evaluation Scheme:** Let the support technology be one for observing and recording the movement of cars along roads, trains along rail tracks, aircraft in airspace (along air-lanes), or "some such thing". We can evaluate the quality of "some such" support technology by interpreting the following specification pattern:

Let *is_close* be a predicate which holds if two positions are close to one-another. Proximity is a
s442 fuzzy notion, so let the *is_close* predicate be "tunable", i.e., by set to "any degree" of closeness.

```
type
   Vehicle, Position
   continuous_Movement = Time → (Vehicle →m Position)
   discrete_Movement = Time →m (Vehicle →m Position)
value
   obs_and_record_Mvmt: continuous_Movement → discrete_Movement
   is_close: Position × Position → Bool
   quality_Support_Technology: is_close → Bool
   quality_Support_Technology(is_close) ≡
      ∀ cmvt:continuous_Movement •
         let dmvt = obs_and_record_Mvmt(cvmt) in
         dom dvmt ⊆ DOMAIN cvmt ∧
         ∀ t:Time • t ∈ dom dvmt
            ∀ v:Vehicle • v ∈ dom dvmt(t) ⇒ is_close(((cvmt)(t))(v),((dvmt)(t))(v)) end
```

s443

The above scheme can be interpreted as follows: For any given sub-domain of movement, be it road traffic, train traffic, air traffic or other, there is a set of technologies that enable observation and recording of such traffic. For a given such technology and a given such traffic, that is, a traffic along a specific route, the predicate *is_close* has to be "instantiated", i.e., "tuned". Then, to test whether the technology delivers an acceptable observation and recording, that is, is of a necessary and sufficient quality, a laboratory experiment — usually quite a resource (equipment, cost and time) consuming affair — has to be carried out before accepting acquisition and installation of that technology for
s444 that route. The experiment ideally compares the actual traffic to that observed and recorded by the contemplated technology. But the actual traffic "does not exist in any recorded form". Hence a "highest possible" movement recording (reference) support technology must first be (experimentally) developed and made available. We then say that whatever that reference technology represents is the actual, but discretised movement. It is that reference movement which is now compared — using *is_close* — to the discretised movement recorded by the support technology being tested. ∎

7.4.2 Discussion s445

For more detailed modelling of specific support technologies, including more concrete models of movement sensors and recorders and of street intersection signals, one will undoubtedly need use other formalisms than the ones mainly used in this paper, for example: Message and Live Sequence Charts, MSCs and LSCs [83–85] and [37,72,92], Petri nets [88,112,120–122], Statecharts [68–71,73], SCs, Duration Calculus [145,146], DC, Temporal Logic of Actions [93,94,102,103], TLA+, Temporal Logic of Reactive Systems [51,97,98,110,114], STeP [27,96], etcetera

7.4.3 Principles s446

7.5 Domain Management and Organisation s447

The term management usually conjures an image of an institution of owners, two or three layers of (hierarchical or matrix) stratified management, workers, and of clients. s448

Management is about resources and resources come in many shapes and forms: manifest equipment, buildings, land, services and/or production goods, financial assets (liquid cash, bonds, stocks, etc.), staff (personnel), customer allegiance, and goodwill[9]. s449

Management decisions as to the monitoring and controlling of resources are often, for pragmatic reasons, classified as strategic, tactical and operational monitor and control decisions and actions.

The borderlines between strategic and tactical, and between tactical and operational monitor and control decisions and actions is set by pragmatic concerns, that is, are hard to characterise precisely. But we shall try anyway. s450

Definition 46 – **Strategy:** *By strategy we shall understand the science and art of formulating the goals of an enterprise and of employing the political, economic, psychological, and institutional resources of that enterprise to achieve those goals.* ■

s451

Definition 47 – **Tactics:** *By tactics we shall understand the art or skill of employing available resources to accomplish strategic goals.* ■

s452

We introduce three kinds of entities to model an essence of strategic, tactical and operational management. Let RES (for resources) designate an indexed set of resources; let ENV (for environment) designate a binding of resource names to resource locations and some of their more static properties — such a schedules, and let Σ (for state) designate the association of resource locations to the more dynamic properties (attributes) of resources, then we might be able to delineate the three major kinds of actions: s453

type
  A, B, C, RES, ENV, Σ
value
  strategic_action: A $\rightarrow$ RES $\rightarrow$ ENV $\rightarrow$ Σ $\rightarrow$ RES
  tactical_action: B $\rightarrow$ RES $\rightarrow$ ENV $\rightarrow$ Σ $\rightarrow$ ENV
  operational_action: C $\rightarrow$ RES $\rightarrow$ ENV $\rightarrow$ Σ $\rightarrow$ Σ

A, B and C are "inputs" chosen by management to reflect strategic or tactical decisions.

Sometimes tactical actions also change the state:

type
  tactical_action: B $\rightarrow$ RES $\rightarrow$ ENV $\rightarrow$ Σ $\rightarrow$ $\Sigma \times$ ENV

s454

A strategic action, strategic_action(a)(res)(ρ)(σ) **as** res′, in principle does not change the environment and state but sets up a new set of resources, res′, for which "future" business is transacted.

[9] Goodwill: the favor or advantage that a business has acquired especially through its brands and its good reputation

A tactical action, tactical_action(a)(res)(ρ)(σ) **as** ρ', changes the environment — typically the scheduling and allocation components of environments.

Operational actions, operational_action(a)(res)(ρ)(σ) **as** σ', changes the state.

s455 The above strategy/tactics/operations "abstraction" is an idealised "story".

Definition 48 – **Resource Monitoring:** *By the monitoring of resources we mean the regular keeping track of these resources: their current value, state-of-quality, location, usage, etc. — including changes in these (i.e., trends).* ■

s456

Definition 49 – **Resource Control:** *By the controlling of resources we mean the acquisition (usually as the result of converting one resources into another), regular scheduling and allocation and final disposal (sale, renewal or "letting go") of these resources.* ■

s457

Definition 50 – **Management:** *By management we mean the strategic, tactical and operational monitoring and controlling of resources .* ■

Definition 51 – **Organisation:** *By organisation we mean the stratification (arranging into graded classes) of management and enterprise actions.* ■

s458
acm-mao

Example 59 – **Management and Organisation:** We continue Examples 42 (Pages 60–62) and 44 (Pages 64–66).

We can claim that the set of models of the description given in Example 42 includes that of
s459 enterprise management and organisation. We refer to Fig. 7.7.

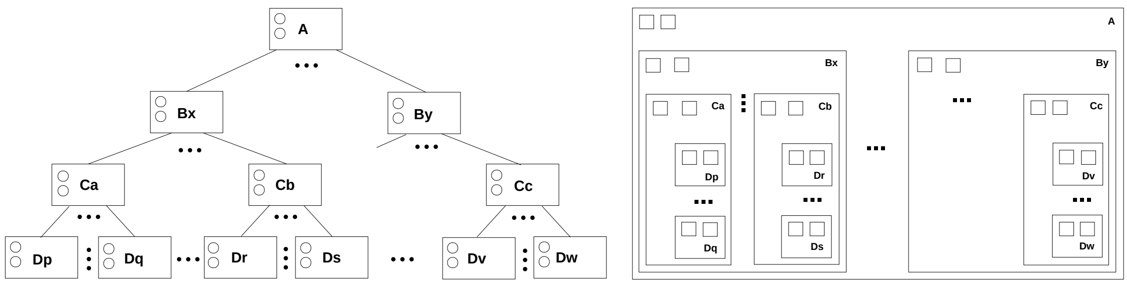


Fig. 7.7. Conventional hierarchical organigram and its mereology diagram

The small, quadratic round-corner boxes of Fig. 7.7 can be thought of as designating staff or other
s460 (atomic) resources.

We will now define a number of strategic/tactical operations on the organisation of an enterprise. For simplicity, bit without any loss of generality, we assume a notion of void parts, that is parts with no connections and, if assemblies, then with no sub-parts. The operations to be defined can be considered 'primitive' only in the sense that more realistic operations on non-void parts can be defined in terms of these primitive operations.

Given this interpretation we can now postulate a number of management operations (over a given
s461 system s).

115. *Assign* a new, void resource p, to a given assembly (i.e., division or department) identified by i.
116. *Move* a given, void resource identified by i, from an assembly identified by f_i to another assembly identified by t_j.
117. *Delete* a given, void resource identified by i.

We ignore, for the time, the issue of connectors. In order to model these operations we need first
s462 introduce some concepts:

118. Given a system, s, and a part, p, of that system,
119. the sequence $\langle\pi_1, \pi_2, \ldots, \pi_{n-1}, \pi\rangle$
120. is the sequence of part identifiers such that π_1 is that of the assembly that s is,
121. that is, is the 1st level part that embraces p, π_n
122. is the identifier of p, and $\imath_i$, for $1<i<n$,
123. is the i'th level part embracing p.

s463

```
value
   void_P: P → Bool
   void_P(p) ≡ obs_KIs(p)={} ∧ is_A(p) ⇒ obs_Ps(p)={}

type
   Path = AUI*
value
   gen_Path: P → A ~→ Path
   gen_Path(p)(a) ≡
      ⟨obs_AUI(a)⟩^
      if p=a then ⟨⟩
      else let a′:A • a′ ∈ obs_Ps(a)∧p ∈ xtr_Ps(a′) in gen_Path(p)(a′) end end
      pre: p ∈ {a} ∪ xtr_Ps(a)
   gen_all_Paths: A → Path-set
   gen_all_Paths(a) ≡ {gen_Path(p)(a)|p:P•p ∈ xtr_Ps(a)}
```

s464

124. *Assigning* a new, void part, p, to a system, s, results in a new system, s'. p is in this new system. Let the path to p be $\pi\ell$. Let the set of all paths of s be $pths$. Then the set of all paths of s' is $pths \cup \{\pi\ell\}$. Thus it follows that the set, ps', of all parts of s', is p together with the set, ps, of all parts of s : $ps' = ps \cup \{p\}$.
125. *Moving* a given, void part, p, of a system, s, results in a new system s'. Let the path to p in s be $\pi\ell$, and let the path to p in s' be $\pi\ell'$. Then the set of paths of the two systems relate as follows: $pths \setminus \{\pi\ell\} = pths' \setminus \{\pi\ell'\}$ and $ps' = ps \cup \{p\}$.
126. *Deleting* a given, void part, p, from a system, s, results in a new system, s'. The new system has exactly one less path than the set of all paths of s. And we have: $pths \setminus \{\pi\ell\} = pths'$ and $ps' = ps \setminus \{p\}$.

s465

```
semantic types
   S, A, U, P=A|U
value
   get_P: S → (AI|UI) ~→ P
   get_P(s)(i) ≡ let p:P • p ∈ xtr_Ps(s)∧obs_AUI(p)=i in p end
      pre ∃ p:P • p ∈ xtr_Ps(s)∧obs_AUI(p)=i

syntactic types
   MgtOp = AP | MP | DP | MA | CA
   AP == AsgP(pt:P,ai:AI)
   MP == MovP(ai:AI,fai:AI,tai:AI)
   DP == DelP(ai:AI)
```

s466

```
value
   int_MgtOp: MgtOp ~→ S ~→ S

   int_MgtOp(AsgP(p,i))(s) as s′
      pre   void_P(p) ∧ obs_AUI(p)∉ xtr_AUIs(s)
```

```
    post obs_Ps(s) ∪ {u} = obs_Ps(s′) ∧ obs_Ks(s)=obs_Ks(s′) ∧
          gen_all_Paths(s)∪{gen_Path(p)(s′)}=gen_all_Paths(s′)

int_MgtOp(MovP(i,fi,ti))(s) as s′
    pre void_P(get_P(s)(i)) ∧ i≠fi∧i≠ti∧fi≠ti∧{i,fi,ti}⊆xtr_AUIs(s)
    post obs_Ps(s) = obs_Ps(s′) ∧ obs_Ks(s)=obs_Ks(s′) ∧
          gen_all_Paths(s)\{gen_Path(p)(s)} = gen_all_Paths(s′)\{gen_Path(p)(s′)}

int_MgtOp(DelP(i))(s) as s′
    pre void_P(get_P(s)(i)) ∧ i ∈ xtr_AUIs(s)
    post obs_Ps(s′) = obs_Ps(s)\{get_P(s)(i)} ∧ obs_Ks(s)=obs_Ks(s′) ∧
          gen_all_Paths(s′)=gen_all_Paths(s)\{gen_Path(p)(s)}
```

s467 Similar connector operations can be postulated (narrated and formalised):

127. *Insert* a new (internal or external) connector, k, in a system s between parts i and j, or just emanating from (incident upon) part i;
128. *Move* a given connector's connections from parts $\{i,j\}$ to parts $\{i,k\}$, $\{\ell,k\}$ or $\{k,j\}$; and
129. *Delete* a given connector.

These operations would have to suitably update connected parts' connector identifier attributes.

The hierarchical organigram of Fig. 7.7 on page 94 portrays one organisation form. So-called matrix-organisations, cf. Fig. 7.8 are likewise modelled by the mereology concept introduced in Examples 42
s468 and 44.

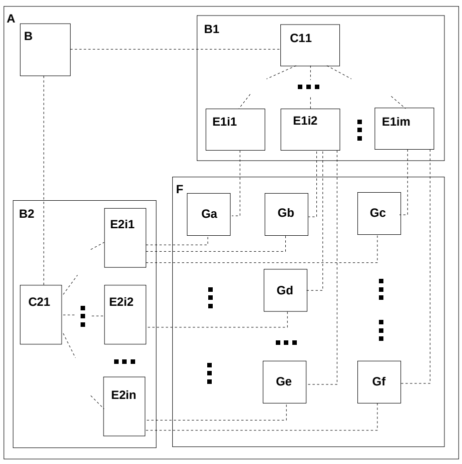


Fig. 7.8. Conventional matrix organigram; mereology diagram is hinted at.

We see, in Fig. 7.8, the use of connectors to underscore the two hierarchies: the strategic and tactical (*B1* and *B2*) and the matrix-sharing of production and service facilities (*F, Ga, ..., Gf*).

This ends Example 59 ■

7.5.1 Principles s469

acm-tsode-2

7.5.2 Discussion s470

7.6 Domain Rules and Regulations s471

Definition 52 – **Domain Rule:** *By a domain rule we understand a text which prescribes how humans and/or technology are expected to behave, respectively function. A domain rule text thus* *denotes a predicate over states, the state before, σ_β, and the state after, σ_α, a human or a technology action. If the predicate is satisfied, then the rule has been adhered to, i.e., the rule has not been "broken". The 'after' state, σ_α, following a rule that has been broken in some 'before' state will be referred to as a 'rule-braking state'.* ■

s472

Definition 53 – **Domain Regulation:** *By a domain regulation we understand a text which prescribes [remedial] actions to be taken in case a domain rule has been "broken". A domain regulation text thus denotes an action, i.e., a state-to-state transformation, one that transforms a 'rule-braking state' σ_α into a (new 'after') state, σ_{OK}, in which the rule now appears to not have been broken.* ■

s473

Example 60 – **Trains Entering and/or Leaving Stations:** For some train stations there is the rule that no two trains may enter and/or leave that station within any (sliding "window") n minute interval — where n typically is 2. If train engine men disregard this rule they may be subject to disciplinary action — as determined by some subsequent audit — and the train may be otherwise diverted through actions from the train station cabin tower. ■

s474

Example 61 – **Rail Track Train Blocking:** Usually rail tracks, that is, longer sequences of linear rail units connecting two train stations are composed of two or more blocks (also sequences of linear rail units). The train blocking rule for trains moving along such rail tracks (obviously in the same direction) is that there must always be an empty block between any two 'neighbouring' trains. (We may consider the connecting stations to serve the rôle of such blocks.) Again, if the rule is broken by some train engine man, then that person may be subject to disciplinary action — as determined by some subsequent audit — et cetera. ■

7.6.1 A Formal Characterisation of Rules and Regulations s475

Schema 2 – **A Rules and Regulations Specification Pattern:**

Let Σ designate the state space of a domain; let Rule designate the syntax category of rules; let RULE designate the semantic type of rules, that is, the denotation of Rules: predicates over pairs of (before and after) states; let Stimulus designate the syntax category of stimuli that cause actions, hence state changes, that is, let STIMULUS finally designate the semantic type of Stimuli. valid_stimulus is now a predicate which "tests" whether a given stimulus and a given rule in a given state, σ, leads to a not-been-broken state. s476

type
  Rule, Stimulus, Σ
  RULE = $\Sigma \times \Sigma \to$ **Bool**
  STIMULUS = $\Sigma \to \Sigma$
value
  $\mathcal{M}_{rule}$: Rule → RULE
  $\mathcal{M}_{stimulus}$: Stimulus → STIMULUS
  $\mathcal{V}$alid_stimulus: Stimulus → Rule → Σ → **Bool**
  $\mathcal{V}$alid_stimulus(stimulus)(rule)(σ) ≡
    (($\mathcal{M}_{rule}$)(rule))(σ,($\mathcal{M}_{stimulus}$(stimulus))(σ))

s477

Let Rule_and_Regulation designate the syntax category of pairs of rules and related regulations; let Regulation designate the semantic type of regulations, that is, the denotation of Regulations: state transformers from broken to OK states. s478

type
  Regulation
  Rule_and_Regulation = Rule × Regulation
  REGULATION = $\Sigma \to \Sigma$

value
  $\mathcal{M}_{regulation}$: Regulation $\rightarrow$ REGULATION
axiom
  $\forall$ (rule_syntax,regulation_syntax):Rule_and_Regulation, stimulus:Stimulus, σ:Σ •
    $\sim\mathcal{V}$alid_stimulus(stimulus)(rule_syntax)(σ) $\Rightarrow$
      $\exists$ σ':Σ •
        ($\mathcal{M}_{regulation}$(regulation_syntax))(σ)=σ'
        $\wedge$ ($\mathcal{M}_{rule}$(rule_syntax))(σ,σ')

■

The formal characterisation expresses, in its last lines, that for every rule that may be broken there must be a regulation which "brings" the enterprise "back-on-track", back to an acceptable state in which the rule is no longer broken.

7.6.2 Principles s479

Rules and regulations are best treated by separately describinging their pragmatics, their semantics, and their syntax — the latter two were hinted at in Sect. 7.6.1.

7.6.3 Discussion s480

Many more examples could be given, and also formalised. We leave that to the next section, Sect. 7.7.

7.7 Domain Scripts, Licenses and Contracts s481

acm-tsode-3

Definition 54 – **Script:** *By a domain script we shall understand a structured text which can be interpreted as a set of rules ("in disguise").* ■

s482
cacm-timetable

Example 62 – **Timetables:** We shall view timetables as scripts.

In this example (that is, Pages 98–102) 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.

s483 In Example. 65 on contracts for bus traffic, we shall assume the timetable scripts of this section.

We all have some image of how a timetable may manifest itself. Figure 7.9 shows some such images.

Fig. 7.9. Some bus timetables: Italy, India and Norway

s484

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.[10] 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".

The Syntax of Timetable Scripts

130. 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.
131. A TimeTable associates to Bus Line Identifiers a set of Journies.
132. Journies are designated by a pair of a BusRoute and a set of BusRides.
133. 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.
134. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
135. 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.
136. 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.
137. A Fraction is a **Real** properly between 0 and 1.
138. The Journies must be well_formed in the context of some net.

s485

type
130. T, BLId, BId
131. TT = BLId $\overrightarrow{m}$ Journies
132. Journies$'$ = BusRoute $\times$ BusRides
133. BusRoute = BusStop $\times$ BusStop* $\times$ BusStop
134. BusRides = BId $\overrightarrow{m}$ BusSched
135. BusSched = T $\times$ T* $\times$ T
136. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
137. Frac = {|r:**Real**•0<r<1|}
138. 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. s486

Well-formedness of Journies

139. A set of journies is well-formed
140. if the bus stops are all different[11],
141. if a defined notion of a bus line is embedded in some line of the net, and
142. if all defined bus trips (see below) of a bus line are commensurable.

value
139. wf_Journies: Journies $\rightarrow$ N $\rightarrow$ **Bool**
139. wf_Journies((bs1,bsl,bsn),js)(hs,ls) $\equiv$

[10] We do not distinguish between a time and a time description. That is, when we say August 22, 2009, 11: 58 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 !".

[11] This restriction is, strictly speaking, not a necessary domain property. But it simplifies our subsequent formulations.

140. diff_bus_stops(bs1,bsl,bsn) $\wedge$
141. is_net_embedded_bus_line($\langle$bs1$\rangle$$^\frown$bsl$^\frown$$\langle$bsn$\rangle$)(hs,ls) $\wedge$
142. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

s487

143. The bus stops of a journey are all different
144. if the number of elements in the list of these equals the length of the list.

value
143. diff_bus_stops: BusStop $\times$ BusStop* $\times$ BusStop $\rightarrow$ **Bool**
143. diff_bus_stops(bs1,bsl,bsn) $\equiv$
144. **card elems** $\langle$bs1$\rangle$$^\frown$bsl$^\frown$$\langle$bsn$\rangle$ = **len** $\langle$bs1$\rangle$$^\frown$bsl$^\frown$$\langle$bsn$\rangle$

s488 We shall refer to the (concatenated) list ($\langle$bs1$\rangle$$^\frown$bsl$^\frown$$\langle$bsn$\rangle$ = **len** $\langle$bs1$\rangle$$^\frown$bsl$^\frown$$\langle$bsn$\rangle$) of all bus stops as the bus line.

145. To explain that a bus line is embedded in a line of the net
146. let us introduce the notion of all lines of the net, lns,
147. and the notion of projecting the bus line on link sector descriptors.
148. 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.

s489

value
145. is_net_embedded_bus_line: BusStop* $\rightarrow$ N $\rightarrow$ **Bool**
145. is_net_embedded_bus_line(bsl)(hs,ls)
146. **let** lns = lines(hs,ls),
147. cbln = compress(proj_on_links(bsl)(**elems** bsl)) **in**
148. $\exists$ ln:Line • ln $\in$ lns $\wedge$ cbln $\in$ projs_on_links(ln) **end**

s490

149. Projecting a list (*) of BusStop descriptors (mkBS(hi,li,f,hi′)) onto a list of Sector Descriptors ((hi,li,hi′))
150. we recursively unravel the list from the front:
151. if there is no front, that is, if the whole list is empty, then we get the empty list of sector descriptors,
152. else we obtain a first sector descriptor followed by those of the remaining bus stop descriptors.

value
149. proj_on_links: BusStop* $\rightarrow$ SectDescr*
149. proj_on_links(bsl) $\equiv$
150. **case** bsl **of**
151. $\langle\rangle \rightarrow \langle\rangle$,
152. $\langle$mkBS(hi,li,f,hi′)$\rangle$$^\frown$bsl′ $\rightarrow$ $\langle$(hi,li,hi′)$\rangle$$^\frown$proj_on_links(bsl′)
152. **end**

s491

153. By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.
154. The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.
155. We express the function recursively.
156. If the argument sector descriptor list an empty result sector descriptor list is yielded;
157. else
158. 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"
159. in front of the compression of the rest of the argument sector descriptor list.

s492

153. compress: SectDescr* → SectDescr-**set** → SectDescr*
154. compress(sdl)(sds) ≡
155. **case** sdl **of**
156. ⟨⟩ → ⟨⟩,
157. ⟨sd⟩⌒sdl′ →
158. (**if** sd ∈ sds **then** ⟨sd⟩ **else** ⟨⟩ **end**)
159. ⌒compress(sdl′)(sds\{sd}) **end**

In the last recursion iteration (line 159.) the continuation argument sds\{sd} can be shown to be empty: {}. s493

160. We recapitulate the definition of lines as sequences of sector descriptions.
161. Projections of a line generate a set of lists of sector descriptors.
162. 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
160. Line′ = (HI×LI×HI)* **axiom** ... **type** Line = ...
value
161. projs_on_links: Line → Line′-**set**
161. projs_on_links(ln) ≡
162. {⟨isl(i)|i:⟨1..**len** isl⟩⟩|isx:**Nat-set**•isx⊆**inds** ln∧isl=sort(isx)}

s494

163. 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.
164. The result list of (arbitrary) indices must contain all the members of the argument set;
165. and "earlier"elements of the list must precede, in value, those of "later" elements of the list.

value
163. sort: **Nat-set** → **Nat***
163. sort(isx) **as** isl
164. **post card** isx = lsn isl ∧ isx = **elems** isl ∧
165. ∀ i:**Nat** • {i,i+1}⊆**inds** isl ⇒ isl(i)<isl(i+1)

s495

166. The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:
167. All the intermediate bus stop times must equal in number that of the bus stop list.
168. 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.
169. 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. s496
170. 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.
171. 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.
172. As for Item 171 but now with respect to last, resp. destination bus stop.
173. And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops
174. they must exist and fit time-wise.

s497

```
value
166. commensurable_bus_trips: Journies → N → Bool
166. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
167.   ∀ (t1,til,tn):BusSched•(t1,til,tn)∈ rng js∧len til=len bsl∧
168.     case len til of
169.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
170.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧fit((til(1),t2),(bsl(1),bsn))(hs,ls),
171.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
172.           real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
173.           ∀ i:Nat•{i,i+1}⊆inds til ⇒
174.             real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

s498

175. 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:
176. All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).
177. There exists links, l, l′, for the identified bus stop links, li, li′,
178. such that these links connect the identified bus stop hubs.
179. Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops
180. The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.
181. 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.
182. The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.
183. So we leave also this as an exercise to the reader to complete.

s499

```
175. real_and_fit: (T×T)×(BusStop×BusStop) → N → Bool
175. real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
176.   {hi,hi′,hi″,hi‴}⊆his(hs)∧
177.   ∃ l,l′:L•{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
178.     obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
179.   afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

180. distance: BusStop × BusStop → N → Distance
181. distance(bs1,bs2)(n) ≡ ... [ left as an exercise ! ] ...

182. afit: TI → Distance → Bool
183.   [ time interval fits distance between bus stops ]
```

This ends Example 62 ∎

s500
acm-tsode-3

Definition 55 – **Licenses:** *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 [136].* ∎

s501

Definition 56 – **Contract:** *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.* ∎

s502 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. s503

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. s504

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. s505

cacm-hcll

Example 63 – **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. s506

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. s507

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. s508

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 [7].[12] s509

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. 7.10 on the next page for an idealised hospitalisation plan.

s510

[12] As this work, [7], has yet to be completed the syntax and annotations given here may change.

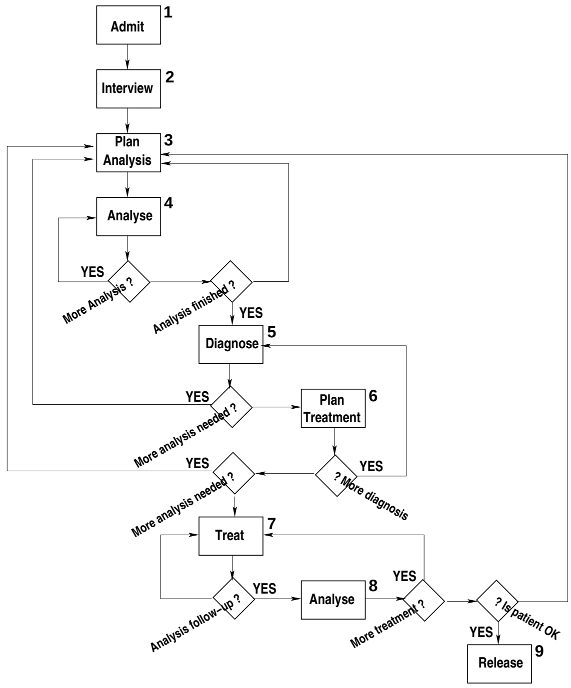


Fig. 7.10. An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

s511

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.

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

s512 and prevent it being misused!

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 × 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**

s513

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.

s514

(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.

s515

(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.

s516

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. s517Each 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.

This ends Example 63 ∎

s518
cacm-pall

Example 64 – **A Public Administration License Language:**

The Three Branches of Government By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)[13], 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.). s519

Documents A crucial means of expressing public administration is through *documents*.[14] 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.)

s520

Documents are *created*, *edited* and *read*; and documents can be *copied*, *distributed*, the subject of *calculations* (interpretations) and be *shared* and *shredded* . s521

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. s522

[13] *De l'esprit des lois* (*The Spirit of the Laws*), published 1748

[14] Documents are, for the case of public government to be the "equivalent" of artistic works.

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
s523 documents. We now intend to express these *authorisations as licenses*.

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
s524 all the documents any one document-creation or -editing was based on.

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)
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

s525

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)

s526

(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.[15] A set of operations
s527 on (named) documents are granted.

(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
s528 unique document identifiers (UDI).

(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[16]) typed by a document class name (dcn:DCn)

[15]This constraint can be enforced by letting the actor name be part of the license name.

[16]— hence there is an assumption here that the create operation is invoked by the licensee exactly (or at most) once.

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 (zero, one or more) documents.[17] The “based on” documents are moved from licensor to licensee. s529

(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. s530

(10.) **Reading** a document only changes its “having been read” status (etc.) — as per [14]. 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. s531

(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 [14]. 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. s532

(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. s533

(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. s534

(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. s535

(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. s536

(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.

(18.) A document can be **shredded**. It seems pointless to shred a document if that was the only right granted wrt. document. s537

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)

[17] They can therefore be traced (etc.) — as per [14].

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)

s538 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
s539 the "system".

(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
s540 — in which cases operations are inherited.

(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
s541 of the master are inherited to the copy.

(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
s542 or more either extensions or restrictions, and finally, perhaps, a withdrawal.

(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
s543 opening of the sharing session, can be read by all so-granted other actors.

(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
s544 this document and eventually to re-license the licensee to further work.

(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
s545 no longer operate on it.

(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.

This ends Example 64 ■

acm-tsode-3
s546
cacm-bscl
Example 65 – **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.)

s547 *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[18]. The cancellation rights are spelled out in the contract[19]. A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor[20]. Etcetera.

A Pragmatics and Semantics Analysis : s548

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. 62. 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 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. s549

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[21] We assume that such insertions must also be reported back to the contractor. s550

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.) s551

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. s552

Syntax We treat separately, the syntax of contracts (for a schematised example see Page 109) and the syntax of the actions implied by contracts (for schematised examples see Page 110).

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.

s553

We assume a context (a global state) in which all contract actions (including contracting) takes place and in which the implicit net is defined. s554

184. contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.
185. 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.

[18] We do not treat this aspect further in this book.
[19] See Footnote 18.
[20] See Footnote 18.
[21] Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

186. A contract body stipulates a timetable and the set of operations that are mandated or allowed by the contractor.
187. 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.

s555

type
184. CId, CNm
185. Contract = CId × CNm × CNm × Body
186. Body = Op-**set** × TT
187. Op == ″start″ | ″cancel″ | ″insert″ | ″subcontract″

An abstract example contract:
(cid,cnm_i,cnm_j,({″start″,″cancel″,″insert″,″sublicense″},tt))

s556

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 109) 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′.

s557

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
s558 subcontracting action can (thus) be simply transformed into a contract as shown on Page 109.

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 190 on the facing page

s559

We observe that the essential information given in the start, cancel and insert action prescriptions is
s560 the same; and that the RSL record-constructors (mkSta, mkCan, mkIns) make them distinct.

Uniqueness and Traceability of Contract Identifications

188. There is a "root" contract name, rcid.
189. There is a "root" contractor name, rcnm.

value
188 rcid:CId
189 rcnm:CNm

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. s561 s562

190. Such a contract name generator is a function which given a contract identifier, a sub-contractor name and the time at which the new contract identifier is generated, yields the unique new contract identifier.
191. 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
190 gen_CId: CId $\times$ CNm $\times$ Time $\rightarrow$ CId
191 obs_CId: CId $\xrightarrow{\sim}$ CIdL [**pre** obs_CId(cid):cid$\neq$rcid]
191 obs_CNm: CId $\xrightarrow{\sim}$ CNm [**pre** obs_CNm(cid):cid$\neq$rcid]
191 obs_Time: CId $\xrightarrow{\sim}$ Time [**pre** obs_Time(cid):cid$\neq$rcid]

s563

192. All contract names are unique.

axiom
192 $\forall$ cid,cid$'$:CId•cid$\neq$cid$'\Rightarrow$
192 obs_CId(cid)$\neq$obs_CId(cid$'$) $\vee$ obs_CNm(cid)$\neq$obs_CNm(cid$'$)
192 $\vee$ obs_LicNm(cid)=obs_CId(cid$'$)$\wedge$obs_CNm(cid)=obs_CNm(cid$'$)
192 $\Rightarrow$ obs_Time(cid)$\neq$obs_Time(cid$'$)

s564

193. 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
193 contract_trace: CId $\rightarrow$ LCIdCNmTTrace
193 contract_trace(cid) $\equiv$
193 **case** cid **of**
193 rcid $\rightarrow$ $\langle\rangle$,
193 _ $\rightarrow$ contract_trace(obs_LicNm(cid))$^\frown\langle$obs_TraceTriple(cid)$\rangle$
193 **end**

193 obs_TraceTriple: CId $\rightarrow$ TraceTriple
193 obs_TraceTriple(cid) $\equiv$
193 mkTrTr(obs_CId(cid),obs_CNm(cid),obs_Time(cid))

The trace is generated in the chronological order: most recent contract name generation times last. s565

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
$\forall$ licn:LicNm •
$\forall$ trace:LicNmLeeNmTimeTrace • trace $\in$ license_trace(licn) $\Rightarrow$
$\forall$ i:**Nat** • {i,i+1}$\subseteq$**inds** trace $\Rightarrow$ s_t(trace(i))<s_t(trace(i+1))

s566

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
s567 sub-contractors. We now examine, in some detail, what the states consist of.

Global State

The net is part of the global state (and of bus traffics). We consider just the bus traffic.

194. 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.
195. A bus is positioned either
196. at a hub (coming from some link heading for some link), or
197. on a link, some fraction of the distance from a hub towards a hub, or
198. at a bus stop, some fraction of the distance from a hub towards a hub.

s568

type
136. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

194. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos)))
195. BPos = atHub | onLnk | atBS
196. atHub == mkAtHub(s_fl:LI,s_hi:HI,s_tl:LI)
197. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
198. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
Frac = {|f:**Real**•0<f<1|}

s569 We shall consider BusTraffic (with its Net) to reflect the global state.

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 $\overrightarrow{m}$ (LicNm $\overrightarrow{m}$ (Body×TT))
SubLicΣ = LeeNm $\overrightarrow{m}$ (LicNm $\overrightarrow{m}$ 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
s570 the second LicNm.

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.[22] s571

```
type
   BusNo
   BusΣ = FreeBusesΣ × ActvBusesΣ × BusHistsΣ
   FreeBusesΣ = BusStop →m BusNo-set
   ActvBusesΣ = BusNo →m BusInfo
   BusInfo = BLId×BId×LicNm×LeeNm×BusTrace
   BusHistsΣ = Bno →m BusInfo*
   BusTrace = (Time×BusStop)*
   LorBusΣ = LeeNm →m (LicNm →m ((BLId×BId) →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. s572

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
```

s573

```
   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
```

[22] 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.

((freeσ\{bno},actvσ ∪ [bno↦(blid,bid,licnm,leenm,⟨⟩)]),bno) **end**
pre bs ∈ **dom** freeσ ∧ bno ∈ freeσ(bs) ∧ bno ∉ **dom** actvσ ∧ [bs exists ...]

s574

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.

199. 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.
200. 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 —
s575 and these are unchanged wrt. constancy or variability of contract holders.
201. There is an initial bus traffic, tr.
202. There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.
203. To maintain the bus traffic "spelled out", in total, by timetable tt one needs a number of buses.
204. 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.
205. We leave it to the reader to define a function which ascertain the minimum number of buses needed to implement traffic tr.

s576

value
199. net : N,
200. leens : LeeNm-**set**,
201. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
202. t_0 : T • t_0 ≥ **min dom** tr,
203. min_no_of_buses : **Nat** • necessary_no_of_buses(itt),
204. busnos : BusNo-**set** • **card** busnos ≥ min_no_of_buses
205. necessary_no_of_buses: TT → **Nat**

s577

206. 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]).
207. The initial, i.e., the distinguished, contract has a name, root_licn.
208. The initial contract can only perform the `"sublicense"` operation.
209. The initial contract has a timetable, tt.
210. The initial contract can thus be made up from the above.

s578

value
206. root_leen,init_ln : LeeNm • root_leen ∉ leens ∧ initi_leen ∈ leens,
207. root_licn : LicNm
208. iops : Op-**set** = {"`sublicense`"},
209. itt : TT,
210. init_lic:License = (root_licn,root_leen,(iops,itt),init_leen)

s579

Initial Sub-contractor Contract States

type
InitLicΣs = LeeNm $\overrightarrow{m}$ LicΣ
value
ilσ:LicΣ=([init_leen ↦ [root_leen ↦ [iln ↦ init_lic]]]
∪ [leen ↦ [ ] | leen:LeeNm • leen ∈ leenms\{init_leen}],[ ],[ ])

s580

Initial Sub-contractor Bus States

211. Initially each sub-contractor possesses a number of buses.
212. No two sub-contractors share buses.
213. We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.
214. We do not prescribe a "satisfiable and practical" such initial assignment (ibσs).
215. But we can constrain ibσs.
216. The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.
217. Active bus states must be empty.
218. No two free bus states must share buses.
219. All bus histories are void. s581

type
211. AllBuses′ = LeeNm $\overrightarrow{m}$ BusNo-**set**
212. AllBuses = {|ab:AllBuses′•∀ {bs,bs′}⊆**rng** ab∧bns≠bns′⇒bns ∩ bns′={}|}
213. InitBusΣs = LeeNm $\overrightarrow{m}$ BusΣ
value
212. allbuses:Allbuses • **dom** allbuses = leenms ∪ {root_leen} ∧ ∪ **rng** allbuses = busnos

213. ibσs:InitBusΣs
214. wf_InitBusΣs: InitBusΣs → **Bool**
215. wf_InitBusΣs(iσs) ≡
216. **dom** iσs = leenms ∧
217. ∀ (_,abσ,_):BusΣ•(_,abσ,_) ∈ **rng** iσs ⇒ abσ=[ ] ∧
218. ∀ (fbiσ,abiσ),(fbjσ,abjσ):BusΣ •
218. {(fbiσ,abiσ),(fbjσ,abjσ)}⊆**rng** iσs
218. ⇒ (fbiσ,actiσ)≠(fbjσ,actjσ)
218. ⇒ **rng** fbiσ ∩ **rng** fbjσ = {}
219. ∧ actiσ=[ ]=actjσ

Communication Channels : s582
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. s583

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. s584

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 = ...

s585

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 = ...

s586

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)

s587

Buses↔Time Channel

Each bus needs to know what time it is.

channel { b_to_t[b] | b:BNo • b ∈ busnos } BTMSG
type BTMSG ...

s588

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.
s589 Thus there are n such channels.

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.

s590 *The System Behaviour* :
The system behaviour starts by establishing a number of licenseholder and bus_ride behaviours and
s591 the single time_clock and bus_traffic behaviours

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**(t_0) ‖ **bus_traffic**(tr)

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. s592

Only a designated licenseholder behaviour is initialised to a single, received license. s593

Semantic Elaboration Functions

The Licenseholder Behaviour

220. The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.
221. 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.

220. **licenseholder**: LeeNm → (LicΣ×BusΣ) → **Unit**
221. **licenseholder**(leen)(licσ,busσ) ≡
221.   **licenseholder**(leen)((**lic_ops**⌈⌉**bus_mon**⌈⌉**neg_licenses**)(leen)(licσ,busσ))

s594

The Bus Behaviour

222. Buses ply the network following a timed bus route description.
    A timed bus route description is a list of timed bus stop visits.
223. A timed bus stop visit is a pair: a time and a bus stop.
224. 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.
225. Bus behaviours start with a "nil" bus route description.

s595

type
222. TBR = TBSV*
223. TBSV = Time × BusStop
value
224. conTBR: BusRoute × BusSched → TBR
224. conTBR((dt,til,at),(bs1,bsl,bsn)) ≡
224a)   ⟨(dt,bs1)⟩
224b)   ⌢ ⟨(til[i],bsl[i])|i:**Nat**•i:⟨1..**len** til⟩⟩
224c)   ⌢ ⟨(at,bsn)⟩
        pre: **len** til = **len** bsl
type
225. BRD == "nil" | TBR

s596

226. The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,
227. The bus repeatedly observes the time, t, and its position, po, in the traffic.
228. There are now four case distinctions to be made.

229. If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.
230. 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.
231. If the bus is 'en route', at a bus stop, then it so informs its owner and continues the journey.
232. In all other cases the bus continues its journey

s597

```
value
226. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
226.    in,out l_to_b[leen,bno], in,out b_to_tr[bno], in b_to_t[bno] Unit
226. bus_ride(leen,bno)(licn,brd) ≡
227.   let t = b_to_t[bno]? in
227.   let (bus,pos) = (b_to_tr[bno]!reqBusAndPos(bno,t) ; b_to_tr[bno]?) in
228.   case (brd,pos) of
229.     (″nil″,mkAtBS(_,_,_,_)) →
229.          let (licn,brd′) = (l_to_b[leen,bno]!reqBusRid(pos);l_to_b[leen,bno]?) in
229.          bus_ride(leen,bno)(licn,brd′) end
230.     (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
230s          l_to_b[l,b]!BusΣMsg(t,pos);
230          l_to_b[l,b]!BusHistΣMsg(licn,bno);
230          l_to_b[l,b]!FreeΣ_ActΣMsg(licn,bno) ;
230          bus_ride(leen,bno)(ilicn,″nil″),
231.     (⟨(t,pos),(t′,bs′)⟩⌢brd′,mkAtBS(_,_,_,_)) →
231s          l_to_b[l,b]!BusΣMsg(t,pos) ;
231          bus_ride(licn,bno)(⟨(t′,bs′)⟩⌢brd′),
232.     _ → bus_ride(leen,bno)(licn,brd) end end end
```

s598

In formula line 227 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 230s and 231s:

```
Simple:  230s  l_to_b[l,b]!BusΣMsg(pos);
Revised: 230r  l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));

Simple:  231s  l_to_b[l,b]!BusΣMsg(pos);
Revised: 231r  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.
-
-
-

s599

The Global Time Behaviour

233. The time_clock is a never ending behaviour — started at some time t_0.
234. The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.
235. At any moment the time_clock behaviour may not be inquired.
236. After a skip of the clock or an inquiry the time_clock behaviour continues, non-deterministically either maintaining the time or advancing the clock! s600

```
value
233. time_clock: T →
233.     in,out {l_to_t[leen] | leen:LeeNm • leen ∈ leenms}
233.     in,out {b_to_t[bno] | bno:BusNo • bno ∈ busnos}  Unit
233.  time_clock(t) ≡
235.    (skip ⌈⌉
234.    (⌈⌉{l_to_t[leen]? ; l_to_t[leen]!t | leen:LeeNm•leen ∈ leens})
234.    ⌈⌉ (⌈⌉{b_to_t[bno]? ; b_to_t[bno]!t | bno:BusNo•bno ∈ busnos})) ;
236.    (time_clock(t) ⌈⌉ time_clock(t+δt))
```

s601

The Bus Traffic Behaviour

237. There is a single bus_traffic behaviour. It is, “mysteriously”, given a constant argument, “the” traffic, tr.
238. 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.
239. The request for a bus position comes from some bus.
240. The bus positions are part of the traffic at time t.
241. The bus_traffic behaviour, after informing of a bus position reverts to “itself”. s602

```
value
237.  bus_traffic: TR → in,out {b_to_tr[bno]|bno:BusNo•bno ∈ busnos} Unit
237.  bus_traffic(tr) ≡
239.    ⌈⌉ { let reqBusAndPos(bno,time) = b_to_tr[b]? in assert b=bno
238.         if time ∉ dom tr then chaos else
240.         let (_,bps) = tr(t) in
238.         if bno ∉ dom tr(t) then chaos else
238.         b_to_tr[bno]!bps(bno) end end end end | b:BusNo•b ∈ busnos} ;
241.    bus_traffic(tr)
```

s603

License Operations

242. 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 110.
243. To perform any licensed operation the sub-contractor needs to know the time and
244. 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.
245. Thus there is a case distinction amongst four alternatives.
246. This case distinction is expressed in the four lines identified by: 246.
247. All the auxiliary functions, besides the action arguments, require the same state arguments. s604

```
value
242. lic_ops: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
242. lic_ops(leen)(licσ,busσ) ≡
243.   let t = (time_channel(leen)!req_Time;time_channel(leen)?) in
244.   let (licn,act) = choice(licσ)(busσ)(t) in
245.   (case act of
246.      mkCon(blid,bid)  → cndct(licn,leenm,t,act),
246.      mkCan(blid,bid)   → cancl(licn,leenm,t,act),
246.      mkIns(blid,bid)   → insrt(licn,leenm,t,act),
246.      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)
```

s605

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)
```

s606

248. The **local_bus_mon**itoring function models all the interaction between a contract holder and its despatched buses.
249. We show only the communications from buses to contract holders.
250.
251.
252.
253.
254.
255.
256.
257.
258.

s607

```
248. local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
249.       → in {l_to_b[leen,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
248. local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
250.    let (bno,msg) = ⌈⌉{(b,l_to_b[l,b]?)|b:BNo•b ∈ allbuses(leen)} in
254.    let (blid,bid,licn,lorn,trace) = abσ(bno) in
251.    case msg of
252.      BusΣMsg(t,bs) →
256.        let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
256.        (licσ,(fbσ,abσ′,histσ)) end,
258.      BusHistΣMsg(licn,bno) →
258.        let lbσ′ =
258.           update_LorBusΣ(obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))(lbσ)  in
258.        l_to_l[leen,obs_LorNm(licn)]!Licensor_BusHistΣMsg(licn,leen,bno,blid,bid,tr);
```

```
258.     ((rlσ,slσ,lbσ′),busσ) end
257.   FreeΣ_ActΣMsg(licn,bno) →
258.     let (fbσ′,abσ′) = update_FreeΣ_ActΣ(bno,bs)(fbσ,abσ) in
258.     (licσ,(fbσ′,abσ′)) end
258. end end end
```

s608

259.
260.
261.
262.
263.

s609

```
259. licensor_bus_mon: lorn:LorNm → (LicΣ×BusΣ)
259.     → in {l_to_l[ lorn,leen ]|leen:LeeNm•leen ∈ leenms\{lorn}} (LicΣ×BusΣ)
259. licensor_bus_mon(lorn)(licσ,busσ) ≡
259.   let (rlσ,slσ,lbhσ) = licσ in
259.   let (leen,Licensor_BusHistΣMsg(licn,leen″,bno,blid,bid,tr))
                         = ⌈⌉{(leen′,l_to_l[ lorn,leen′ ]?)|leen′:LeeNm•leen′ ∈ leenms\{lorn}} in
259.   let lbhσ′ =
259.     update_BusHistΣ(obs_LorNm(licn),licn,leen″,(blid,bid),(bno,trace))(lbhσ) in
259.   l_to_l[ leenm,obs_LorNm(licnm) ]!Licensor_BusHistΣMsg(b,blid,bid,lin,lee,tr);
259.   ((rlσ,slσ,lbhσ′),busσ)
259.   end end end
```

s610

License Negotiation

264.
265.
266.
267.
268.
269.
270.
271.
272.
273.
274.
275.

s611

264.
265.
266.
267.
268.
269.
270.
271.
272.
273.
274.

s612

The Conduct Bus Ride Action

276. 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. s613
   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.

s614

276.
276a)
276b)
276c)
276d)
276e)
276f)
276g)

s615

The Cancel Bus Ride Action

277. 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 276b) 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 !

s616

277.
277a)
277b)
277c)

s617

The Insert Bus Ride Action

278. 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 276b) 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 276d)–276g).

s618

278.
278a)
278b)
278c)
278d)

s619

The Contracting Action

279. 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.

s620

279.
279a)
279b)
279c)
279d)
279e)
279f)

s621

Discussion This ends Example 65 ■

acm-tsode-3

7.7.1 Principles s622

7.7.2 Discussion s623

7.8 Domain Human Behaviour s624

acm-tsode-4

Definition 57 – **Human Behaviour:** *By* **human behaviour** *we mean any of a quality spectrum of carrying out assigned work: from (i)* **careful, diligent** *and* **accurate***, via (ii)* **sloppy** *dispatch, and (iii)* **delinquent** *work, to (iv) outright* **criminal** *pursuit.* ■

s625

Example 66 – **A Casually Described Bank Script:** Our formulation amounts to just a (casual) rough sketch. It is followed by a series of three larger examples (Examples 67–69). Each of these elaborate on the theme of (bank) scripts.

The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest s626
on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment. s627

We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts.

(i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage bal-
s628 ance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. This ends Example 66 ■

s629

Example 67 – **A Formally Described Bank Script:** First we must informally and formally define the bank state:

There are clients (c:C), account numbers (a:A), mortgage numbers (m:M), account yields (ay:AY) and mortgage interest rates (mi:MI). The bank registers, by client, all accounts (ρ:A_Register) and all mortgages (μ:M_Register). To each account number there is a balance (α:Accounts). To each mortgage number there is a loan (ℓ:Loans). To each loan is attached the last date that interest was paid on the loan.

s630

```
value
   r, r′:Real axiom ...
type
   C, A, M, Date
   AY′ = Real, AY = {| ay:AY′ • 0<ay≤r |}
   MI′ = Real, MI = {| mi:MI′ • 0<mi≤r′ |}
   Bank′ = A_Register × Accounts × M_Register × Loans
   Bank = {| β:Bank′ • wf_Bank(β)|}
   A_Register = C →m A-set
   Accounts = A →m Balance
   M_Register = C →m M-set
   Loans = M →m (Loan × Date)
   Loan,Balance = P
   P = Nat
```

s631 Then we must define well-formedness of the bank state:

```
value
   ay:AY, mi:MI

   wf_Bank: Bank → Bool
   wf_Bank(ρ,α,μ,ℓ) ≡ ∪ rng ρ = dom α ∧ ∪ rng μ = dom ℓ
axiom
   ay<mi [ ∧ ... ]
```

s632 We — perhaps too rigidly — assume that mortgage interest rates are higher than demand/deposit account interest rates: ay<mi.

Operations on banks are denoted by the commands of the bank script language. First the syntax:

```
type
   Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
   OpA == mkOA(c:C)
   CloA == mkCA(c:C,a:A)
   Dep == mkD(c:C,a:A,p:P)
   Wdr == mkW(c:C,a:A,p:P)
   OpM == mkOM(c:C,p:P)
   Pay == mkPM(c:C,a:A,m:M,p:P,d:Date)
```

CloM == mkCM(c:C,m:M,p:P)
Reply = A | M | P | OkNok
OkNok == ok | notok

value
period: Date $\times$ Date $\rightarrow$ Days [for calculating interest]
before: Date $\times$ Date $\rightarrow$ **Bool** [first date is earlier than last date]

And then the semantics: s633

int_Cmd(mkPM(c,a,m,p,d))(ρ,α,μ,ℓ) $\equiv$
  let (b,d$'$) = ℓ(m) **in**
  if α(a)$\geq$p
    then
      let i = interest(mi,b,period(d,d$'$)),
        ℓ' = ℓ $\dagger$ [m$\mapsto\ell$(m)$-$(p$-$i)]
        α' = α $\dagger$ [a$\mapsto\alpha$(a)$-$p,$a_i\mapsto\alpha(a_i)$+i] **in**
      ((ρ,α',μ,ℓ'),ok) **end**
    else
      ((ρ,α',μ,ℓ),nok)
  end end
  pre c $\in$ **dom** μ $\wedge$ a $\in$ **dom** α $\wedge$ m $\in$ μ(c)
  post before(d,d$'$)

  interest: MI $\times$ Loan $\times$ Days $\rightarrow$ P

This ends Example 67 ∎ s634

Example 68 – **Bank Staff or Programmer Behaviour:** Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments (cf. Example 67).

We would characterise such a clerk as being *diligent,* etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations "tally up", or lets others do so. We would characterise a clerk as being *sloppy* if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being *delinquent* if that person systematically forgets these checks. And we would call such a person a *criminal* if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater. s635

Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 67).

We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. This ends Example 68 ∎ s636

Example 69 – **A Human Behaviour Mortgage Calculation:** Example 67 gave a semantics to the mortgage calculation request (i.e., command) as would a diligent bank clerk be expected to perform it. To express, that is, to model, how sloppy, delinquent, or outright criminal persons (staff?) could behave we must modify the int_Cmd(mkPM(c,a,m,p,d$'$))(ρ,α,μ,ℓ) definition.

int_Cmd(mkPM(c,a,m,p,d))(ρ,α,μ,ℓ) $\equiv$
  let (b,d$'$) = ℓ(m) **in**

if q(α(a),p) /* α(a)$\leq$p$\vee\alpha$(a)=p$\vee\alpha$(a)$\leq$p$\vee$... */
  then
    let i = f_1(interest(mi,b,period(d,d$'$))),
      $\ell' = \ell \dagger$ [m$\mapsto f_2$(ℓ(m)−(p−i))]
      $\alpha' = \alpha \dagger$ [a$\mapsto f_3$(α(a)−p),$a_i\mapsto f_4$(α(a_i)+i),
          $a_{\text{“staff”}} \mapsto f_{\text{“staff”}}(\alpha(a_{\text{“staff”}})$+i)] **in**
    ((ρ,α',μ,ℓ'),ok) **end**
  else
    ((ρ,α',μ,ℓ),nok)
end end
pre c $\in$ **dom** $\mu \wedge$ m $\in \mu$(c)

q: P $\times$ P $\stackrel{\sim}{\to}$ **Bool**
f_1,f_2,f_3,f_4,$f_{\text{“staff”}}$: P $\stackrel{\sim}{\to}$ P [typically: $f_{\text{“staff”}} = \lambda$p.p]

s637 The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ of Example 67 are deliberately left undefined. They are being defined by the “staffer” when performing (incl., programming) the mortgage calculation routine.

The point of Example 67 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) to perform (incl., correctly program) it before one can “pinpoint” all the places where lack of diligence may “set in”. The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ designate those places.

The point of Example 67 is also that we must first domain-define, “to the best of our ability” all the places where human behaviour may play other than a desirable role. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour. This ends Example 69 ■

s638

Example 70 – **Transport Net Building:**
We show the example in two stages: First we show a description of a diligent operation; then of a less careful operation.

Sub-example 1 *(of Example 70 –)* **A Diligent Operation:** The int_Insert operation of Example 10 Page 11 was expressed without stating necessary pre-conditions:

11 [23] 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. s639
12 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.
13 If the command is of the form 2oldH(hi$'$,l,hi$'$) 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 hi$''$ must be the identifier of a(nother) hub in hs.

s640

14 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.
15 If the command is of the form 2newH(h$'$,l,h$''$) then
   ⋆1 h$'$ — left to the reader as an exercise (see formalisation !),

[23] See Page 13 for Item 11 et cetera.

⋆2 l — left to the reader as an exercise (see formalisation !), and
⋆3 h″ — left to the reader as an exercise (see formalisation !).

s641

value
12′ pre_int_Insert: Ins → N → **Bool**
12″ pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2 s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
case op **of**
13 2oldH(hi′,l,hi″) → {hi′,hi″}⊆iohs(hs),
14 1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∧h∉ hs∧obs_HI(h)∉ iohs(hs),
15 2newH(h′,l,h″) → {h′,h″}∩ hs={}∧{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
end

These must be **carefully** expressed and adhered to in order for staff to be said to carry out the link insertion operation **accurately**. This ends Sub-example 70.1 ■

s642

Sub-example 2 *(of Example 70 –)* **A Sloppy via Delinquent to Criminal Operation:** We replace systematic checks (∧) with partial checks (∨), etcetera, and obtain various degrees of **sloppy** to **delinquent**, or even **criminal** behaviour.

value
12′ pre_int_Insert: Ins → N → **Bool**
12″ pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2 s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
case op **of**
13 2oldH(hi′,l,hi″) → hi′ ∈ iohs(hs)∨hi″isin iohs(hs),
14 1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∨h∉ hs∨obs_HI(h)∉ iohs(hs),
15 2newH(h′,l,h″) → {h′,h″}∩ hs={}∨{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
end

This ends Sub-example 70.2 ■

This ends Example 70 ■

7.8.1 A Meta Characteristic of Human Behaviour s643

Commensurate with the above, humans interpret rules and regulations differently, and not always consistently — in the sense of repeatedly applying the same interpretations. s644

Schema 3 – **A Human Behaviour Specification Pattern:**

type
[1] α: Action = $\Sigma \stackrel{\sim}{\to} \Sigma$-**infset**
value
[2] hum_int: Rule → Σ → RUL-**infset**
[3] action: Stimulus → Σ → Σ
[4] hum_beha: Stimulus × Rules → Action → $\Sigma \stackrel{\sim}{\to} \Sigma$-**infset**
[5] hum_beha(sy_sti,sy_rul)(α)(σ) **as** σset
[6] **post**
[7] σset = $\alpha(\sigma)$ ∧ action(sy_sti)(θ) ∈ θset
[8] ∧ ∀ σ':Σ•σ' ∈ σset ⇒
[9] ∃ se_rul:RUL•se_rul ∈ hum_int(sy_rul)(σ)⇒se_rul(σ,σ')

■ s645

The above is, necessarily, sketchy: [1] There is a possibly infinite variety of ways of interpreting some rules. [2] A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. “Suits” means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not.

The above-stated axioms express how it is in the domain, not how we would like it to be. For that we have to establish requirements.

7.8.2 Principles s646

7.8.3 Discussion s647

7.9 Opening and Closing Stages s648

acm-tsode-e

We cover in this section the following aspects of domain engineering:

- opening stages (Sect. 7.9.1);
- closing stages (Sect. 7.9.2); and
- domain engineering documentation — (Sect. 7.9.3).

Sections

7.9.1 Opening Stages s649

For completeness, we shall briefly list the opening stages of domain engineering. They are:

1. domain stake-holder identification (and subsequent liaison);
2. rough sketching of business processes;
3. domain acquisition literature study, Internet study, on-site interviews, questionnaire preparation, questionnaire fill-in, and questionnaire handling — resulting in a great number of **domain description units**;
4. domain analysis (based on domain description units) and concept formation, and
5. domain “terminologisation”.

Stakeholder Identification and Liaison s650

Domain Acquisition s651

Domain Analysis s652

Terminoligisation s653

7.9.2 Closing Stages s654

For completeness, we shall, as in Sect. 7.9.1, briefly list the closing stages of domain engineering. They are:

1. domain verification, model checking and testing – the assurance of properties of the formalisation of the domain model (Sect. 7.9.2);
2. domain validation – the assurance of the veracity of the informal, i.e., the narrative domain description (Sect. 7.9.2); and
3. domain theory formation (Sect. 7.9.2).

Other than this brief mentioning we shall not cover these, from an engineering view-point rather important stages — but refer to [17].

Verification, Model Checking and Testing s655

Domain Validation s656

Domain Theory s657

7.9.3 Domain Engineering Documentation s658

7.9.4 Conclusion s659

7.10 Exercises

See Items 11–18 (of Appendix B, starting Page 196).

Part V

Requirements

8

Requirements Engineering

s660 acm-rtre-a

8.1 Characterisations

Definition 58 – **IEEE Definition of 'Requirements':** *By a requirements we understand (cf. IEEE Standard 610.12 [82]): "A condition or capability needed by a user to solve a problem or achieve an objective".* ■

s661

Principle 4 – **Requirements Engineering [1]:** *Prescribe only those requirements that can be objectively shown to hold for the designed software.* ■

Principle 5 – **Requirements Engineering [2]:** *When prescribing requirements, formulate, at the same time, tests (theorems, properties for model checking) whose actualisation should show adherence to the requirements* ■

s662

Definition 59 – **Requirements:** *By requirements we shall understand a document which prescribes desired properties of a machine: (i) what entities the machine shall "maintain", and what the machine shall (must; not should) offer of (ii) functions and of (iii) behaviours (iv) while also expressing which events the machine shall "handle".* ■

s663

A requirements prescription ideally specifies externally observable *properties* of simple entities, functions, events and behaviours of **the machine** such as the requirements stake-holders wish them to be. s664

Above we used the term 'ideally'. Even in good practice the requirements engineer may, here and there in the requirements prescription, resort to prescribe the requirements more by *how* it effects the *what* rather than only (i.e., 'ideally') prescribe the requirements by *what* the machine is to offer. s665

The **machine** is what is required, that is, the **hardware** and **software** that is to be designed and which are to satisfy the requirements.

It is a highlight of this document that requirements engineering has a scientific foundation and that that scientific foundation is the domain theory, that is the properties of the domain as modelled by a domain description. s666

Conventional requirements engineering, as covered in a great number of software engineering textbooks [62, 113, 115, 133, 140], does not have (such) a scientific foundation. This foundation allows us to pursue requirements engineering in quite a new manner.

The key idea of the kind of requirements engineering that we shall present is that a major part of the requirements can be systematically "derived" from a description of the domain in which the requirements 'reside'.

8.2 The Core Stages of Requirements Engineering s667

The core stages of requirements engineering are therefore those of ‘deriving’ the following requirements facets: business process re-engineering (Sect. 8.4), domain requirements[1] (Sect. 8.6 on page 141), interface requirements[2] (Sect. 8.8 on page 154) and machine requirements[3] (Sect. 8.9 on page 166).

8.3 On Opening and Closing Requirements Engineering Stages s668

acm-bpr We refer to Sect. 8.10.

8.4 Business Process Re-Engineering s669

We remind the reader of Sect. 7.2 (in which we covered the concept of ‘business processes’).

Definition 60 – **Business Process Re-Engineering:** *By business process re-engineering we understand the reformulation of previously adopted business process descriptions, together with additional business process engineering work.* ■

s670

Business process re-engineering (BPR) is about *change*, and hence BPR is also about *change management*. The concept of workflow is one of these “hyped” as well as “hijacked” terms: They sound good, and they make you “feel” good. But they are often applied to widely different subjects, albeit having some phenomena in common. By workflow we shall, very loosely, understand the physical movement of people, materials, information and “centre (‘locus’) of control” in some organisation (be it a factory, a hospital or other).

8.4.1 Michael Hammer’s Ideas on BPR s671

Michael Hammer, a guru of the business process re-engineering “movement”, states [66]:

1. *Understand a method of re-engineering before you do it for serious.*

So that is what this section is about.

2. *One can only re-engineer processes.*

Clearly Hammer utters an untenable dogma. We ma-u also need to re-engineer such phenomena as simple entities and functions.

3. *Understanding the process is an essential first step in re-engineering.*

And then he goes on to say: *“but an analysis of those processes is a waste of time. You must place strict limits, both on time you take to develop this understanding and on the length of the description you make.”* Needless to say we question this latter part of the third item.

4. *If you proceed to re-engineer without the proper leadership, you are making a fatal mistake. If your leadership is nominal rather than serious, and isn’t prepared to make the required commitment, your efforts are doomed to failure.*

By leadership is basically meant: “upper, executive management”.

---

[1] Domain requirements are in conventional textbooks referred to as functional requirements

[2] Interface requirements are in conventional textbooks referred, more-or-less, to as user requirements.

[3] Machine requirements are in conventional textbooks referred, more-or-less, to as system requirements.

5. *Re-Engineering requires radical, breakthrough ideas about process design. Re-Engineering leaders must encourage people to pursue stretch goals*[4] *and to think out of the box; to this end, leadership must reward creative thinking and be willing to consider any new idea.* s672

This is clearly an example of the US guru, "new management"-type 'speak'!

6. *Before implementing a process in the real world create a laboratory version in order to test whether your ideas work. ... Proceeding directly from idea to real-world implementation is (usually) a recipe for disaster.*

Our careful both informal and formal description of the existing domain processes, as covered in Sect. 7, as well as the similarly careful prescription of the re-engineered business processes shall, in a sense, make up for this otherwise vague term "laboratory version".

7. *You must re-engineer quickly. If you can't show some tangible results within a year, you will lose the support and momentum necessary to make the effort successful. To this end "scope creep" must be avoided at all cost. Stay focused and narrow the scope if necessary in order to get results fast.*

We obviously do not agree, in principle and in general, with this statement.

8. *You cannot re-engineer a process in isolation. Everything must be on the table. Any attempts to set limits, to preserve a piece of the old system, will doom your efforts to failure.*

We can only agree. But the wording is like mantras. As a software engineer, founded in science, such statements as the above are not technical, are not scientific. They are "management speak".

9. *Re-Engineering needs its own style of implementation: fast, improvisational, and iterative.*

We are not so sure about this statement either! Professional engineering work is something one neither does fast nor improvisational.

10. *Any successful re-engineering effort must take into account the personal needs of the individuals it will affect. The new process must offer some benefit to the people who are, after all, being asked to embrace enormous change, and the transition from the old process to the new one must be made with great sensitivity as to their feelings.*

This is nothing but a politically correct, pat statement! It would not pass the negation test: Nobody would claim the opposite. Real benefits of re-engineering often come from not requiring as many people, i.e., workers and management, in the corporation as before re-engineering. Hence: What about the "feelings" of those laid off?

8.4.2 What Are BPR Requirements? s673

Two "paths" lead to business process re-engineering:

- A client wishes to improve enterprise operations by deploying new computing systems (i.e., new software). In the course of formulating requirements for this new computing system a need arises to also re-engineer the human operations within and without the enterprise.
- An enterprise wishes to improve operations by redesigning the way staff operates within the enterprise and the way in which customers and staff operate across the enterprise-to-environment interface. In the course of formulating re-engineering directives a need arises to also deploy new software, for which requirements therefore have to be enunciated.

One way or the other, business process re-engineering is an integral component in deploying new computing systems.

[4] A 'stretch goal' is a goal, an objective, for which, if one wishes to achieve that goal, one has to stretch oneself.

8.4.3 Overview of BPR Operations s674

We suggest six domain-to-business process re-engineering operations. They are based on the facets that were prominent in the process of constructing a domain description.

1. introduction of some new and removal of some old **intrinsics;**
2. introduction of some new and removal of some old **support technologies;**
3. introduction of some new and removal of some old **management and organisation substructures;**
4. introduction of some new and removal of some old **rules and regulations;**
5. related **scripting;** and
6. introduction of some new and removal of some old work practices (relating to **human behaviours**);

8.4.4 BPR and the Requirements Document s675

Requirements for New Business Processes

The reader must be duly "warned": The BPR requirements are not for a computing system, but for the people who "surround" that (future) system. The BPR requirements state, unequivocally, how those people are to act, i.e., to use that system properly. Any implications, by the BPR requirements, as to concepts and facilities of the new computing system must be prescribed (also) in the domain and interface requirements.

Place in Narrative Document s676

We shall thus, in Sects. 8.4.5–8.4.9, treat a number of BPR facets. Each of whatever you decide to focus on, in any one requirements development, must be prescribed. And the prescription must be put into the overall requirements prescription document.

s677

As the BPR requirements "rebuilds" the business process description part of the domain description[5], and as the BPR requirements are not directly requirements for the machine, we find that they (the BPR requirements texts) can be simply put in a separate section.

s678

There are basically two ways of "rebuilding" the domain description's business process's description part (D_{BP}) into the requirements prescription part's BPR requirements (R_{BPR}). Either you keep all of D as a base part in R_{BPR}, and then you follow that part (i.e., R_{BPR}) with statements, R'_{BPR}, that express the new business process's "differences" with respect to the "old" (D_{BP}). Call the result R_{BPR}. s679Or you simply rewrite (in a sense, the whole of) D_{BP} directly into R_{BPR}, copying all of D_{BP}, and editing wherever necessary.

Place in Formalisation Document s680

The above statements as how to express the "merging" of BPR requirements into the overall requirements document apply to the narrative as well as to the formalised prescriptions.

Principle 6 – **Documentation:** *We may assume that there is a formal domain description, $\mathcal{D}_{BP}$, (of business processes) from which we develop the formal prescription of the BPR requirements.* s681 *We may then decide to either develop entirely new descriptions of the new business processes, i.e., actually prescriptions for the business re-engineered processes, $\mathcal{R}_{BPR}$; or develop, from $\mathcal{D}_{BP}$, using a suitable schema calculus, such as the one in* `RSL`*, the requirements prescription $\mathcal{R}_{BPR}$, by suitable parameterisation, extension, hiding, etc., of the domain description $\mathcal{D}_{BP}$.* ∎

[5]— Even if that business process description part of the domain description is "empty" or nearly so!

8.4.5 Intrinsics Review and Replacement s682

Definition 61 – **Intrinsics Review and Replacement:** *By intrinsics review and replacement we understand an evaluation as to whether current intrinsics stays or goes, and as to whether newer intrinsics need to be introduced.* ▪

s683

Example 71 – **Intrinsics Replacement:** A railway net owner changes its business from owning, operating and maintaining railway nets (lines, stations and signals) to operating trains. Hence the more detailed state changing notions of rail units need no longer be part of that new company's intrinsics while the notions of trains and passengers need be introduced as relevant intrinsics. ▪

Replacement of intrinsics usually point to dramatic changes of the business and are usually not done in connection with subsequent and related software requirements development.

8.4.6 Support Technology Review and Replacement s684

Definition 62 – **Support Technology Review and Replacement:** *By support technology review and replacement we understand an evaluation as to whether current support technology as used in the enterprise is adequate, and as to whether other (newer) support technology can better perform the desired services.* ▪

s685

Example 72 – **Support Technology Review and Replacement:** Currently the main information flow of an enterprise is taken care of by printed paper, copying machines and physical distribution. All such documents, whether originals (masters), copies, or annotated versions of originals or copies, are subject to confidentiality. As part of a computerised system for handling the future information flow, it is specified, by some domain requirements, that document confidentiality is to be taken care of by encryption, public and private keys, and digital signatures. However, it is realised that there can be a need for taking physical, not just electronic, copies of documents. s686The following business process re-engineering proposal is therefore considered:Specially made printing paper and printing and copying machines are to be procured, and so are printers and copiers whose use requires the insertion of special signature cards which, when used, check that the person printing or copying is the person identified on the card, and that that person may print the desired document. All copiers will refuse to copy such copied documents — hence the special paper. Such paper copies can thus be read at, but not carried outside the premises (of the printers and copiers). And such printers and copiers can register who printed, respectively who tried to copy, which documents. Thus people are now responsible for the security (whereabouts) of possible paper copies (not the required computing system). s687The above, somewhat construed example, shows the "division of labour" between the contemplated (required, desired) computing system (the "machine") and the "business re-engineered" persons authorised to print and possess confidential documents.

It is implied in the above that the re-engineered handling of documents would not be feasible without proper computing support. Thus there is a "spill-off" from the business re-engineered world to the world of computing systems requirements. ▪

8.4.7 Management and Organisation Re-Engineering s688

Definition 63 – **Management and Organisation Re-Engineering:** *By management and organisation re-engineering we understand an evaluation as to whether current management principles and organisation structures as used in the enterprise are adequate, and as to whether other management principles and organisation structures can better monitor and control the enterprise.* ▪

s689

Example 73 – **Management and Organisation Re-Engineering:** A rather complete computerisation of the procurement practices of a company is being contemplated. Previously procurement was manifested in the following physically separate as well as designwise differently formatted paper documents: *requisition form, order form, purchase order, delivery inspection form, rejection and return form*, and *payment form*. The supplier had corresponding forms: *order acceptance and quotation form, delivery form, return acceptance form, invoice form, return verification form*, and *payment acceptance form*. The current concern is only the procurement forms, not the supplier forms. s690The proposed domain requirements are mandating that all procurer forms disappear in their paper version, that basically only one, the procurement document, represents all phases of procurement, and that order, rejection and return notification slips, and payment authorisation notes, be effected by electronically communicated and duly digitally signed messages that represent appropriate subparts of the one, now electronic procurement document. s691The business process re-engineering part may now "short-circuit" previous staff's review and acceptance/rejection of former forms, in favour of fewer staff interventions.

The new business procedures, in this case, subsequently find their way into proper domain requirements: those that support, that is monitor and control all stages of the re-engineered procurement process. ■

8.4.8 Rules and Regulations Re-Engineering s692

Definition 64 – **Rules and Regulation Re-Engineering:** *By rules and regulations re-engineering we understand an evaluation as to whether current rules and regulations as used in the enterprise are adequate, and as to whether other rules and regulations can better guide and regulate the enterprise.* ■

Here it should be remembered that rules and regulations principally stipulate business engineering processes. That is, they are — i.e., were — usually not computerised.

s693

Example 74 – **Rules and Regulations Re-Engineering:** Our example continues that of Example 60 on page 97. We kindly remind the reader to re-study that example. Assume now, due to re-engineered support technologies, that interlock signalling can be made magnitudes safer than before, without interlocking. Thence it makes sense to re-engineer the rule of Example 60 from: *In any three-minute interval at most one train may either arrive to or depart from a railway station* into: *In any 20-second interval at most two trains may either arrive to or depart from a railway station.*

This re-engineered rule is subsequently made into a domain requirements, namely that the software system for interlocking is bound by that rule. ■

8.4.9 Script Re-Engineering s694

On one hand, there is the engineering of the contents of rules and regulations, and, on another hand, there are the people (management, staff) who script these rules and regulations, and the way in which these rules and regulations are communicated to managers and staff concerned.

s695

Definition 65 – **Script Re-Engineering:** *By script re-engineering we understand evaluation as to whether the way in which rules and regulations are scripted and made known (i.e., posted) to stakeholders in and of the enterprise is adequate, and as to whether other ways of scripting and posting are more suitable for the enterprise.* ■

s696

Example 75 – **Health-care Script Re-Engineering:** We refer to Example 63 (Pages 103–105). Let us assume that the situation before this business re-engineering process starts, in relation to hospital health-care, was that there was no physically visible notion of a health-care license language, but the the requirements now calls for such a language to be introduced with as much computer & communication support as is reasonable and that the hospital(s) in question are to become "paper-less". Now we can foresee a number of business process re-engineering based on the concept that such a health-care license language has been designed. s697For every action performed by a medical staff, whether an admittance, annamnese, planning analysis, carrying out our analysis, diagnostics, treatment planning, treatment (in all forms), et cetera, action (cf. Fig. 7.10 on page 104), there now has to be prescribed, by and for the hospital health-care staff a BPR prescription, which outlines what the staff members must do in preparation of the action, the action (probably nothing new here), and in concluding the action so that the medical staff performs the necessary "chores" assumed by the health-care license language software. ■

8.4.10 Human Behaviour Re-Engineering s698

Definition 66 – **Human Behaviour Re-Engineering:** *By human behaviour re-engineering we understand an evaluation as to whether current human behaviour as experienced in the enterprise is acceptable, and as to whether partially changed human behaviours are more suitable for the enterprise.* ■

s699

Example 76 – **Human Behaviour Re-Engineering:** A company has experienced certain lax attitudes among members of a certain category of staff. The progress of certain work procedures therefore is re-engineered, implying that members of another category of staff are henceforth expected to follow up on the progress of "that" work.

In a subsequent domain requirements stage the above re-engineering leads to a number of requirements for computerised monitoring of the two groups of staff. ■

8.4.11 A Specific Example of BPR s700

Example 77 – **A Toll-road System (I):**

Example 10 (Pages 9–17) outline a generic model of a domain of roads (links) and their intersections (hubs). We shall base some of the requirements examples of Sect. 8 on an instantiation of that domain model (Example 10) to a specific toll-road system. In this example we shall rough sketch that toll-road system. First we refer to Fig. 8.1.

s701

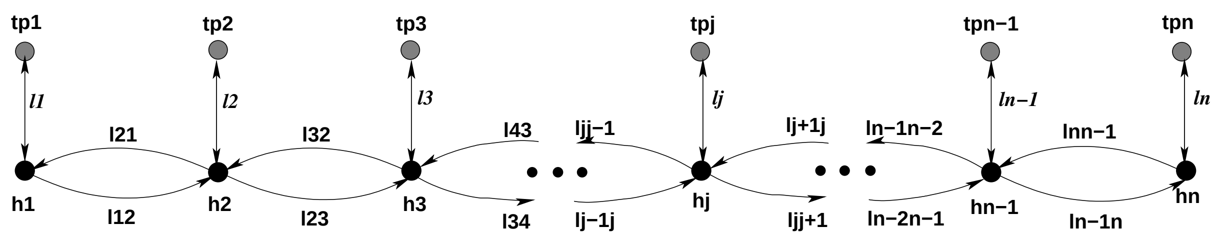


Fig. 8.1. A simple, linear toll road net:
tp_i: tboll plaza i,
ti_1, ti_n: terminal (or toll plaza) intersection k,
h_k: intermediate intersection (hub) k, $1<k<n$
ℓ_i: toll plaza link i,
l_{xy}: toll-way link from i_x to i_y, y=x+1 or y=x-1 and 1≤x<n.

s702

We first explain the kind of toll-roads semi-generically hinted at by Fig. 8.1 on the previous page. The core of the (semi-generic) toll-road is the "linear" stretch of pairs of one-way links $(\ell_{j_{j+1}}, \ell_{j+1_j})$ between adjacent hubs h_j and h_{j+1}. This is the actual toll-road. In order to enter and leave the toll-road there are entries and exits. These are in the form of toll plazas tp_i. Simple two-way links, ℓ_j, connect toll plaza tp_j (via toll plaza intersection t_{ij} to toll-road intersection h_j.

s703

We must here state that toll plazas are equipped with toll booths for cars entering the toll-road system and for cars leaving the toll-road system.

The basic business process of this toll-road system includes (i) the maintenance of all roads, intersections and toll plaza toll booths; (ii) the travel, through the toll-road system of a toll-paying car; and (iii) the monitoring and control of toll-paying car traffic within the toll-road system.

s704

We shall only summarise the travel (i) business process: (1) A car enters the toll-road system at toll plaza tp_i. (2) A toll booth gate prevents the car from fully entering the system. (3) The car is issued a toll slip by an entry toll-booth at toll plaza tp_i. (4) The toll-booth gate allows the car from entering the system. (5) The car travels along toll plaza link ℓ_j to toll-road hub h_j. [Let, in the following h_j now be h_i.] s705 (6) At toll-road hub h_i the car decides whether to continue driving along the toll-road proper or to leave the toll-road system along toll plaza link ℓ_j to toll plaza tp_i. In the latter case the business process description continues with Item (8). (7) The car travels, along toll-road link either $\ell_{i_{i+1}}$ or $\ell_{i_{i-1}}$, between toll-road hub h_i and hub h_{i+1} respectively hub h_{i+-}. [Let, in the following this target hub now be h_i.] The next business process step is now described in Item (6). (8) At toll plaza tp_i the car enters an exit toll-booth. (9) A toll-booth gate prevents the car from leaving the toll-road system. s706 (10) The previously issued toll slip is now presented at the toll-booth. (11) The toll-booth calculates the fare from entry plaza to exit plaza[6]. (13) The car (driver) somehow[7] (14) The toll-booth gate allows the car to leave the toll-road system. (15) The car leaves the toll-road system. This ends Example 77 ■

8.4.12 Discussion: Business Process Re-Engineering s707

Who Should Do the Business Process Re-Engineering?

It is not in our power, as software engineers, to make the kind of business process re-engineering decisions implied above. Rather it is, perhaps, more the prerogative of appropriately educated, trained and skilled (i.e., gifted) other kinds of engineers or business people to make the kinds of decisions implied above. Once the BP re-engineering has been made, it then behooves the client stakeholders to further decide whether the BP re-engineering shall imply some requirements, or not.

s708

Once that last decision has been made in the affirmative, we, as software engineers, can then apply our abstraction and modelling skills, and, while collaborating with the former kinds of professionals, make the appropriate prescriptions for the BPR requirements. These will typically be in the form of domain requirements, which are covered extensively in Sect. 8.6.

General s709

Business process re-engineering is based on the premise that corporations must change their way of operating, and, hence, must "reinvent" themselves. Some corporations (enterprises, businesses, etc.) are "vertically" structured along functions, products or geographical regions. This often means that business processes "cut across" vertical units. Others are "horizontally" structured

[6]We shall not detail this calculation. Its proper calculation may involve that the system has traced the car's passage through all hubs.

[7]We shall not detail that "somehow": whether it is by cash payment, via credit card, or by means of a toll-road system credit mechanism "built-into" the car.

along coherent business processes. This often means that business processes "cut across" functions, products or geographical regions. In either case adjustments may need to be made as the business (i.e., products, sales, markets, etc.) changes. We otherwise refer to currently leading books on business process re-engineering: [65, 66, 81, 86].

8.5 Requirements Acquisition s710

acm-rtre-a

Requirements 'reside' in the domain. That means: one can not possibly utter a reasonably comprehensive set of requirements without stating the domain "to which they apply". Therefore we first describe the domain before we next prescribe the requirements. And therefore we shall "base our requirements acquisition" on a supposedly existing domain description.

s711

To *'base our requirements acquisition ... etc.'* shall mean that we carefully go through the domain description (found most appropriate for the requirements at hand) with the requirements stake-holders asking them a number of questions. Which these questions are will be dealt with soon.

For domain acquisition there were, in principle, no prior domain description documents, really, to refer to. Hence an elaborate set of procedures had to be followed in order to solicit and elicit domain acquisition units. Before such elicitation could be done in any systematic fashion the domain engineer had to study the domain, by whatever informal means available. Now there is the domain description.

s712

From a purely linguistic point of view we can think of decomposing requirements acquisition relative to the domain description along three axes: the first axis of domain requirements — being those which can be expressed sôlely using terms from the domain; the second axis of machine requirements — being those which can be expressed sôlely using terms from the machine; and the third axis of interface requirements — being those which can be expressed using terms from both the domain and the machine.

The next three sections, Sects. 8.6–8.9, shall therefore be structured into two parts: the respective requirements acquisition part and the corresponding requirements modelling part.

8.6 Domain Requirements s713

acm-rtre-b

8.6.1 A Small Domain Example

In exemplifying several of the very many kinds of domain and machine requirements we need a small domain description. This small domain description will be that of a timetable, cf. Example 78. The sequence of machine requirements examples based on Example 78 will, furthermore, be expressed using the **scheme** and **class** modularisation constructs of RSL.

s714 tt0

Example 78 – **A Domain Example: an 'Airline Timetable System':**

We choose a very simple domain: that of a traffic timetable, say flight timetable. In the domain you could, in "ye olde days", hold such a timetable in your hand, you could browse it, you could look up a special flight, you could tear pages out of it, etc. There was no end as to what you could do to such a timetable. So we will just postulate a sort, *TT*, of timetables.

s715

Airline customers, *client*s, in general, only wish to inquire a timetable (so we will here omit treatment of more or less "malicious" or destructive acts). But you could still count the number of digits "7" in the timetable, and other such ridiculous things. So we postulate a broadest variety of inquiry functions, *qu:QU*, that apply to timetables, *tt:TT*, and yield values, *val:VAL*.

s716

Specifically designated airline *staff* may, however, in addition to what a *client* can do, update the timetable. But, recalling human behaviours, all we can ascertain for sure is that update functions, *up:UP,* apply to timetables and yield two things: another, replacement timetable, *tt:TT,* and a result, *res:RES,* such as: *"your update succeeded"*, or *"your update did not succeed"*, etc. In essence this is all we can say for sure about the domain of timetable creations and uses.

s717

We can view the domain of the timetable, clients and staff as a behaviour which nondeterministically alternates (⌈⌉) between the *client* querying the timetable *client_0(tt),* and the *staff* updating the same *staff_0(tt).*

s718

```
scheme TI_TBL_0 =
  class
    type
      TT, VAL, RES
      QU = TT → VAL
      UP = TT → TT × RES
    value
      client_0: TT → VAL, client_0(tt) ≡ let q:QU in q(tt) end
      staff_0: TT → TT × RES, staff_0(tt) ≡ let u:UP in u(tt) end

      tim_tbl_0: TT → Unit
      tim_tbl_0(tt) ≡
            (let v = client_0(tt) in tim_tbl_0(tt) end)
          ⌈⌉ (let (tt′,r) = staff_0(tt) in tim_tbl_0(tt′) end)
  end
```

The *timetable* function, *tim_tbl*, is here seen as a never ending process, hence the type **Unit.** It nondeterministically[8] alternates between "serving" the clients and the staff. Either of these two non-deterministically chooses from a possibly very large set of queries, respectively updates. This ends Example 78 ■

8.6.2 Acquisition s719

acm-rtre-b

Common to the acquisition and modelling of domain requirements are the following sub-stages:

- projection,
- instantiation,
- determination,
- extension and
- fitting.

sub-stages.

With each and every stake-holder group the domain engineer(s) go through the domain description and asks the following questions:

- Which of the simple entities, functions, events and behaviour (parts) of the domain do you wish to be represented somehow in, i.e., projected onto the machine ?

s720

- Which of the simple entities, functions, events and behaviour (parts) of the domain do you wish to be less generic, more instantiated in the machine ?
- Which of the simple entities, functions, events and behaviour (parts) of the domain do you wish to appear more deterministic in the machine ?
- Are there simple entities, functions, events and behaviours that could be in the domain but are not there because their "existence" is not feasible — if so, with computing and communication are they now feasible and should the domain thus be extended ?

[8] The nondeterminism referred to is internal in the sense that no outside behaviour influences the choice.

- Given that there may be several, parallel ongoing requirements development for related parts of the domain, should they be fitted ?

s721

For each of these five sub-stages of domain requirements the acquisition consists in asking these questions and marking the domain description cum emerging domain requirements document with the answers:

- circling-in the domain description parts that are to be part of the domain requirements (i.e., projection)
- marking those parts with possible directives as to instantiation and determination;
- making adequate notes on possible extensions
- and fittings.

Once this domain requirements acquisition has taken place for all groups of stake-holders the requirements engineers can proceed to interface requirements acquisition, Sect. 8.8.1.

8.6.3 Projection

s722

Definition 67 – **Projection:** *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* ▪

s723 arms-projection

Example 79 – **Projection: A Road Maintenance System:** The requirements are for a road maintenance system. That is, maintenance of link and hub (road segment and road intersection) surfaces, the monitoring of their quality and road repair.

Instead of listing all the phenomena and concepts of the domain that are "projected away", we list those few that remain: hubs, links, hub identifiers and link identifiers; nets, corresponding observer functions, and corresponding axioms.

s724

type
  1: H, L,[9]
  2: N = H-**set** × L-**set**
axiom
  2: ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ks≥1
type
  3: HI, LI
value
  4a: obs_HI: H → HI, obs_LI: L → LI
axiom
  4b: ∀ h,h′:H, l,l′:L • h≠h′
      ⇒ obs_HI(h)≠obs_HI(h′) ∧ l≠l′⇒obs_LI(l)≠obs_LI(l′)

s725

value
  5a: obs_HIs: L → HI-**set**
  6a: obs_LIs: H → LI-**set**
  5b: ∀ l:L • **card** obs_HIs(l)=2 ∧
  6b: ∀ h:H • **card** obs_LIs(h)≥1 ∧
  5a: ∀ (hs,ls):N • ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ obs_LIs(h) ⇒

[9]Formula numbers refer to narrative text items as from Page 9 etc.

```
        ∃ l′:L • l′ ∈ ls ∧ li=obs_LI(l′) ∧ obs_HI(h) ∈ obs_HIs(l′) ∧
6a:   ∀ l:L • l ∈ ls ⇒
        ∃ h′,h″:H • {h′,h″}⊆hs ∧ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}
7:     ∀ h:H • h ∈ hs ⇒ obs_LIs(h) ⊆ iols(ls)
8:     ∀ l:L • l ∈ ls ⇒ obs_HIs(h) ⊆ iohs(hs)
   iohs: H-set → HI-set, iols: L-set → LI-set
   iohs(hs) ≡ {obs_HI(h)|h:H•h ∈ hs}
   iols(ls) ≡ {obs_LI(l)|l:L•l ∈ ls}
```

This ends Example 79 ■

acm-rtre-b
atrs-projection
s726

Example 80 – **Projection: A Toll-road System:** For the 'Toll-road System', as outlined in Example 77, in addition to what was projected for the 'Road Maintenance System' of Example 79, the following entities and most related functions are projected: hubs, links, hub and link identifiers; nets, that is, hub state and hub state spaces and link states and link state spaces, corresponding observer functions, corresponding axioms and syntactic and semantic wellformedness predicates.

s727

```
type
   LΣ′ = L_Trav-set
   L_Trav = (HI × LI × HI)
   LΣ = {| lnkσ:LΣ′ • syn_wf_LΣ{lnkσ} |}
   HΣ′ = H_Trav-set
   H_Trav = (LI × HI × LI)
   HΣ = {| hubσ:HΣ′ • wf_HΣ{hubσ} |}
   HΩ = HΣ-set, LΩ = LΣ-set
value
   obs_HΣ: H → HΣ, obs_LΣ: L → LΣ
   obs_HΩ: H → HΩ, obs_LΩ: L → LΩ
axiom
   ∀ h:H • obs_HΣ(h) ∈ obs_HΩ(h) ∧ ∀ l:L • obs_LΣ(l) ∈ obs_LΩ(l)
```

For hubs, links, hub identifiers and link identifiers and nets see the above projection; and for the missing axioms and wellformedness predicates see Example 58 (Pages 89–91).

This ends Example 80 ■

8.6.4 Instantiation s728

acm-rtre-b

Definition 68 – **Instantiation:** *By domain instantiation we understand an operation that applies to a (projected and possibly determined) 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* ■

arms-instantiation
s729

Example 81 – **Instantiation: A Road Maintenance System:** We continue Example 79.

11. The road net consist of a sequence of one or more road segments.
12. A road segment can be characterised by a pair of hubs and a pair of links connected to these hubs.
13. Neighbouring road segments share a hub.
14. All hubs are otherwise distinct.
15. All links are distinct.
16. The two links of a road segment connects to the hubs of the road segment.
17. We can show that road nets are specific instances of concretisations of the former, thus more abstract road nets.

s730

```
type
11 RN = RS*,
12 RS = H × (L × L) × H
axiom
   ∀ rn:RN •
13   ∀ i:Nat • {i,i+1}⊆inds rn ⇒ let (_,_,h)=rn(i),(h′,_,_)=rn(i+1) in h=h′ end ∧
14      len rn + 1 = card{h,h′|h,h′:H•(h,_,h′)∈ elems rn} ∧
15      2∗(len rn) = card{l,l′|l,l′:L•(_,(l,l′),_)∈ elems rn} ∧
16      ∀ (h,(l,l′),h′):RS • (h,(l,l′),h′) ∈ elems rn ⇒
               obs_Σ(l)={(obs_HI(h),obs_HI(h′))} ∧ obs_Σ(l′)={(obs_HI(h′),obs_HI(h))}
value
17   abs_N: RN → N
     abs_N(rsl) ≡
        ({h,h′|(h,_,h′):RS • (h,_,h′) ∈ elems rsl},{l,l′|(_,(,l,l′),_):RS • (_.(l,l′),_) ∈ elems rsl})
```

This ends Example 81 ■

s731 acm-rtre-b atrs-instantiation

Example 82 – **Instantiation: A Toll-road System:** We continue Example 80. The 1st version domain requirements prescription, Example 80, is now updated with respect to the properties of the toll-road net: We refer to Fig. 8.1 on page 139 and the preliminary description given in Example 77. There are three kinds of hubs: tollgate hubs and intersection hubs: terminal intersection hubs and proper, intermediate intersection hubs. Tollgate hubs have one connecting two way link. linking the tollgate hub to its associated intersection hub. s732 Terminal intersection hubs have three connecting links: (i) one, a two-way link, to a tollgate hub, (ii) one one-way link emanating to a next up (or down) intersection hub, and (iii) one one-way link incident upon this hub from a next up (or down) intersection hub. Proper intersection hubs have five connecting links: one, a two way link, to a tollgate hub, two one way links emanating to next up and down intersection hubs, and two one way links incident upon this hub from next up and down intersection hub. (Much more need be narrated.) As a result we obtain a 2nd version domain requirements prescription.

s733

```
type
   TN = ((H × L) × (H × L × L))* × H × (L × H)
value
   abs_N: TN → N
   abs_N(tn) ≡ (tn_hubs(tn),tn_hubs(tn))
   pre wf_TN(tn)

   tn_hubs: TN → H-set,
   tn_hubs(hll,h,(_,hn)) ≡
      {h,hn} ∪ {thj,hj|((thj,tlj),(hj,lj,lj′)):((H×L)×(H×L×L))•((thj,tlj),(hj,lj,lj′))∈ elems hll}
   tn_links: TN → L-set
   tn_links(hll,_,(ln,_)) ≡
      {ln} ∪ {tlj,lj,lj′|((thj,tlj),(hj,lj,lj′)):((H×L)×(H×L×L))•((thj,tlj),(hj,lj,lj′))∈ elems hll}

theorem ∀ tn:TN • wf_TN(tn) ⇒ wf_N(abs_N(tn))
```

s734

```
type
   LnkM == plaza | way
value
```

```
wf_TN: TN → Bool
wf_TN(tn:(hll,h,(ln,hn))) ≡
    wf_Toll_Lnk(h,ln,hn)(plaza) ∧ wf_Toll_Ways(hll,h) ∧
    wf_State_Spaces(tn) [to be defined under Determination]
wf_Toll_Ways: ((H×L)×(H×L×L))* × H → Bool
wf_Toll_Ways(hll,h) ≡
   ∀ j:Nat • {j,j+1}⊆inds hll ⇒
      let ((thj,tlj),(hj,ljj′,lj′j)) = hll(j),(_,(hj′,_,_)) = hll(j+1) in
      wf_Toll_Lnk(thj,tlj,hj)(plaza) ∧
      wf_Toll_Lnk(hj,ljj′,hj′)(way) ∧ wf_Toll_Lnk(hj′,lj′j,hj)(way) end ∧
      let ((thk,tlk),(hk,lk,lk′)) = hll(len hll) in
      wf_Toll_Lnk(thk,tlk,hk)(plaza) ∧
      wf_Toll_Lnk(hk,lk,hk′)(way) ∧ wf_Toll_Lnk(hk′,lk′,hk)(way)  end
```

s735

```
wf_Toll_Lnk: (H×L×H) → LnkM → Bool
wf_Toll_Lnk(h,l,h′)(m) ≡
   obs_Ps(l)={(obs_HI(h),obs_LI(l),obs_HI(h′)),(obs_HI(h′),obs_LI(l),obs_HI(h))} ∧
   obs_Σ(l) = case m of
                   plaza → obs_Ps(l),
                   way → {(obs_HI(h),obs_LI(l),obs_HI(h′))} end
```

This ends Example 82 ■

8.6.5 Determination s736

acm-rtre-b

Definition 69 – **Determination:** *By domain determination 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 made deterministic, or specific, some function results or some behaviours of the former* ■

s737
tt1-dd
tt0-dd

Example 83 – **Timetable System Determination:** We make airline timetables more specific, more deterministic. There are given notions of departure and arrival times, and of airports, and of airline flight numbers. s738

```
scheme TI_TBL_2 =
   extend TI_TBL_1 with
      class
         type
            T, An, Fn
      end
```

s739A timetable consists of a number of air flight journey entries. Each entry has a flight number, and a list of two or more airport visits. an airport visit consists of three parts: An airport name, and a pair of (gate) arrival and departure times. s740

```
scheme TI_TBL_3 =
   extend TI_TBL_2 with
      class
         type
            JR′ = (T × An × T)*
            JR = {| jr:JR′ • len jr ≥ 2 ∧ ... |}
            TT = Fn →m JR
      end
```

We illustrate just one, simple form of airline timetable queries. A simple airline timetable query either just browses all of an airline timetable, or inquires of the journey of a specific flight. s741 The simple browse query thus need not provide specific argument data, whereas the flight journey query needs to provide a flight number. A simple update query inserts a new pairing of a flight number and a journey to the timetable, whereas a delete query need just provide the number of the flight to be deleted.

s742

The result of a query is a value: the specific journey inquired, or the entire timetable browsed. The result of an update is a possible timetable change and either an "OK" response if the update could be made, or a "Not OK" response if the update could not be made: Either the flight number of the journey to be inserted was already present in the timetable, or the flight number of the journey to be deleted was not present in the timetable.

That is, we assume above that simple airline timetable queries only designate simple flights, with one aircraft. For more complex air flights, with stopovers and changes of flights, see Sect. 86 on page 150.

You may skip the rest of the example, its formalisation, if your reading of this paper does not include the various formalisations.

s743

First, we formalise the syntactic and the semantic types:

```
scheme TI_TBL_3Q =
   extend TI_TBL_3 with
      class
         type
            Query == mk_brow() | mk_jour(fn:Fn)
            Update == mk_inst(fn:Fn,jr:JR) | mk_delt(fn:Fn)
            VAL = TT
            RES == ok | not_ok
      end
```

Then we define the semantics of the query commands:

s744

```
scheme TI_TBL_3U =
   extend TI_TBL_3 with
      class
         value
            𝓜q: Query → QU
            𝓜q(qu) ≡
               case qu of
                  mk_brow() → λtt:TT•tt,
                  mk_jour(fn)
                     → λtt:TT • if fn ∈ dom tt
                                 then [fn↦tt(fn)] else [] end
      end   end
```

And, finally, we define the semantics of the update commands:

s745

```
scheme TI_TBL_3U =
   extend TI_TBL_3 with
      class
            𝓜u: Update → UP
            𝓜u(up) ≡
               case qu of
                  mk_inst(fn,jr) → λtt:TT •
```

```
                if fn ∈ dom tt
                   then (tt,not_ok) else (tt ∪ [fn↦jr],ok) end,
              mk_delt(fn) → λtt:TT •
                if fn ∈ dom tt
                   then (tt \ {fn},ok) else (tt,not_ok) end
    end   end
```

We can "assemble" the above into the timetable function — calling the new function the timetable system, or just the system function.

s746

Before we had:

```
value
   tim_tbl_0: TT → Unit
   tim_tbl_0(tt) ≡
          (let v = client_0(tt) in tim_tbl_0(tt) end)
       ⌈⌉ (let (tt′,r) = staff_0(tt) in tim_tbl_0(tt′) end)
```

Now we get:

```
value
   system: TT → Unit
   system() ≡
          (let q:Query in let v = M_q(q)(tt) in system(tt) end end)
       ⌈⌉ (let u:Update in let (r,tt′) = M_u(q)(tt) in system(tt′) end end)
```

s747 Or, for use in Example 99:

```
   system(tt) ≡ client(tt) ⌈⌉ staff(tt)

   client: TT → Unit
   client(tt) ≡
      let q:Query in let v = M_q(q)(tt) in system(tt) end end

   staff: TT → Unit
   staff(tt) ≡
      let u:Update in let (r,tt′) = M_u(q)(tt) in system(tt′) end end
```

We remind the reader that the above example can be fully understood by just reading the rough-sketch texts, that is, without reading their formalisations. This ends Example 83

■

s748 acm-rtre-b arms-determination

Example 84 – **Determination: A Road Maintenance System:** We continue Example 81.We shall, in this example, claim that the following items constitute issues of more determinate nature of the 'Road Management System' under development. fixing the states of links and hubs; endowing links and hubs with such attributes as road surface material (concrete, asphalt, etc.), state of road surface wear-and-tear, hub and link areas, say in m^2, time units needed for and cost of ordinary cleaning of m^2s of hub and link surface; time units needed for and cost of ordinary repairs of m^2s of hub and link surface; etcetera. s749

18. The two links of a road segment are open for traffic in one direction and in opposite directions only.
19. Hubs are always in the same state, namely one that allows traffic from incoming links to continue onto all outgoing links.
20. Hubs and Links have a number of attributes that allow for the monitoring and planning of hub and link surface conditions, i.e., whether in ordinary or urgent need of cleaning and/or repair.

s750

```
axiom
   ∀ rn:RN •
18 ∀ (h,(l,l′),h′):RS • (h,(l,l′),h′) ∈ elems rn ⇒
         obs_LΣl(l) = {(obs_Hl(h),obs_Ll(l),obs_Hl(h′))} ∧
         obs_LΣl(l′) = {(obs_Hl(h′),obs_Ll(l′),obs_Hl(h))} ∧
19 ∀ i:Nat • {i,i+1}⊆inds rn •
         let ((h,(l,l′),h′),(h′,(l′′,l′′′),h′′)) = (rn(i),rn(i+1)) in
         case i of
            1 →      obs_HΣ(h) = {(obs_Ll(l),obs_Hl(h),obs_Ll(l′))},
            len rn → obs_HΣ(h′) = {(obs_Ll(l′),obs_Hl(h′),obs_Ll(l))},
            _ →      obs_HΣ(h′)
                     = {(obs_Ll(l),obs_Hl(h′),obs_Ll(l′)),(obs_Ll(l),obs_Hl(h′),obs_Ll(l′))}
         end end
type
20 Surface, WearTear, Area, OrdTime, OrdCost, RepTime, RepCost, ...
value
20 obs_Surface: (H|L)→Surface, obs_WearTear: (H|L)→WearTear, ...
```

This ends Example 84 ■

s751 acm-rtre-b atrs-determination

Example 85 – **Determination: A Toll-road System:** We continue Example 82. We single out only two 'determinations': *The link state spaces.* There is only one link state: the set of all paths through the link, thus any link state space is the singleton set of its only link state. *The hub state spaces* are the singleton sets of the "current" hub states which allow these crossings: (i) from terminal link back to terminal link, (ii) from terminal link to emanating tollway link, (iii) from incident tollway link to terminal link, and (iv) from incident tollway link to emanating tollway link. Special provision must be made for expressing the entering from the outside and leaving toll plazas to the outside.

s752

```
wf_State_Spaces: TN → Bool
wf_State_Spaces(hll,hn,(thn,tln)) ≡
   let ((th1,tl1),(h1,l12,l21)) = hll(1),
       ((thk,ljk),(hk,lkn,lnk)) = hll(len hll) in
   wf_Plaza(th1,tl1,h1) ∧ wf_Plaza(thn,tln,hn) ∧
   wf_End(h1,tl1,l12,l21,h2) ∧ wf_End(hk,tln,lkn,lnk,hn) ∧
   ∀ j:Nat • {j,j+1,j+2}⊆inds hll ⇒
      let (,(hj,ljj,lj′j)) = hll(j),((thj′,tlj′),(hj′,ljj′,lj′j′)) = hll(j+1) in
      wf_Plaza(thj′,tlj′,hj′) ∧ wf_Interm(ljj,lj′j,hj′,tlj′,ljj′,lj′j′) end end

wf_Plaza(th,tl,h) ≡
   obs_HΣ(th) = [ crossings at toll plazas ]
      {(external,obs_Hl(th),obs_Ll(tl)),
         (obs_Ll(tl),obs_Hl(th),external),
         (obs_Ll(tl),obs_Hl(th),obs_Ll(tl))} ∧
       obs_HΩ(th)={obs_HΣ(th)} ∧
       obs_LΩ(tl) = {obs_LΣ(tl)}
```

s753

```
wf_End(h,tl,l,l′) ≡
   obs_HΣ(h) = [ crossings at 3−link end hubs ]
      {(obs_Ll(tl),obs_Hl(h),obs_Ll(tl)),(obs_Ll(tl),obs_Hl(h),obs_Ll(l)),
       (obs_Ll(l′),obs_Hl(h),obs_Ll(tl)),(obs_Ll(l′),obs_Hl(h),obs_Ll(l))} ∧
```

obs_HΩ(h) = {obs_HΣ(h)} ∧
obs_LΩ(l) = {obs_LΣ(l)} ∧ obs_LΩ(l′) = {obs_LΣ(l′)}

wf_Interm(ul_1,dl_1,h,tl,ul,dl) ≡
obs_HΣ(h) = {[crossings at properly intermediate, 5−link hubs]
(obs_LI(tl),obs_HI(h),obs_LI(tl)),(obs_LI(tl),obs_HI(h),obs_LI(dl_1)),
(obs_LI(tl),obs_HI(h),obs_LI(ul)),(obs_LI(ul_1),obs_HI(h),obs_LI(tl)),
(obs_LI(ul_1),obs_HI(h),obs_LI(ul)),(obs_LI(ul_1),obs_HI(h),obs_LI(dl_1)),
(obs_LI(dl),obs_HI(h),obs_LI(tl)),(obs_LI(dl),obs_HI(h),obs_LI(dl_1)),
(obs_LI(dl),obs_HI(h),obs_LI(ul))} ∧
obs_HΩ(h) = {obs_HΣ(h)} ∧ obs_LΩ(tl) = {obs_LΣ(tl)} ∧
obs_LΩ(ul) = {obs_LΣ(ul)} ∧ obs_LΩ(dl) = {obs_LΣ(dl)}

Not all determinism issues above have been fully explained. But for now we should — in principle — be satisfied. This ends Example 85 ■

8.6.6 Extension s754

acm-rtre-b

Definition 70 – **Extension:** *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, an operation that is not only feasible but also computable in reasonable time* ■

s755
tt1-de

Example 86 – **Timetable System Extension:**
We assume a projected and instantiated timetable (see Sect. 83 on page 146).

A query of a timetable may, syntactically, specify an airport of origin, a_o, an airport of destination, a_d, and a maximum number, n, of intermediate stops. The query semantically designates the set of all those trips of one up to n direct air journeys between a_o and a_d, i.e., trips where the passenger may change flights (up to $n-1$ times) at intermediate airports.

s756

scheme TI_TBL_3C =
extend TI_TBL_3 **with**
class
type
Query′ == Query | mk_conn(fa:An,ta:An,n:**Nat**)
VAL′ = VAL | CNS
CNS = (JR*)-**set**
value
$\mathcal{M}_q$(mk_conn(fa,ta,n)) **as**
pre ...
post ...
end

Here we leave it to the reader to define the “connections” function! At present you need not be concerned with the fact that TI_TBL_3C does not include the timetable initialisation command. To secure that we need to “juggle” some of the previously defined TI_TBL_x schemes. We omit showing this.

This ends Example 86 ■

s757
atrs-extension

Example 87 – **Extension: A Toll-road System:** We continue Examples 77 and 85. In the rough sketch of the toll-road business processes (Example 77) references were made to a concept of a toll-booth.

The domain extension is that of the controlled access of vehicles to and departure from the toll road net: the entry to (and departure from) tollgates from (respectively to) an "an external" net — which we do not describe; the new entities of tollgates with all their machinery; s758 the user/machine functions: upon entry: driver pressing entry button, tollgate delivering ticket; upon exit: driver presenting ticket, tollgate requesting payment, driver providing payment, etc.

One added (extended) domain requirements: as vehicles are allowed to cruise the entire net payment is a function of the totality of links traversed, possibly multiple times. s759 This requires, in our case, that tickets be made such as to be sensed somewhat remotely, and that intersections be equipped with sensors which can record and transmit information about vehicle intersection crossings. (When exiting the tollgate machine can then access the exiting vehicles sequence of intersection crossings — based on which a payment fee calculation can be done.)

All this to be described in detail — including all the thinks that can go wrong (in the domain) and how drivers and tollgates are expected to react.

We suggest only some signatures:

s760

type
  Mach, Ticket, Cash, Payment, Map_TN
value
  obs_Cash: Mach → Cash, obs_Tickets: M → Ticket-**set**
  obs_Entry, obs_Exit: Ticket → HI, obs_Ticket: V → (Ticket|nil)
  calculate_Payment: (HI×HI) → Map_TN → Payment
  press_Entry: M → M × Ticket [gate up]
  press_Exit: M × Ticket → M × Payment
  payment: M × Payment → M × Cash [gate up]

This ends Example 87 ■

8.6.7 Fitting s761

acm-rtre-b

Definition 71 – **Fitting:** *By domain requirements fitting we understand an operation that applies to two or more, say m, projected and possibly determined, instantiated and extended domain descriptions, i.e., to two or more, say m, original domain requirements prescriptions, and yields $m+n$ (resulting, revised original plus new, shared) domain requirements prescriptions. The m revised original domain requirements prescriptions resulting from the fitting prescribe most of the original (m) domain requirements. The n (new, shared) domain requirements prescriptions resulting from the fitting prescribe requirements that are shared between two or more of the m revised original domain requirements* ■

s762 arms+atrs-fitting

Example 88 – **Fitting: Road Maintenance and Toll-road Systems:** We end the series of examples that illustrate requirements for a road maintenance respectively a toll-road system (Examples 77–87). We postulate two domain requirements: We have outlined a domain requirements development for software support for road maintenance; and we have outlined a domain requirements development for software support for a toll-road system.

s763

We can therefore postulate that there are two domain requirements developments, both based on the transport domain: one, $d_{r_{\text{road-maint.}}}$, for a toll road computing system monitoring and controlling vehicle flow in and out of toll plazas, and another, $d_{r_{\text{toll-road}}}$, for a toll link and intersection (i.e., hub) building and maintenance system monitoring and controlling link and hub quality and for development.

s764

The fitting procedure now identifies the shared of awareness of the net by both $d_{r_{\text{road-maint.}}}$ and $d_{r_{\text{toll-road}}}$ of nets (N), hubs (H) and links (L). We conclude from this that we can single out a common requirements for software that manages net, hubs and links. Such software requirements basically amounts to requirements for a database system. A suitable such system, say a relational database management system, DBrel, may already be available with the customer.

s765

In any case, where there before were two requirements ($d_{r_{\text{road-maint.}}}, d_{r_{\text{toll-road}}}$) there are now four: (i) $d'_{r_{\text{road-maint.}}}$, a modification of $d_{r_{\text{road-maint.}}}$ which omits the description parts pertaining to the net; (ii) $d'_{r_{\text{toll-road}}}$, a modification of $d_{r_{\text{toll-road}}}$ which likewise omits the description parts pertaining to the net; (iii) $d_{r_{\text{net}}}$, which contains what was basically omitted in $d'_{r_{\text{road-maint.}}}$ and $d'_{r_{\text{toll-road}}}$; and (iv) $d_{r_{\text{db:i/f}}}$ (for database interface) which prescribes a mapping between type names of $d_{r_{\text{net}}}$ and relation and attribute names of DB_{rel}.

Much more can and should be said, but this suffices as an example. This ends Example 88 ■

8.7 A Caveat: Domain Descriptions versus Requirements Prescriptions s766

acm-rtre-b

8.7.1 Domain Phenomena

When in the domain we describe simple entities by:

type L, H, N

then we mean that L, H and N denote types of real, actually in the domain occurring phenomena l:L, h:H and n:N (as here, from Example 10, links, hubs and nets).

8.7.2 Requirements Concepts s767

When, however, in the requirements, we describe simple entities by the same identifiers then we mean that L, H and N denote types of representation of domain phenomena l:L, h:H and n:N, not the "the real thing", but "only" representations thereof.

8.7.3 A Possible Source of Confusion s768

We have decided not to make a syntactic distinction between these two kinds of (simple entity, operation, event and behaviour) names. The context, that is, the fact that such names occur in a section on requirements, is enough, we think, to make the distinction clear. When there can be doubt, as we shall see in the next section, on Interface Requirements (Sect. 8.8), then we shall "spell out" the difference, viz., L (domain) versus LINK (requirements).

8.7.4 Relations of Requirements Concepts to Domain Phenomena s769

We did not bother to warn the reader, in Sects. 8.6.3–8.6.7, about the possible source of confusion that lies in mistaking a requirements concepts for "the real thing": its domain phenomena "counterpart". But we find that it is high time now, before we enter the section on 'Interface Requirements' (just below), to highlight that the simple entities, operations, events and behaviours referred to in requirements are concept whereas those of domains are phenomena.

s770

The reason (for now emphasising the difference) is simple: interface requirements are about the relations between phenomena of the domain and concepts of the software (being required).

When, in the domain, we name a (simple entity, operation, event or behaviour) phenomena $\mathcal{D}$, and when in the requirements, we name a corresponding (simple entity, operation, event or behaviour) $\mathcal{R}$; then, by corresponding, we mean that there is an unprovable relation

$$\mathcal{R} \models \mathcal{D}.$$

s771

We cannot possibly formally claim that

$$\mathcal{R} = \mathcal{D} \quad \text{or even} \quad \mathcal{R} \simeq \mathcal{D},$$

The $\mathcal{R}$ is a mathematical model of some requirements concept whereas $\mathcal{D}$ is thought of as "the real thing". Let us understand the above, seemingly contradictory statement: The $\mathcal{D}$ is expressed mathematically, so it must be conceptual, as is $\mathcal{R}$. Therefore they ought be comparable. If we take this view that both are mathematical models, then all is OK and we can compare them. If, however, we take the view that the names (of what is assumed, or claimed, to be domain phenomena in $\mathcal{D}$) denote "the real things, out there in the actual world", then we cannot compare them.

s772How do we, in the following, reconcile these two views ? We do so as follows: On one hand we write

$$\mathcal{R} \models \mathcal{D}$$

to mean that the requirements abstractly models the domain, while, when we write

$$\mathcal{R} \quad \textsf{abstractly refines} \quad \mathcal{D}$$

or abs_D($\mathcal{R}$eq) = $\mathcal{D}$om we mean that the mathematical model of the requirements is a refinement of the mathematical model of the domain — in which latter phenomena names are considered names of mathematical concepts.

8.7.5 Sort versus Type Definitions s773

As a principle we prefer to use sorts and observer functions:

Example 89 – **Domain Types and Observer Functions:**

```
type
   N, L, H, LI, HI, Location, Length
value
   obs_Ls: N → L-set, obs_Hs: N → H-set
   obs_LI: L → LI, obs_HI: H → HI
   obs_LIs: H → LI-set, obs_HIs: L → HI-set
   obs_Location: L → Location, obs_Length: L → Real
```

■

rather then type definitions:

s774

Example 90 – **Requirements Types and Decompositions:**

```
type
   LI, HI, Location, Length
   N = L-set × H-set
   L = LI × (HI×HI) × Location × Length × ...
   H = HI × LI-set × Length × ...
value
   (ls,hs):N, (li,(fhi,thi),loc,len,...):L, (hi,lis,len,...):H
```

■

when defining simple domain entities. The type definitions are then typically introduced in requirements prescriptions. As shown in the last formula line of Example 90, the observer functions of domain descriptions can then be simply effected by decompositions.

Discussion s775

8.8 Interface Requirements s776

acm-rtre-c

8.8.1 Acquisition s777

Interface requirements acquisition evolves around a notion of shared phenomena and concepts of the domain. These are listed now.

- **Shared Simple Entities:** The shared simple entities are those simple entities that 'occur' in the domain but must also be represented by the machine.
- **Shared Operations:** The shared operations are those operations of the domain that can only be partially 'executed' by the machine.
- **Shared Events:** The shared events are those events of the domain that must be brought to the attention of the machine.
- **Shared Behaviours:** The shared behaviours are those behaviours of the domain that can only be partially 'processed' by the machine.

s778Again the requirements engineers "walk" through the domain description together with each group of requirements stake-holders marking up all the shared phenomena and concepts, and decides their basic principles resolution, which are duly noted in the evolving interface requirements document.

8.8.2 Shared Simple Entity Requirements s779

Definition 72 – **Shared Simple Entity:** *By a shared simple entity we understand a simple entity that 'occurs' in the domain but must also be represented by the machine.* ∎

shared-se-ru s780

Example 91 – **Shared Simple Entities: Railway Units:** We may think of a train traffic monitoring and control system being interface requirements developed. The following phenomena are then identified as among those being shared: *rail units*, *signals*, *road level crossing gates*, *train sensors* (optical sensor sensing passing trains) and *trains.* This ends Example 91 ∎

shared-se-tru s781

Example 92 – **Shared Simple Entities: Toll-road Units:** We may think of a toll-road traffic monitoring and control system being interface requirements developed. The following phenomena are identified as among those being shared: *links*, *hubs*, *cars*, (optical sensors sensing passing cars) *toll-both gates*, *toll-booth externally arriving car sensor*, *toll-booth internally arriving car sensor*, *toll-booth request slip sensor* and *toll-booth accept slip sensor.* This ends Example 92 ∎

acm-rtre

shared-tndr s782

Example 93 – **Shared Simple Entities: Transport Net Data Representation:** We deliberately formulated Examples 91: "Shared Simple Entities: Railway Units" and 92: "Shared Simple Entities: Toll-road Units" so as to conjure the image of two very similar set of requirements. These are now made into one set. In this example we focus on the machine representation of simple entities. We now continue these examples as well as Example 84.

s783

21. The shared simple entities are the links and the hubs. In the domain we referred to these by the sort names L and H, in the machine they will be represented by the types $LINK$ and HUB
22. Now we must make sure that we can abstract $LINK$s and HUBs "back" into L and H.

23. A number of properties that could be observed of links and hubs in the domain must be represented, somehow, in the machine. (Again we refer to Example 84.) Some properties are:link and hub *Location*, link *Length*, road (link and hub) *Surface* material (concrete, macadamised, dirt road, etc.), road (link and hub) *WearTear* (surface quality), *Date* last surveyed (i.e., monitored) *Date* last maintained (i.e., controlled) with respect to surface quality, next scheduled *Date* of survey, etc. s784
24. Let us call the pair of sets of representations of *LINK*s and *HUB*s for *NET*. We omit, in this example, the modelling of net attributes.
25. We postulate an abstraction function, abs_N, which from a concretely represented *net:NET* abstracts the abstract *n*(et) in *N*(et).
26. Tentatively we might impose the following representation theorem (a relation) between concrete and abstract nets: the links [hubs] (in *L* [in *H*]) that can be abstracted from any concrete net *net* must be those observable in the abstracted net.

s785

```
type
21 Length, Surface, WearTear, Date, L_Location, H_Location
23 LINK = LI×(HI×HI)×L_Location×Length×Surface×WearTear×(Date×Date×Date)× ...
23 HUB = HI×H_Location×Surface×WearTear×(Date×Date×Date)× ...
24 NET = LINK-set × HUB-set
value
22 abs_L: LINK → L, abs_H: HUB → L
25 abs_N: NET → N
theorems:
25 ∀ (links,hubs):NET, ∃ n:N •
25   (links,hubs) |= n ∧
25   abs_N(links,hubs) abstractly refines n ∧
26   let ls = {abs_L(link)|link:LINK • link ∈ links}
26       hs = {abs_H(hub)|hub:HUB • hub ∈ hubs} in
26   obs_Ls(abs_N(net))|=ls ∧ obs_Hs(abs_N(net))|=hs end
```

We shall discuss the representation of concrete hubs in Example 94. This ends Example 93 ■

s786 acm-rtre-c

In Example 93 ("Shared Simple Entities: Transport Net Data Representation") we kept an abstract representation of links and hubs. At some time in the software development process we are forced to decide on a concrete type representation of links and hubs so that we can implement those types through the use of a practical programming language. Here we shall choose, as already hinted at in Example 88, to represent links and hubs as tuples in relations of a relational database management system.

s787 shared-tnhs

Example 94 – **Representation of Transport Net Hubs:**

We continue Example 93 ("Shared Simple Entities: Transport Net Data Representation").

With the hints given in the text paragraph just preceding this example we are now ready to suggest a concrete type for *LINK*s and *HUB*s, namely as tuples or respective relations, but with the twist that we do not endow a concrete hub representation with the set of link identifiers that, in the domain, can be observed from that hub since, as we shall shortly show, that information can be calculated from the set of links having the same hub identifiers as that of the hub. s788 You may now object, if you did not already wonder way back in Example 10, as to why we did not already include this in the domain model of the net. The answer is: Yes, we could have done that, but we prefer to have modelled links and hubs, as we did it in Example 10, since we think that that is a most abstract, "no tricks" model. The "tricks" we refer to is represented below by the *xtr_LIs* function. s789

type
  L_Location, H_Location
  LINK = LI×(HI×HI)×L_Location×Length×Surface×WearTear×(Date×Date×Date)× ...
  HUB = HI×H_Location×Surface×WearTear×(Date×Date×Date)× ...
  RDB = LINK-**set** × HUB-**set**
value
  xtr_LIs: HUB → RDB → LI-**set**
  xtr_LIs(hi,hl,s,wt,(ld,md,nd),...)(ls,hs) ≡
    {li|li:LI•
      ∃ link:(li′,(hi′,hi″),ll,lgt,s,wt,(d′,d″,d‴),...):LINK•
      link ∈ ls ∧ (hi=hi′ ∨ hi=hi″)}

This ends Example 94 ■

acm-rtre-c s790

The requirements example that now follows to some extent deviate from the ideal of expressing requirements: rather than expressing properties, the *what*, these requirements express an *abstract design*, the *how*. One might very well claim that the example that now follows really should be moved to a section on Software Design since it can be said to be such an *abstract design*. Be that as it may, we have chosen to place the next example here, under shared entity data initialisation, as it illustrates that concept rather well. The point is that the more we include considerations of the machine the the more operational, that is, the less *what* the more *how* the interface requirements becomes.

shared-tndi s791

Example 95 – **Shared Simple Entities: Transport Net Data Initialisation:** We continue Example 93 ("Shared Simple Entities: Transport Net Data Representation"). We now focus on the initialisation of simple entity data.

27. Input of representations of simple transport net entities, that is, representations of links and hubs, is by means of a software package, call it `NetDataInput`.
28. `NetDataInput` assumes a rather old-fashioned constellation of a graphic user interface (GUI), `NetDataGUI`, in-data and a conventional relational database `NetDataRDB`.
29. The `NetDataRDB` is here thought of as just consisting of two relations `ls:LINKS` and `hs:HUBS`.
s792
30. Each relation consists of a set of `LINK`, respectively `HUB` "tupleisations" of links and hubs — which, to repeat, are representations of links and hubs .
31. When `NetDataInput` is invoked, the `NetDataInput` GUI shall open in a window with a click-able, simple *either/or* choice icon: `Road Net` or `Rail Net`.
32. Clicking one of these shall result in replacing the *either/or* window being replaced by a window for the input of net units for the selected choice of net. s793
    In the following we shall treat only the `Road Net` variant of this interface requirements.
33. The `Road Net` (GUI) window has the following alternative click-able choice icons: `link` and `hub`.
34. Clicking the `Road Net link` icon shall result in replacing the `Road Net` window being replaced by a window for the input of representations of road links. Similarly for clicking the `Road Net link` icon.
    In the following we shall treat only the `Road Net link` icon variant of this interface requirements.
s794
35. The `Road Net link` window has the following named fields:

- `link identifier` $\boxed{\text{FILL IN}}$
- `hub identifier 1` $\boxed{\text{FILL IN}}$
- `hub identifier 2` $\boxed{\text{FILL IN}}$
- `link location` $\boxed{\text{FILL IN}}$
- `link surface` $\boxed{\text{FILL IN}}$
- `link wear & tear` $\boxed{\text{FILL IN}}$
- a triple of link dates:
  - ⋆ `last survey` $\boxed{\text{FILL IN}}$
  - ⋆ `last maintenance` $\boxed{\text{FILL IN}}$
  - ⋆ `next survey` $\boxed{\text{FILL IN}}$
- et cetera.
- $\boxed{\texttt{submit}}$

Each field, except for the `submit` icon, consists of a name part and an input, $\boxed{\text{fill in}}$, part. (In the formalisation below the type names "cover" both parts.) The system shall assign unique link identifiers, i.e., "fill-in" the link identifier automatically.

s795

36. Clicking the `submit` icon shall result in the following checks:
    - The composite in-data, keyed into the input parts of the `Road Net link` window fields are vetted:
      - ⋆ Is the `link identifier` already defined ?
      - ⋆ Do the `location` co-ordinates conflict with earlier input ?
      - ⋆ Are the dates in an appropriate chronological order ?
      - ⋆ Et cetera.
    - If checks are OK, then the following actions are performed:
      - ⋆ The *NetDataRDB* `link` relation is updated to reflect the new link tuple.
      - ⋆ The `Road Net` system then reverts to the `Road Net link` window, allowing, however, the input staff to select the alternative (i) `Road Net hub` window, or (ii) to request a partial or full vetting of the state of the *NetDataRDB* `link` and `hub` relations, (iii) or to conclude the input on net data.

s796

Here we stop our 'abstract' narrative of interface requirements for data (i.e., simple entity) initialisation. We next rough-sketch the beginnings of a formalisation.

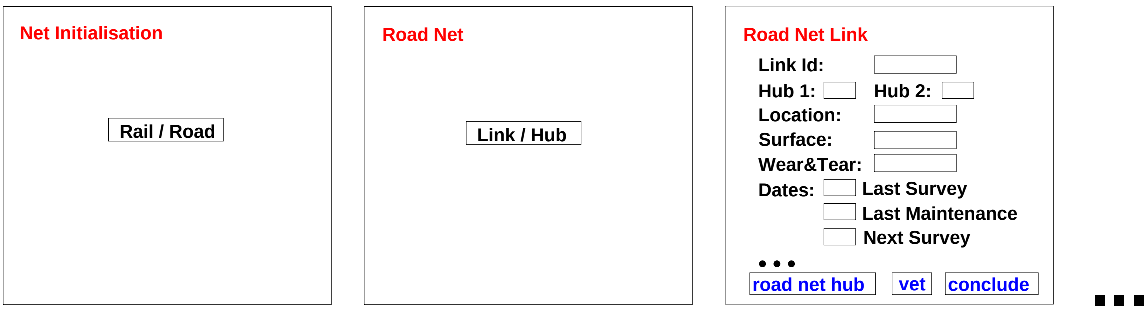


Fig. 8.2. Three snapshots of `NetDataInput`

s797

type

27 NDI
29 RDB = Links × Hubs
30 Links = LINK-**set**
30 Hubs = HUB-**set**
31 GUI == EitherOrW | RoadNetW | RailNetW
31 EitherOrW == roadnet | railnet
33 RoadNetW == link | hub
35 LINK = LI×(HI×HI)×L_Location×Surface×WearTear×(Date×Date×Date)× ...
35 HUB = HI×H_Location×Surface×WearTear×(Date×Date×Date)× ...
Response == ok | (not_ok × Error_Msg)
Error_Msg

value

28 obs_GUI: NDI → GUI, obs_RDB: NDI → RDB
32 select_RoadOrRail: GUI → GUI
34 select_Link_or_Hub: GUI → GUI
36 submit_Link_input: GUI → GUI × Response

We leave it to the reader to complete the interface requirements for shared simple entity initialisation. A similar set of interface requirements can be established for for shared simple entity refreshment. This ends Example 95 ■

8.8.3 Shared Operation Requirements s798

acm-rtre-c

Definition 73 – **Shared Operation:** *By a shared operation we understand an operation of the domain that can only be partially 'executed' by the machine — with the remaining operation parts being "executed" by a human or some "gadget" of the domain "outside" of our concern.* ■

We start by giving a consolidated domain description of a fragment of a financial services industry, in other words: Example 96 is not a requirements prescription. But it will be the basis for a shared operations interface requirements prescription.

pfm-shrd-tops s799

Example 96 – **Shared Operations: Personal Financial Transactions:** With the advent of the Internet,i.e., computing and communications, and with the merging of in-numerous functionalities of the financial service sector, we are witnessing the ability of some clients of the financial service industry to handle most of their transactions "themselves".

In the first part, Items 37–45 of this example, we rough-sketch the state and the signatures of some of the client operations of a financial service industry. In the second part, Items 50–62, we rough sketch client states and behaviours.

s800

37. The financial service industry includes one or mare banking and securities instrument trading services.
    a) Banks are uniquely identified (*BId*).
    b) Banks offer accounts:
        i. a client may have one or more demand/deposit accounts and
        ii. one or more mortgage accounts, identified by account numbers;

       accounts, in this simplified example, holds a balance of (deposited or mortgaged) money.
    c) Two or more clients may share accounts and bank registers correlate client names (*C*) to account and (*A*) mortage (*M*) account numbers.
    d) We do not describe bank identifiers, client names, demand/deposit account numbers, mortgage account numbers,

s801

type

37	Banks, SecTrad
37a	Banks = BId $\overrightarrow{m}$ Bank
37b	Bank = Registers × Accounts × Mortgages × ...
37(b)i	Accounts = A $\overrightarrow{m}$ Account
37(b)ii	Mortgages = M $\overrightarrow{m}$ Mortgage
37c	Registers = C $\overrightarrow{m}$ (A|M)-**set**
37d	C, A, BId, Account, Mortgage, Account, Mortgage, OrdNr

s802

38. Bank clients
    a) *open account* and
    b) *close* (demand/deposit and mortgage) *accounts*,
    c) *deposit* money into accounts,
    d) *transfer* money to (possibly other client) accounts (possibly in other banks and to/from security traders' bank accounts — the latter reference also to buy and sell offer order numbers, *OrdNr*), and

e) *withdraw* (cash) money (say, through an ATM).
39. Client supplied arguments to and
40. responses from these banking operations are also not further described.

s803

value
38a openacct: Arg* → Banks → Banks × Response
38b closeacct: Arg* → Banks → Banks × Response
38c deposit: Arg* → Banks → Banks × Response
38d transfer: Arg* → Banks → Banks × Response
38e withdraw: Arg* → Banks → Banks × Response
type
39 Arg = BId | C | A | M | OrdNr | Amount | Cash | Date | Time
40 Response = (A | M | Cash | ... | Date | Time)-**set**

s804

41. A securities exchange keeps track of buy and sell offers, of suspended such offerings and of transacted trading.
42. Basic concepts of trading, apart from buying, selling, suspension and concluded trading, are
   a) securities instrument identifications (*SId*);
   b) quantities offered for selling or buying, or traded (*Quant*);
   c) the order numbers of placed offers (*OrdNr*),
   d) prices (*Price*),
   e) dates (*Date*) and
   f) times (*Time*).

type
41 SecTrad = BuyOfrs × SellOfrs × Suspension × Tradings
42 SId, Quant, OrdNr, Price, Date, Time, ...

s805

43. Securities trading allows clients
   a) to place buy and
   b) sell offers,

   giving their client and bank identification, their bank demand/deposit account number (from which to withdraw [i.e., demand], resp. into which to deposit) buying or selling prices), the securities instrument [e.g., stock] identifier, quantity to be bought or sold, the high, respectively the low price acceptable, and the last date of the offer.
44. Clients may inquire as to the trading status of their offer.
45. We can therefore think of the following kinds of client transaction "codes" (*Cmd*): *omkt* (observe the market), *open* (some kind of bank or securities trading account: demand/deposit, mortgage, trading, etc.), *deposit, withdraw, transfer, close, buy offer, sell offer, inquire,* et cetera.

s806

value
43a int_buyofr: Arg* → SecTrad → SecTrad × Response
43b int_sellofr: Arg* → SecTrad → SecTrad × Response
44 trading: SecTrad → SecTrad × Response

type
45 Cmd == obsmkt|analmkt|openacct|deposit|withdraw|transfer|closeacct|...|buyofr|sellofr|trading|...

s807

We can, finally, suggest crucial components of the securities exchange state:

46. *BuyOfr*s map client names, *C*, into (client) bank identifiers, *BId*, and client account numbers, *A*, which then map into *OrdNr*s, which (then again) map into a quadruple of securities instrument identifications, *SId*,*Quant*ity of instrument to be bought, the preferred lowest *Price* and the *Date* of placement or order.
47. *SellOfr*s have same components as buy offers — but now the *Price* designate a highest price.
48. *Suspension*s just list order number and date and time of suspension.
49. *Tradings* list pertinent information.

type
46 BuyOfr = OrdNr $\overrightarrow{m}$ (C × BId × A) $\overrightarrow{m}$ (SId × Quant × Price × Date)
47 SellOfr = OrdNr $\overrightarrow{m}$ (C × BId × A) $\overrightarrow{m}$ (SId × Quant × Price × Date)
48 Suspension = OrdNr $\overrightarrow{m}$ (C × Date × Time)
49 Tradings = OrdNr $\overrightarrow{m}$ SId × Quant × Price × (C×Bid×A×(Date×Time))

s808

We now rough-sketch a concept of 'personal finance management' operations. It is in this part, not the first, that this example reveals that it is an example of an operation that is shared between the domain and the machine. We maintain, however, that the example is still that of a domain description. The operation is that of a client managing own, personal finances. s809This 'personal finance management' operation is a composite operation. It is a sequence of "one-step" operations, each operation being a banking or a securities trading. (For simplicity, but without any loss of generality, we limit the example to just these two sets of operations.) Each such operation results in a date- and time-stamped response (Item 40 on the preceding page). Each response is studied by the client. The client may then decide to proceed with further 'one-step' operations or end this sequence "at this time" — allowing, of course, the client to resume 'personal finance management' at a later 'personal finance management session'. s810

40. Each of the banking and securities instrument operations result in a response.
50. This response becomes part of the client's 'finance management' state, $\Pi\Phi\Sigma$.
51. We refer to the global financial service industry state as Ω.
    Besides the banks and securities trading, the global financial service industry state (ω:Ω) is thought of as including all those aspects of the clients of this industry which affects and/or reflects the financial situation.
52. A 'personal finance management (*pfm*) *session*' is now a conditional iteration (formula Line 56 below) of personal finance management operations. s811
53. A client can always *obs*erve the dated and timed *responses* received as a result of past *p*ersonal *f*inance *m*anagement operations.
54. An iteration of 'personal finance management' starts with the client analysing (ω*_anal_mkt*) the market based on *past responses*;
55. followed by an analysis (*cli_anal_mkt*) of the personal financial situation based on the market *response*.
56. If the analysis advises some 'personal finance management'
57. then the client inquires, *what_to_do*, past responses and as to which transaction, *cmd*, and with which arguments, *argl*, such a transaction should be performed.
    This operation, *what_to_do*, is not computable. It is an operation performed basically by the client.

s812

58. The client then performs (*Int_Cmd*) this (i.e., the *cmd*) transaction. The transaction usually transforms the finance industry state (ω) into a next state (ω') and always yields a *date*- and
59. Once the transaction has been concluded the client reverts to the 'personal finance management (pfm) session' with an updated "past responses" (merge_pfm) and in the new global state, ω'.
60. Else, that is, if the analysis "advises" no transactions, the 'personal finance management' state, $\pi\phi\sigma$ and the global financial state, ω, is left unchanged and the (i.e., this) session ends.
61. To perform a transaction depends on which kind of transaction, cmd, has been advised.

62. We leave the interpretation of Int_Cmd to the reader.

π The lines, below, marked π designate actions that are performed by the client or jointly between the client and the financial system (designated by an ω or ω' argument).

s813

type
50–51 $\Pi\Phi\Sigma$, Ω
53 Responses = (Date$\times$Time) $\overrightarrow{m}$ Response
45 Cmd == obsmkt|analmkt|openacct|deposit|withdraw|transfer|closeacct|...|buyofr|sellofr|trading|...
value
53 obs_Responses: $\Pi\Phi\Sigma \to$ Responses
52 pfm_session: $\Pi\Phi\Sigma \to \Omega \to \Omega \times \Pi\Phi\Sigma$
52 pfm_session($\pi\phi\sigma$)(ω) $\equiv$
53π **let** past_responses = obs_Responses($\pi\phi\sigma$) **in**
54π **let** ω_response = ω_anal_mkt(past_responses)(ω) **in**
55π **let** $\pi\phi\sigma$_response = cli_anal_mkt(response)($\pi\phi\sigma$) **in**
56π **if** advice_pfm_action($\pi\phi\sigma$_response)
57π **then let** (cmd,argl) = what_to_do($\pi\phi\sigma$_response)(ω) **in**
58   **let** (response,ω') = Int_Cmd(cmd,argl)(ω) **in**
59π pfm_session(merge_pfm(response,date,time)($\pi\phi\sigma$))(ω') **end end end**
60   **else** (ω,$\pi\phi\sigma$) **end end end**
54 ω_anal_mkt: Responses $\to \Omega \to$ Response
55π cli_anal_mkt: Response $\to \Pi\Phi\Sigma \to$ Response
56 advice_pfm_action: Response $\to$ **Bool**
57π what_to_do: Responses $\to \Omega \to$ Cmd $\times$ Arg*
59π merge_pfm: Responses $\to \Pi\Phi\Sigma \to \Omega \to \Omega \times \Pi\Phi\Sigma$

s814

61. To perform a transaction depends on which kind of transaction, cmd, has been advised.
63. A case distinction is made between the very many kinds of transactions (listed in Item 45).
64. The *obs_mkt* and *anal_pfm* operations do not change the state of the financial industry.

    We think of these operations as not being computable functions. Rather we think of them as a more-or-less "informed" study, by the client of the market of financial instruments including the status of those enterprises whose stocks are traded.
65. The *arg*ument *l*ist of the *open* transaction indicates which kind of account is to be established (demand/deposit, mortgage, etc.).
66. The *arg*ument *l*ist of the *buy* *of*fer transaction indicates which kind of securities (stocks, oil, metals, or other commodities) is sought, in which quantity, at which price level, up till which date, et cetera.

s815

value
61 Int_Cmd: (Cmd $\times$ Arg*) $\to \Omega \to \Omega \times$ Response
61 Int_Cmd(cmd,argl)(ω) $\equiv$
63 **case** cmd **of**
64π obsmkt $\to$ (ω,eval_obs_mkt(argl)(ω)),
64π analmkt $\to$ (ω,ω_anal_mkt(argl)(ω)),
...
65π openacct $\to$ int_open_acct(argl)(ω),
...
66π buyofr $\to$ int_buyofr(argl)(ω),
...
63 **end**

64π eval_obs_mkt: Arg* $\rightarrow$ Ω $\rightarrow$ Response
64π ω_anal_mkt: Arg* $\rightarrow$ Ω $\rightarrow$ Response
65 int_open_acct: Arg* $\rightarrow$ Ω $\rightarrow$ Ω $\times$ Response
66 int_buyofr: Arg* $\rightarrow$ Ω $\rightarrow$ Ω $\times$ Response

This ends Example 96 ■

acm-rtre-c s816

The reader may well ask: What in Example 96 illustrates the shared operations interface requirements ? We have already indicated part of the answer to this question by the π annotations. Why is this a reasonable question ? It is a reasonable question because we have not made that abundantly clear. That is, we have not discussed the placement of π annotations in much detail. Example 96 could really be construed as a domain description based in the intrinsics, support technology, management and organisation and human behaviour regime.

8.8.4 Shared Event Requirements s817

acm-rtre-c

Definition 74 – **Shared Event:** *By a shared event we understand an event of the domain that must be brought to the attention of the machine.* ■

We defer the exemplification of shared events till our treatment of 'shared behaviour'. In Sect. 8.8.5 we exemplify 'shared behaviours'. The 'step of development', from the specification of Example 96 (*Shared Operations*) to the specification of Example 97 (*Shared Behaviours*) is not a formal refinement: but it can be made into such a formally verifiable refinement. So we pose that as a relevant MSc Thesis topic.

8.8.5 Shared Behaviour Requirements s818

Definition 75 – **Shared Behaviour:** *By a shared behaviours we understand a behaviour of the domain that can only be partially 'processed' by the machine — with the remaining behaviour being provided by humans or some "gadgets" of the domain, "outside" of our concern.* ■

pfm-shrd-bhvs s819

Example 97 – **Shared Behaviours: Personal Financial Transactions:**

67. There is an index set, *CI*, of clients.
68. For each client there is an "own"
69. 'personal finance management' state $\pi\phi\sigma_{ci} : \Pi\Phi\Sigma$ (cf. 50 on page 160).
70. The finance industry "grand state" $\omega : \Omega$ is as before (cf. Item and formula line 51 on page 160).
71. The system consists of
    a) an indexed set of *client* behaviours
    b) and one finance industry "grand state" behaviour *omega*.
72. We model communications between clients and the financial industry to occur over client-industry channels.
73. We model communications over these channels as being of type *M*. *M* will be "revealed" as we go on.

s820

type
67 CI
68 $\Pi\Phi\Sigma$s = CI $\overrightarrow{m}$ $\Pi\Phi\Sigma$
69 $\Pi\Phi\Sigma$
70 Ω = Banks $\times$ SecTrad $\times$ ...

value
67 cis:CI-**set**
68 $\pi\phi\sigma$s:$\Pi\Phi\Sigma$s
70 ω:Ω
71 system: **Unit** $\rightarrow$ **Unit**
71 system() $\equiv$
71a $\parallel$ {client(ci)($\pi\phi\sigma$s(ci))|ci:CI•ci $\in$ cis}
71b $\parallel$ omega(ω)
channel
72 {c_ω_ch[ci]|ci:CI•ci $\in$ cis} M
type
73 M

s821

Let us first consider the issue of events. First the events arise in "the market", here symbolised with the global state $\omega : \Omega$.

74. The *omega(ω)* behaviour and the *client(ci)($\pi\phi\sigma$s(ci))* behaviours, for all *clients*, are cyclic — expressed through 'tail recursion' over possibly updated states.
75. To model events we let the *omega(ω)* behaviour alternate between either
    a) inquiring its state as to unusual situations in, the status *status* of, "the market", and,
    b) if so, inform an arbitrary subsets of *clients* of such "events" and
    c) continuing in an unchanged global financial system state

    or
76. servicing *client* requests — from any client.

s822

type
Event
value
omega: Ω $\rightarrow$ **in**,**out** {c_ω_ch[ci]|ci:CI•ci $\in$ cis} **Unit**
omega(ω) $\equiv$
75b (**let** scis:CI-**set** • scis$\subseteq$cis **in**
75a **let** event = ωstatus(ωsnapshot(ω)) **in**
75b **if** nok_status(event) **then** {c_ω_ch[ci]!event|ci:CI•ci $\in$ scis} **end**;
75c omega(ω) **end** **end**)
75 $\lceil\rceil$
76 $\lceil\rceil$ {**let** req = c_ω_ch[ci] ? **in** ... see Items81–88b ... **end** | ci:CI • ci $\in$ cis}
75a ωsnapshot: Ω $\rightarrow$ Ω
75a ωstatus: Ω $\rightarrow$ Event
75b nok_status: Event $\rightarrow$ **Bool**

s823

Then we consider how clients respondto becoming aware of unusual events in "the market". s824

77. Clients alternate between handling events:
78. first deciding (76–78) to "listen to the market",
    a) then updating an own personal finance state ($\pi\phi\sigma$),
    b) and then coming a client behaviour in that new personal finance state,

    and
79. handling ordinary, that is, personal finance management (*pfm*) or
80. just idling.

s825

value
77 client: ci:CI $\rightarrow$ $\Pi\Phi\Sigma$ $\rightarrow$ **in**,**out** c_ω_ch[ci] **Unit**
77 client(ci)($\pi\phi\sigma$) $\equiv$
78 (**let** event = c_ω_ch[ci]? **in**
78a **let** $\pi\phi\sigma'$ = update_$\pi\phi\sigma$($\pi\phi\sigma$)(event) **in**
78b client(ci)($\pi\phi\sigma'$) **end end**)
77 $\lceil\rceil$
79 client(ci)(pfm_session(ci)($\pi\phi\sigma$))
77 $\lceil\rceil$
80 client(ci)($\pi\phi\sigma$)

s826

Let us now turn to the treatment of usual financial transactions. The main functions, in Example 96, are *pfm_session* (Items 52–60 and formulas on Pages 161–161) and *Int_Cmd* (Items 63–66 and formulas on Pages 161–162) We now analyse these two functions. We refer, in the following to the formula lines on Pages 161–161 and Pages 161–162. The analysis is with respect to what actions, π, are expected to occur in the *client* behaviour and what actions are expected from the financial industry (i.e., to occur in the *omega* behaviour).

s827

In *pfm_session* (Pages 161–161), formula lines

- 53π: *obs_Responses($\pi\phi\sigma$)*,
- 55π: *cli_anal_mkt(response)($\pi\phi\sigma$)*,
- 56π: **if** *ok_or_nok* = *ok* and
- 59π: *merge_pfm(response,date,time)($\pi\phi\sigma$)*

are expected from the *client* behaviour, all others from the industry, i.e., the *omega* behaviour.

In *Int_Cmd* (Pages 161–162), formula lines

- 64π: *eval_obs_mkt(argl)(ω)*,
- 64π: *ω_anal_mkt(argl)(ω)*,
- 65π: *int_open_acct(argl)(ω)*,
- . . . ,
- 66π: *int_buyofr(argl)(ω)*,
- . . . ,

are expected from the *client* behaviour.

s828 Of the functions itemised above, the

81. (54π) *ω_anal_mkt(past_responses)(ω)*,
82. (64π) *eval_obs_mkt(ω)*,
83. (65π) *int_open_acct(argl)(ω)*,
84. . . . ,
85. (66π) *int_buyofr(argl)(ω)*,
86. . . . ,

functions — as invoked by the *client*, in the *pfm_session* and the *Int_Cmd* behaviours — require access to the global financial service industry state ω.

The idea is now, for the *client*, in its *Int_Cmd* (see Formula lines 61.0 onwards [Page 166]) to communicate these functions, as function named argument lists, cf. Formula lines 81–86 below.

s829

type
FCT == Anal_mkt|Obs_mkt|...|Open_aact|...|Buy_Ofr|...
81 Anal_mkt = mkAnalMkt(argl:Arg*)
82 Obs_mkt = mkObsMkt(argl:Arg*)
83 Open_acct = mkOpenAcct(argl:Arg*)
84 ...
85 Buy_Ofr = mkBuyOfr(argl:Arg*)
86 ...
channel
72 {c_ω_ch[ci]|ci:CI•ci $\in$ cis}: Event | FCT | Response

s830

87. The *omega* behaviour thus alternates
    a) between accepting and responding to either of the many forms of functions, *FCT*
    b) or generating event notifications.
88. If the *omega* behaviour of its own will, that is, internally non-deterministically chooses to accept a client initiate request it externally non-deterministically chooses which client request to serve.
    a) The *omega* behaviour deciphers the request;
    b) applies the communicated function to (possibly communicated arguments) and the ω"grand state"; and communicates the result back to the chosen client.

s831

```
value
     omega: Ω → in,out {c_ω_ch[ci]|ci:CI•ci ∈ cis}  Unit
     omega(ω) ≡
87b    (let scis:CI-set • scis⊆cis in
87b     let event = ωstatus(ωsnapshot(ω)) in
87b     if nok_status(event) then {c_ω_ch[ci]!event|ci:CI•ci ∈ scis} end;
87b     omega(ω) end end)
87    ⌈⌉
88    ⌈⌉ {let req = c_ω_ch[ci] ? in
88a        case req of
87a          mkObsMkt(argl) → c_ω_ch[ci] ! eval_obs_mkt(argl)(ω) ; omega(ω),
87a          mkAnalMkt(argl) → c_ω_ch[ci] ! ω_anal_mkt(argl)(ω) ; omega(ω),
87a          mkOpenAcct(argl) →
88a            let (ω′,res) = int_open_acct(argl)(ω) in c_ω_ch[ci] ! res ; omega(ω′) end,
87a          ...
87a          mkBuyOfr(argl) →
88a            let (ω′,res) = int_buyofr(argl)(ω) in c_ω_ch[ci] ! res ; omega(ω′) end,
87a          ...
88a        end end | ci:CI • ci ∈ cis}
```

s832

Some auxiliary functions:

```
value
64π  eval_obs_mkt: Arg* → Ω → Response
64π  ω_anal_mkt: Arg* → Ω → Response
65   int_open_acct: Arg* → Ω → Ω × Response
66   int_buyofr: Arg* → Ω → Ω × Response
```

There is only minor changes, marked $\surd$, to pfm_session: (52) an additional argument, ci:CI and (58) an additional argument, (ci) to Int_Cmd(ci)(cmd,argl).

s833

```
value
52√  pfm_session: ci:CI → ΠΦΣ → in,out ωch  ΠΦΣ
52√  pfm_session(ci)(πφσ) ≡
53     let past_responses = obs_Responses(πφσ) in
54     let response = ωch[ci]!mkObsMkt(past_responses); ωch[ci] ? in
55     let response = analyse_pfm(past_responses)(πφσ) in
56     if advice_pfm_action(response)
57       then (let (cmd,argl) = what_pfm_to_do(response) in
58√          let response = Int_Cmd(ci)(cmd,argl) in
59√          pfm_session(ci)(merge_pfm(response)(πφσ)) end end) end
```

```
60          else πφσ end end end

56   advice_pfm_action: Response → Bool
55   analyse_pfm: Responses → ({|ok|}|Event) → {|ok|nok|}
```

Similarly the changes to Int_Cmd are obvious and marked, as before, with $.i, i = 0, 1, 2, 3$:

s834

```
value
61.0  Int_Cmd: ci:CI × (Cmd × Arg*) → in,out ωch  Response
61    Int_Cmd(ci)(cmd,argl) ≡
61        case cmd of
61.2        obsmkt → ωch[ci]!mkObsMkt(argl) ; ωch[ci]?
61.2        analpfm → ωch[ci]!mkAnalMkt(argl) ; ωch[ci]?
61          ...
61.2        openacct → ωch[ci]!mkOpenAcct(argl) ; ωch[ci]?
61          ...
61.3        buyofr → ωch[ci]!mkBuyOfr(argl) ; ωch[ci]?
61          ...
61        end
```

This ends Example 97 ■

Discussion s835

acm-rtre-c

TO BE WRITTEN

8.9 Machine Requirements s836

acm-rtre-d

Definition 76 – **Machine Requirements:** *By machine requirements we understand those requirements that can be expressed sôlely in terms of (or with prime reference to) machine concepts* ■

8.9.1 An Enumeration of Machine Requirements Issues s837

There are many separable machine requirements. To find one's way around all these separable machine requirements we shall start by enumerating the very many that we shall overview.

s838

1. Performance Requirements — from Page 167
   a) Storage Requirements — from Page 168
   b) Machine Cycle Requirements — from Page 168
   c) Other Resource Consumption Requirements — from Page 168
2. Dependability Requirements — from Page 168
   a) Accesability Requirements — from Page 170
   b) Availability Requirements — from Page 170
   c) Integrity Requirements — from Page 171
   d) Reliability Requirements — from Page 171
   e) Safety Requirements — from Page 171
   f) Security Requirements — from Page 171

s839

3. Maintenance Requirements — from Page 172

8.9.2 Performance Requirements s840

Definition 77 – **Performance Requirements:** *By performance requirements we understand machine requirements that prescribe storage consumption, (execution, access, etc.) time consumption, as well as consumption of any other machine resource: number of CPU units (incl. their quantitative characteristics such as cost, etc.), number of printers, displays, etc., terminals (incl. their quantitative characteristics), number of "other", ancillary software packages (incl. their quantitative characteristics), of data communication bandwidth, etcetera* ■

s841

Pragmatically speaking, performance requirements translate into financial resources spent, or to be spent.

Example 98 – **Timetable System Performance:** We continue Example 86 on page 150. The machine shall serve 1000 users and 1 staff member. Average response time shall be at most 1.5 seconds, when the system is fully utilised. This ends Example 98 ■

General s842

Till now we may have expressed certain (functions and) behaviours as generic (functions and) behaviours. From now on we may have to "split" a specified behaviour into an indexed family of behaviours, all "near identical" save for the unique index. And we may have to separate out, as a special behaviour, (those of) shared entities.

s843

Example 99 – **Timetable System Users and Staff:** We continue Example 83 on page 146 and Example 98. In Example 83 the sharing of the timetable between users and staff was expressed parametrically.

system(tt) $\equiv$ client(tt) $\sqcap$ staff(tt)

client: TT $\rightarrow$ **Unit**
client(tt) $\equiv$ **let** q:Query **in let** v = $\mathcal{M}_q$(q)(tt) **in** system(tt) **end end**

staff: TT $\rightarrow$ **Unit**
staff(tt) $\equiv$
  let u:Update **in let** (r,tt′) = $\mathcal{M}_u$(u)(tt) **in** system(tt′) **end end**

We now factor the timetable entity out as a separate behaviour, accessible, via indexed communications, i.e., channels, by a family of client behaviours and the staff behaviour. s844

```
type
   CIdx /* Index set of, say 1000 terminals */
channel
   { ct[i]:QU,tc[i]:VAL | i:CIdx }
   st:UP,ts:RES
value
   system: TT → Unit
   system(tt) ≡ time_table(tt) ∥ (∥ {client(i)|i:CIdx}) ∥ staff()
```

s845

```
   client: i:CIdx → out ct[i] in tc[i]  Unit
   client(i) ≡ let qc:Query in ct[i]!M_q(qc) end tc[i]?;client(i)

   staff: Unit → out st in ts  Unit
   staff() ≡ let uc:Update in st!M_u(uc) end let res = ts? in staff() end

   time_table: TT → in {ct[i]|i:CIdx},st  out {tc[i]|i:CIdx},ts  Unit
   time_table(tt) ≡
      [] {let qf = ct[i]? in tc[i]!qf(tt) end | i:CIdx}
      [] let uf = st? in let (tt′,r)=uf(tt) in ts!r; time_table(tt′) end end
```

s846 Please observe the "shift" from using [] in *system* earlier in this example to [] just above. The former expresses nondeterministic internal choice. The latter expresses nondeterministic external choice. The change can be justified as follows: The former, the nondeterministic internal choice, was "between" two expressions which express no external possibility of influencing the choice. The latter, the nondeterministic external choice, is "between" two expressions where both express the possibility of an external input, i.e., a choice. The latter is thus acceptable as an implementation of the former. This ends Example 99 ∎

s847 The next example, Example 100, continues the performance requirements expressed just above. Those two requirements could have been put in one phrase, i.e., as one prescription unit. But we prefer to separate them, as they pertain to different kinds (types, categories) of resources: terminal
s848 + data communication equipment facilities versus time and space.

Example 100 – **Storage and Speed for n-Transfer Travel Inquiries:** We continue Example 86 on page 150. When performing the n-*Transfer Travel Inquiry* (rough sketch) prescribed above, the first — of an expected many — result shall be communicated back to the inquirer in less than 5 seconds after the inquiry has been submitted, and, at no time during the calculation of the "next" results must the storage buffer needed to calculate these exceed around 100,000 bytes. ∎

Storage Requirements s849

Machine Cycle Requirements s850

Other Resource Consumption s851

8.9.3 Dependability Requirements s852

To properly define the concept of *dependability* we need first introduce and define the concepts of *failure*, *error*, and *fault*.

s853

Definition 78 – **Failure:** *A machine failure occurs when the delivered service deviates from fulfilling the machine function, the latter being what the machine is aimed at [119]* ∎

s854

Definition 79 – **Error:** *An error is that part of a machine state which is liable to lead to subsequent failure. An error affecting the service is an indication that a failure occurs or has occurred [119]* ∎

s855

Definition 80 – **Fault:** *The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a fault [119]* ∎

The term hazard is here taken to mean the same as the term fault.

One should read the phrase: "adjudged or hypothesised cause" carefully: In order to avoid an unending trace backward as to the cause,[10] we stop at *the cause which is intended to be prevented or tolerated.*

s856

Definition 81 – **Machine Service:** *The service delivered by a machine is its behaviour as it is perceptible by its user(s), where a user is a human, another machine or a(nother) system which interacts with it [119]* ∎

Definition 82 – **Dependability:** *Dependability is defined as the property of a machine such that reliance can justifiably be placed on the service it delivers [119]* ∎

s857

We continue, less formally, by characterising the above defined concepts [119]. "A given machine, operating in some particular environment (a wider system), may fail in the sense that some other machine (or system) makes, or could in principle have made, a *judgement* that the activity or inactivity of the given machine constitutes a *failure*".

The concept of *dependability* can be simply defined as "the quality or the characteristic of being dependable", where the adjective 'dependable' is attributed to a machine whose failures are judged sufficiently rare or insignificant.

Impairments to dependability are the unavoidably expectable circumstances causing or resulting from "undependability": faults, errors and failures. *Means* for dependability are the techniques enabling one to provide the ability to deliver a service on which reliance can be placed, and to reach confidence in this ability. *Attributes* of dependability enable the properties which are expected from the system to be expressed, and allow the machine quality resulting from the impairments and the means opposing them to be assessed. s858

Having already discussed the "threats" aspect, we shall therefore discuss the "means" aspect of the *dependability tree*.

- Attributes:
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Integrity
  - ⋆ Reliability
  - ⋆ Safety
  - ⋆ Security s859
- Means:
  - ⋆ Procurement
    - · Fault prevention
    - · Fault tolerance
  - ⋆ Validation
    - · Fault removal
    - · Fault forecasting
- Threats:

[10] An example: "The reason the computer went down was the current supply did not deliver sufficient voltage, and the reason for the drop in voltage was that a transformer station was overheated, and the reason for the overheating was a short circuit in a plant nearby, and the reason for the short circuit in the plant was that . . . , etc."

- ⋆ Faults
- ⋆ Errors
- ⋆ Failures

s860

Despite all the principles, techniques and tools aimed at *fault prevention*, *faults* are created. Hence the need for *fault removal*. *Fault removal* is itself imperfect. Hence the need for *fault forecasting*.
s861 Our increasing dependence on computing systems in the end brings in the need for *fault tolerance*.

Definition 83 – **Dependability Attribute:** *By a dependability attribute we shall mean either one*
s862 *of the following: accessibility, availability, integrity, reliability, robustness, safety and security. That is, a machine is dependable if it satisfies some degree of "mixture" of being accessible, available, having integrity, and being reliable, safe and secure* ▪

The crucial term above is "satisfies". The issue is: To what "degree"? As we shall see — in a later section — to cope properly with dependability requirements and their resolution requires that we deploy mathematical formulation techniques, including analysis and simulation, from statistics (stochastics, etc.).

In the next seven subsections we shall characterise the dependability attributes further. In doing so we have found it useful to consult [95].

Accesability Requirements s863

Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Their being granted access to computing time is usually specified, at an abstract level, as being determined by some internal nondeterministic choice, that is: essentially by *"tossing a coin"!* If such internal nondeterminism was carried over,
s864 into an implementation, some *"coin tossers"* might never get access to the machine.

Definition 84 – **Accessibility:** *A system being accessible — in the context of a machine being dependable — means that some form of "fairness" is achieved in guaranteeing users "equal" access to machine resources, notably computing time (and what derives from that)* ▪

s865

Example 101 – **Timetable Accessibility:** Based on Examples 83 on page 146 and 86 on page 150, we can express: The timetable (system) shall be "inquirable" by any number of users, and shall be update-able by a few, so authorised, airline staff. At any time it is expected that up towards a thousand users are directing queries at the timetable (system). And at regular times, say at midnights between Saturdays and Sundays, airline staff are making updates to the timetable (system). No matter how many users are "on line" with the timetable (system), each user shall be given the appearance that that user has exclusive access to the timetable (system). ▪

Availability Requirements s866

Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Once a user has been granted access to machine resources, usually computing time, that user's computation may effectively make the machine unavailable to other users — by "going on and on and on"!

s867

Definition 85 – **Availability:** *By availability — in the context of a machine being dependable — we mean its readiness for usage. That is, that some form of "guaranteed percentage of computing time" per time interval (or percentage of some other computing resource consumption) is achieved — hence some form of "time slicing" is to be effected* ▪

s868

Example 102 – **Timetable Availability:** We continue Examples 83 on page 146, 86 on page 150 and 101 on the preceding page: No matter which query composition any number of (up to a thousand) users are directing at the timetable (system), each such user shall be given a reasonable amount of compute time per maximum of three seconds, so as to give the psychological appearance that each user — in principle — "possesses" the timetable (system). If the timetable system can predict that this will not be possible, then the system shall so advise all (relevant) users. ▪

Integrity Requirements s869

Definition 86 – **Integrity:** *A system has integrity — in the context of a machine being dependable — if it is and remains unimpaired, i.e., has no faults, errors and failures, and remains so, without these, even in the situations where the environment of the machine has faults, errors and failures* ▪

Integrity seems to be a highest form of dependability, i.e., a machine having integrity is 100% dependable! The machine is sound and is incorruptible.

Reliability Requirements s870

Definition 87 – **Reliability:** *A system being reliable — in the context of a machine being dependable — means some measure of continuous correct service, that is, measure of time to failure*
▪

Example 103 – **Timetable Reliability:** Mean time between failures shall be at least 30 days, and downtime due to failure (i.e., an availability requirements) shall, for 90% of such cases, be less than 2 hours. ▪

Safety Requirements s871

Definition 88 – **Safety:** *By safety — in the context of a machine being dependable — we mean some measure of continuous delivery of service of either correct service, or incorrect service after benign failure, that is: Measure of time to catastrophic failure* ▪

Example 104 – **Timetable Safety:** Mean time between failures whose resulting downtime is more than 4 hours shall be at least 120 days. This ends Example 104 ▪

Security Requirements s872

We shall take a rather limited view of security. We are not including any consideration of security against brute-force terrorist attacks. We consider that an issue properly outside the realm of software engineering.

Security, then, in our limited view, requires a notion of *authorised user*, with authorised users being fine-grained authorised to access only a well-defined subset of system resources (data, functions, etc.). An *un-authorised user* (for a resource) is anyone who is not authorised access to that resource.

A terrorist, posing as a user, should normally fail the authorisation criterion. A terrorist, posing as a brute-force user, is here assumed to be able to capture, somehow, some authorisation status. We refrain from elaborating on how a terrorist might gain such status (keys, passwords, etc.)! s873

Definition 89 – **Security:** *A system being secure — in the context of a machine being dependable — means that an un-authorised user, after believing that he or she has had access to a requested system resource: (i) cannot find out what the system resource is doing, (ii) cannot find out how the system resource is working and (iii) does not know that he/she does not know! That is, prevention of un-authorised access to computing and/or handling of information (i.e., data)* ▪

s874

The characterisation of security is rather abstract. As such it is really no good as an a priori design guide. That is, the characterisation gives no hints as how to implement a secure system. But, once a system is implemented, and claimed secure, the characterisation is useful as a guide on how to test for security!

s875

Example 105 – **Timetable Security:** We continue Examples 83 on page 146, 86 on page 150, 101 on page 170, and 102 on the previous page. Timetable users can be any airline client logging in as a user, and such (logged-in) users may inquire the timetable. The timetable machine shall be secure against timetable updates from any user. Airline staff shall be authorised to both update and inquire, in a same session. ∎

s876

Example 106 – **Hospital Information System Security:** General access to (including copying rights of) specially designated parts of a(ny) hospital patient's medical journals is granted, in principle, only to correspondingly specially designated hospital staff. In certain forms of (otherwise well-defined) emergency situations any hospital paramedic, nurse or medical doctor may "hit a panic button", getting access to a hospital patient's medical journal, but with only viewing, not copying rights. Such incidents shall be duly and properly recorded and reported, such that proper post-processing (i.e., evaluation) of such "panic button" accesses can take take place. This ends Example 106 ∎

Robustness Requirements s877

Definition 90 – **Robustness:** *A system is robust — in the context of dependability — if it retains its attributes after failure, and after maintenance* ∎

Thus a robust system is "stable" across failures and "across" possibly intervening "repairs" and "across" other forms of maintenance.

• • •

8.9.4 Maintenance Requirements s878

Definition 91 – **Maintenance Requirements:** *By maintenance requirements we understand a combination of requirements with respect to: (i) adaptive maintenance, (iii) corrective maintenance, (ii) perfective maintenance, (iv) preventive maintenance and (v) extensional maintenance* ∎

s879

Maintenance of building, mechanical, electro-technical and electronic artifacts — i.e., of artifacts based on the natural sciences — is based both on documents and on the presence of the physical artifacts. Maintenance of software is based just on software, that is, on all the documents (including tests) entailed by software.

Adaptive Maintenance Requirements s880

Definition 92 – **Adaptive Maintenance:** *By adaptive maintenance we understand such maintenance that changes a part of that software so as to also, or instead, fit to some other software, or some other hardware equipment (i.e., other software or hardware which provides new, respectively replacement, functions)* ∎

s881

Example 107 – **Timetable System Adaptability:** The timetable system is expected to be implemented in terms of a number of components that implement respective domain and interface requirements, as well as some (other) machine requirements. The overall timetable system shall have these components connected, i.e., interfaced with one another — where they need to be interfaced — in such a way that any component can later be replaced by another component ostensibly delivering the same service, i.e., functionalities and behaviour. This ends Example 107 ∎

Corrective Maintenance Requirements s882

Definition 93 – **Corrective Maintenance:** *By corrective maintenance we understand such maintenance which corrects a software error* ∎

Example 108 – **Timetable System Correct-ability:** Corrective maintenance shall be done remotely: from a developer site, via secure Internet connections. ∎

Perfective Maintenance Requirements s883

Definition 94 – **Perfective Maintenance:** *By perfective maintenance we understand such maintenance which helps improve (i.e., lower) the need for hardware (storage, time, equipment), as well as software* ∎

s884

Example 109 – **Timetable System Perfectability:** The system shall be designed in such a way as to clearly be able to monitor the use of "scratch" (i.e., buffer) storage and compute time for any instance of any query command. ∎

Preventive Maintenance Requirements s885

Definition 95 – **Preventive Maintenance:** *By preventive maintenance we understand such maintenance which helps detect, i.e., forestall, future occurrence of software or hardware errors* ∎

Preventive maintenance — in connection with software — is usually mandated to take place at the conclusion of any of the other three forms of (software) maintenance.

Extensional Maintenance Requirements s886

Definition 96 – **Extensional Maintenance:** *By extensional maintenance we understand such maintenance which adds new functionalities to the software, i.e., which implements additional requirements* ∎

s887

Example 110 – **Timetable System Extendability:** Assume a release of a timetable software system to implement a requirements that, for example, expresses that shortest routes but not that fastest routes be found in response to a travel query. If a subsequent release of that software is now expected to also calculate fastest routes in response to a travel query, then we say that the implementation of that last requirements constitutes extensional maintenance. ∎

s888

• • •

Whenever a maintenance job has been concluded, the software system is to undergo an extensive acceptance test: a predetermined, large set of (typically thousands of) test programs has to be successfully executed.

8.9.5 Platform Requirements s889

Definition 97 – **Platform:** *By a [computing] platform is here understood a combination of hardware and systems software — so equipped as to be able to execute the software being requirements prescribed — and 'more'* ∎

What the 'more' is should transpire from the next characterisations.

Definition 98 – **Platform Requirements:** *By platform requirements we mean a combination of the following: (i) development platform requirements, (ii) execution platform requirements, (iii) maintenance platform requirements and (iv) demonstration platform requirements* ■

s890

Example 111 – **Space Satellite Software Platforms:** Elsewhere prescribed software for some space satellite function is to satisfy the following platform requirements: shall be developed on a Sun workstation under `Sun UNIX`, shall execute on the military `MI1750` hardware computer running its proprietary `MI1750 Operating System`, shall be maintained at the NASA Houston, TX installation of `MI1750 Emulating Sun Sparc Stations`, and shall be demonstrated on ordinary Sun workstations under `Sun UNIX`. ■

Development Platform Requirements s891

Definition 99 – **Development Platform Requirements:** *By development platform requirements we shall understand such machine requirements which detail the specific software and hardware for the platform on which the software is to be developed* ■

Execution Platform Requirements s892

Definition 100 – **Execution Platform Requirements:** *By execution platform requirements we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be executed* ■

Maintenance Platform Requirements s893

Definition 101 – **Maintenance Platform Requirements:** *By maintenance platform requirements we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be maintained* ■

Demonstration Platform Requirements s894

Definition 102 – **Demonstration Platform Requirements:** *By demonstration platform requirements we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be demonstrated to the customer — say for acceptance tests, or for management demos, or for user training* ■

Discussion s895

Example 111 is rather superficial. And we do not give examples for each of the specific four platforms. More realistic examples would go into rather extensive details, listing hardware and software product names, versions, releases, etc.

8.9.6 Documentation Requirements s896

Definition 103 – **Documentation Requirements:** *By documentation requirements we mean requirements of any of the software documents that together make up software: (i) not only code that may be the basis for executions by a computer, (ii) but also its full development documentation: (ii.1) the stages and steps of* application domain description*, (ii.2) the stages and steps of* requirements prescription*, and (ii.3) the stages and steps of* software design *prior to code, with all of the above including all* validation *and* verification *(incl.,* test*) documents.* *s897 In addition, as part of our wider concept of software, we also include (iii) a comprehensive collection of* supporting documents*: (iii.1)* training manuals*, (iii.2)* installation manuals*, (iii.3)* user manuals*, (iii.4)* maintenance manuals*, and (iii.5–6)* development *and* maintenance logbooks ■

s898
We do not attempt, in our characterisation, to detail what such documentation requirements could be. Such requirements could cover a spectrum from the simple presence, as a delivery, of specific ones, to detailed directions as to their contents, informal or formal.

• • •

8.9.7 Discussion: Machine Requirements s899

We have — at long last — ended an extensive enumeration, explication and, in many, but not all cases, exemplification, of machine requirements. When examples were left out it was because the reader should, by now, be able to easily conjure up such examples.

The enumeration is not claimed exhaustive. But, we think, it is rather representative. It is good enough to serve as a basis for professional software engineering. And it is better, by far, than what we have seen in "standard" software engineering textbooks.

8.10 Opening and Closing Stages s900

acm-rtre-e

8.10.1 Opening Stages s901

Stakeholder Identification and Liaison s902

Requirements Acquisition s903

Requirements Analysis s904

Terminoligisation s905

8.10.2 Closing Stages s906

For completeness, we shall, as in Sects. 7.9.2 on page 128 and 8.10.1, briefly list the closing stages of requirements engineering. They are:

1. requirements verification, model checking and testing – the assurance of properties of the formalisation of the requirements model (Sect. 8.10.2);
2. requirements validation – the validation of the veracity of the informal, i.e., the narrative requirements prescription (Sect. 8.10.2);
3. requirements feasibility and satisfiability (Sect. 8.10.2); and
4. requirements theory formation (Sect. 8.10.2).

Verification, Model Checking and Testing s907

Requirements Validation s908

Requirements Satisfiability & Feasibility s909

Requirements Theory s910

8.10.3 Requirements Engineering Documentation s911

8.10.4 Conclusion s912

8.11 Exercises

See Items 19–22 (of Appendix B, starting Page 197).

Part VI

Closing

9

Conclusion

s913 acm-c

10

Acknowledgements

s919 acm-a

This book has been written after I retired from almost 32 years as professor at The Technical University of Denmark, as of April 1, 2007. I have therefore been basically deprived of the daily, invigorating interaction with dear colleagues at DTU Informatics. Instead I was invited, or, more-or-less, invited myself, to lecture at universities in Nancy[1], Graz[2] and Saarbrücken[3]. The present book underwent a number of iterations while presenting its material at intensive 30 lectures plus 15 afternoons of project tutoring. s920I am therefore profoundly grateful (alphabetically listed) to Bernhard Aichernig (TUG), Hermann Maurer (TUG), Dominique Méry (UHP/INRIA), Wolfgang Paul (UdS), Thomas in der Rieden (UdS) and Franz Wotawa (TUG) for hosting me and for giving me the opportunity to refine the course material and be challenged by bright young people from a dozen European and Asian countries.

Sir Tony Hoare provided ...

[1]Oct.–Dec., 2007, University of Henri Poincare (UHP) and INRIA, France
[2]Oct.–Dec., 2008, Technical University of Graz (TUG), Austria
[3]March 2009, University of Saarland, Germany

11

Bibliographical Notes

s921 [1]

Specification languages, techniques and tools, that cover the spectrum of domain and requirements specification, refinement and verification, are dealt with in Alloy: [87], ASM: [123, 124], B/event B: [1, 31], CafeOBJ: [39, 40, 56, 57], CSP [78, 79, 127, 131], DC [145, 146] (Duration Calculus), Live Sequence Charts [37, 72, 92], Message Sequence Charts [83–85], RAISE [15–17, 58, 59, 61] (RSL), Petri nets [88, 112, 120–122], Statecharts [68–71, 73], Temporal Logic of Reactive Systems [97, 98, 110, 114], TLA+ [93, 94, 102, 103] (Temporal Logic of Actions), VDM [24, 25, 53, 54], and Z [74, 75, 134, 135, 142]. Techniques for integrating "different" formal techniques are covered in [6, 28, 29, 63, 126]. The recent book on Logics of Specification Languages [23] covers ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM and Z.

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
2. ACM. Programming Languages and Pragmatics. *Communications of the ACM*, 9(6), August 1966.
3. Afred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., USA, 1977, Januar 1986.
4. Christopher Alexander. *The Timeless Way of Building*, chapter A Pattern Language. Oxford University Press, New York, 1979.
5. James Allen. *Natural Language Understanding*. Benjamin/Cummings Series in Computer Science. The Benjamin/Cummings Publishing Company, Inc., 2727 Sand Hill Road, Menlo Park, California 94025, USA, 1987.
6. Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
7. Yasuhito Arimoto and Dines 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.
8. John W. Backus and Peter Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–1, 1963.
9. Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria, 20 September 1974.
10. Dines 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.
11. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.

[1] acm-bib

12. Dines 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.
13. Dines 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.
14. Dines Bjørner. Documents: A Domain Analysis. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
15. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
16. Dines 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.
17. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
18. Dines 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. Dines Bjørner. *Domain Engineering*. To be submitted to Springer for evaluation in 2009, Expected published 2010. The methodology part of this textbook is about 200 pages long; that part is supported by a number of example appendices of about 300 pages. The examples is that of an oil industry.
20. Dines Bjørner. *Software Engineering, Vol. I: The Triptych Approach, Vol. II: A Model Development*. To be submitted to Springer for evaluation in 2009, Expected published 2010. Either this book ms. is submitted or that of [19] is submitted. Decision on this to be made before Summer 2009. This book was the basis for guest lectures at Techn. Univ. of Graz, Oct.–Dec. 2008.
21. Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. JAIST Press, March 2009. The monograph contains the following chapters: [41–50].
22. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
23. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [31, 40, 53, 58, 67, 74, 103, 107, 124]*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
24. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer–Verlag, 1978. This was the first monograph on *Meta-IV*. .
25. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
26. Dines Bjørner and Ole N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer–Verlag, 1980.
27. Nikolaj Bjørner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16:227–270, 2000.
28. Eerke A. Boiten, John Derrick, and Graeme Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, England, April 4-7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.
29. Michael J. Butler, Luigia Petre, and Kaisa Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15-18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.
30. Dominique Cansell and Dominique Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [39, 60, 75, 102, 108, 123] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
31. Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages 47–152 in [22]. Springer, 2008.
32. R. Carnap. *The Logical Syntax of Language*. Harcourt Brace and Co., N.Y., 1937.
33. R. Carnap. *Introduction to Semantics*. Harvard Univ. Press, Cambridge, Mass., 1942.
34. R. Carnap. *Meaning and Necessity, A Study in Semantics and Modal Logic*. University of Chicago Press, 1947 (enlarged edition: 1956).
35. David Crystal. *The Cambridge Encyclopedia of Language*. Cambridge University Press, 1987, 1988.

36. O.-J. Dahl, Edsger Wybe Dijkstra, and Charles Anthony Richard Hoare. *Structured Programming.* Academic Press, 1972.
37. Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.
38. J.W. de Bakker. *Control Flow Semantics.* The MIT Press, Cambridge, Mass., USA, 1995.
39. Ražvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. CafeOBJ: Logical Foundations and Methodology. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [30, 60, 75, 102, 108, 123] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
40. Răzvan Diaconescu. *Logics of Specification Languages*, chapter A Methodological Guide to the CafeOBJ Logic, pages 153–240 in [22]. Springer, 2008.
41. 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.
42. 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.
43. 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.
44. 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.
45. 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.
46. 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.
47. 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.
48. 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.
49. 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.
50. 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.
51. Buno Dutertre. Complete Proof System for First–Order Interval Temporal Logic. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, IEEE LiCS, pages 36–43. Piscataway, NJ, USA, IEEE CS, 1995. .
52. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1996. 2nd printing.
53. John S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages 453–487 in [22]. Springer, 2008.
54. John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using* `VDM-SL`. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.
55. Chris Fox. *The Ontology of Language: Properties, Individuals and Discourse.* . CSLI Publications, Center for the Study of Language and Information, Stanford University, California, ISA, 2000.
56. K. Futatsugi, A.T. 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.
57. Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* AMAST Series in Computing – Vol. 6.

World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, SINGAPORE 596224. Tel: 65-6466-5775, Fax: 65-6467-7667, E-mail: wspc@wspc.com.sg, 1998.
58. Chris George and Anne E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages 349–399 in [22]. Springer, 2008.
59. Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
60. Chris W. George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [30,39,75,102,108,123] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
61. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
62. Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2002. 2nd Edition.
63. Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1-3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
64. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
65. Michael Hammer and James A. Champy. *Reengineering the Corporation: A Manifesto for Business Revolution*. HarperCollins*Publishers*, 77–85 Fulham Palace Road, Hammersmith, London W6 8JB, UK, May 1993. 5 June 2001, Paperback.
66. Michael Hammer and Stephen A. Stanton. *The Reengineering Revolutiuon: The Handbook*. HarperCollins*Publishers*, 77–85 Fulham Palace Road, Hammersmith, London W6 8JB, UK, 1996. Paperback.
67. Michael R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages 299–347 in [22]. Springer, 2008.
68. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
69. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
70. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
71. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.
72. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
73. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
74. Martin C. Henson, Moshe Deutsch, and Steve Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages 489–596 in [22]. Springer, 2008.
75. Martin C. Henson, Steve Reeves, and Jonathan P. Bowen. Z Logic and its Consequences. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [30, 39, 60, 102, 108, 123] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
76. Charles Anthony Richard Hoare. The Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12(10):567–583, Oct. 1969.
77. Charles Anthony Richard Hoare. Notes on data structuring. In *[36]*, pages 83–174, 1972.
78. Tony Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
79. Tony Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [78]. See also `http://www.usingcsp.com/`.
80. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Reading. Addison-Wesley, 1979.
81. V. Daniel Hunt. *Process Mapping: How to Reengineer Your Business Processes*. John Wiley & Sons, Inc., New York, N.Y., USA, 1996.
82. IEEE Computer Society. IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical report, IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614, 1990.

83. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
84. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
85. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
86. J. Mike Jacka and Paulette J. Keller. *Business Process Mapping: Improving Customer Satisfaction.* John Wiley & Sons, Inc., New York, N.Y., USA, 2002.
87. Daniel Jackson. *Software Abstractions Logic, Language, and Analysis.* The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
88. Kurt Jensen. *Coloured Petri Nets*, volume 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science.* Springer–Verlag, Heidelberg, 1985, revised and corrected second version: 1997.
89. C. B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, 1980.
90. C. B. Jones. *Systematic Software Development — Using VDM.* Prentice-Hall, 1986.
91. C. B. Jones. *Systematic Software Development — Using VDM, 2nd Edition.* Prentice-Hall, 1989.
92. Jochen Klose and Hartmut Wittke. An automata based interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, LNCS 2031, pages 512–527. Springer-Verlag, 2001.
93. Leslie Lamport. The Temporal Logic of Actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995.
94. Leslie Lamport. *Specifying Systems.* Addison–Wesley, Boston, Mass., USA, 2002.
95. J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault–Tolerant Systems.* Springer–Verlag, Vienna, 1992. In English, French, German, Italian and Japanese.
96. Zohar Manna, Anuchit Anuchitanukul, Nikolaj S. Bjørner, Anca Browne, Edward Chang, Michael Colon, Luca de Alfaro, Harish Devarajan, Henny Sipma, and Tomas Uribe. STeP: The Stanford Temporal Prover. technical STAN-CS-TR-94-1518, Computer Science Department, Stanford University, California, USA, July 1994.
97. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications.* Addison Wesley, 1991.
98. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety.* Addison Wesley, 1995.
99. Zohar Manna and Amir Pnueli. Temporal Verification of Reactive Systems: Progress. In *Internet.* Published: http://theory.stanford.edu/˜zm/tvors3.html, 1996.
100. John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
101. D. H. Mellor and Alex Oliver. *Properties.* Oxford Readings in Philosophy. Oxford Univ Press, , May 1997. ISBN: 0198751761, 320 pages, Amazon price: US$ 19.95.
102. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [30, 39, 60, 75, 108, 123] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
103. Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages 401–451 in [22]. Springer, 2008.
104. R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, Halsted Press/John Wiley, New York, 1976.
105. C. Morris. Foundations of the theory of signs. In *International Encyclopedia of Unified Science.* 1(2), Univ. of Chicago Press, 1938.
106. C. Morris. *Signs, Languages and Behaviour.* G. Brazillier, New York, 1955.
107. T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. *Logics of Specification Languages*, chapter CASL – the Common Algebraic Specification Language, pages 241–298 in [22]. Springer, 2008.
108. Till Mossakowski, Anne E. Haxthausen, Don Sanella, and Andzrej Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [30,39,60,75,102,123] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
109. P.D. Mosses. The mathematical semantics of Algol 60. PRG 12, Programming Research Group, Jan. 1974.
110. Ben C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, England, 1986.
111. P. Naur and B. Randall, editors. *Software Engineering: The Garmisch Conference.* NATO Science Committee, Brussels, 1969.

112. Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
113. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice.* Prentice–Hall, 2nd edition, 2001.
114. Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, IEEE CS FoCS, pages 46–57. Providence, Rhode Island, IEEE CS, 1977. .
115. Roger S. Pressman. *Software Engineering, A Practitioner's Approach.* International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.
116. Martin Pěnička and Dines 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.
117. Martin Pěnička, Albena Kirilova Strupchanska, and Dines 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.
118. Anthony Ralston and Philip Rabinowitz. *A First Course in Numerical Analysis.* Dover Pubns; 2nd rev edition, January 2001. ISBN: 048641454X.
119. Brian Randell. On Failures and Faults. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Formal Methods Europe, Springer–Verlag, 2003. Invite Paper.
120. Wolfang Reisig. *A Primer in Petri Net Design.* Springer Verlag, March 1992. 120 pages.
121. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science.* Springer Verlag, May 1985.
122. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets.* Springer Verlag, December 1998. xi + 302 pages.
123. Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [30,39,60,75,102,108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
124. Wolfgang Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [22]. Springer, 2008.
125. John C. Reynolds. *The Semantics of Programming Languages.* Cambridge University Press, 1999.
126. Judi M.T. Romijn, Graeme P. Smith, and Jaco C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM. ISBN 3-540-30492-4.
127. A. W. Roscoe. *Theory and Practice of Concurrency.* C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. Now available on the net: http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.
128. Bertrand Russell. On Denoting. *Mind*, 14:479–493, 1905.
129. Bertrand 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.
130. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.
131. Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach.* Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.
132. D.S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia*, pages 19–46, 1971.
133. Ian Sommerville. *Software Engineering.* Addison-Wesley, 6th edition, 1982–2001.
134. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, January 1988.
135. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
136. 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.
137. Albena Kirilova Strupchanska, Martin Pěnička, and Dines 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.

138. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
139. University of California at Irvine. Business Process Re–engineering, Administrative and Business Services Department. Electronically, on the Web: `http://www.abs.uci.edu/depts/vcabs/4-1.html`, 2004.
140. Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 2000. 2nd Edition.
141. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.
142. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
143. J. C. P. Woodcock and M. Loomes. *Software Engineering Mathematics*. Pitman, London, 1988.
144. Heinz Zemanek. Semiotics and Programming Languages. In *[2]*, pages 139–143, 1966.
145. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.
146. Chao Chen Zhou, Charles Anthony Richard Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

Part VII

Appendices

A

Lecturers' Guide to Using This Book

Pages XIII–XIV suggests a 14-18 lectures schedule. Slight variants of this schedule has been lectured over at Univ. of Henri Poincaré/INRIA at Nancy, France, in the fall of 2007, at Techn. Univ. of Graz, Austria, in the fall of 2008, at Univ. of Saarland, Saarbrücken, Germany, March 2009, at Univ. of Edingburgh, Scotland, Sept.–Oct. 2009, and at Tokyo Univ., Japan, Nov.–Dec. 2009.

A.1 Use of Textbook

- Since the lecture slides, see next section, Sect. A.2, only present very essentials of the text lecturers must study the textbook carefully and students should regularly consult the textbook: the latter by (students) attempting to more-or-less cursorily read the relevant text before the lecture and by re-reading it after the lecture.
- The textbook provides more than 80 pages of examples scattered all over the text. An 'Examples Index', Sect. D.3 (Pages 227–228), lists all examples by 'Example Name'. The lecturer need not cover all these examples.
- Lecturers' version of the textbook[1] provides display line and margin references to the approximately 900 'Lecture Slides': s#i is placed at book text (some of) which also appears on slide #i. s#i
- Lectures based on — and studies of — the textbook rely on the students and readers also pursuing one of the projects listed in Sect. B.2. To that end each chapter of the textbook refers to project exercise items listed in Sect. B.1. There is at least one project exercise item per lecture. In other words: the lecturer is well advised in making sure to emphasise the methodology underlying solutions to these project exercise items.

A.2 Use of Lecture Slides

- Textbook Pages XIII–XIV provide a proposal for composing a course of the textbook in the form of between 14 and 18 lectures.
- Lecturers can obtain, from the publisher, access to "pre-packaged" sets of lecture slides, in either postscript or pdf form.
- For reasons of copyright it is not possible to provide students with access to electronic versions of these slides.[2]
- Some of the proposed lectures may seem a bit long for a session of two 45–50 minute lectures. The lecturer must decide on this length and may therefore be forced to carefully decide whether to skip some examples (which is advised) or to skip some methodology text (which may be

[1] This 'Lecturers Version' is available, electronically, upon request, from the publisher.

[2] The author will try persuade the publisher to allow publisher-registered lecturers to give their students read-access to lecturers' slides during the lecturing and pre-exam period.

possible). In such a case it may be possible for the lecturer to prepare a lecture (.ps or .pdf) file with just a subset of the pages of the "pre-packaged" lecture files.

- All examples contain both itemized or enumerated narrative, English text and, on subsequent slides, oftentimes enumerated formulas. In early lectures it is advised to first "read" the narrative texts, then show (or "read)) the formulas. In later lectures the lecturer can skip the narrative texts and "narrate" the formulas (as if they were their narrative counterparts). Thus basically half the examples slides can be omitted in most lectures.
- The point to be made here, by the lecturer — since it is being made by the book (and its author) — is that (even model-oriented) formal specifications can be "naturally" read, that is, as if they were narratives. (But formalisations can never replace the narratives: the latter must be expressed in such a way as to provide for easy understanding (i.e., reading) by all stake-holders.

A.3 Miscellaneous

-
-
-
-

B

Lecturers' Guide to Projects

This little book contains only one exercise. Normally exercises are listed at the end of each chapter. Instead we shall use this chapter to outline a number of project topics and how the reader (including course students) might tackle one of these projects.

B.1 Project Assignments: Textbook Topic-by-Topic

This section is addressed directly at project students.

There are two ways in which a project makes sense: either, for a self-study, by a motivated reader, or, as a similarly indispensable part of a course.

The idea is, for students, to first choose one of the project or sub-project topics listed in Sect. B.2 (take a brief, that is, hasty look at bold-faced terms of that list now); then, as lectures (and/or your reading of the textbook) progresses, from section-to-section, that is from methodology topic-to-topic, you, the student, try apply the method covered by these sections to the chosen project. This means that when you, the student, have studied the referenced sections itemized below then you, the student, try your hand at the "exercises" written in *slanted text next to the section references:*

1. Chapter 2, Sect. 2.2:

    Formulate the essence of your chosen, i.e., the project domain.

    Cf. Examples 1–9 (Pages 8–9).
2. Chapter 2, Sect. 2.3:

    Enumerate informal, that is natural (i.e., national) language descriptions which ever phenomena (entities, operations, events, behaviour) comes to your mind.

    Try sort your enumerated list such that simplest facts are described first (needing no reference to facts defined later), etc. — as in Example 10.
3. Chapter 4, Sect. 4.1.3 (Pages 23–31). Even though you may not yet be (fully) capable of formalising your chosen abstractions:

    Identify two or three domain phenomena (or concepts) for each of which you then conceive of a suitable abstraction and narrate and, however "feebly" formalise it.

    Cf. the style of Examples 11–14 of Sect. 4.1.3, Pages 24–31).
4. Chapter 4, Sects. 4.2–4.3:

    Discuss, for your chosen two or three examples, resulting from your solution to Item 3, the following attributes of your solutions:
    - *model-orientedness versus property-orientedness*[1],
    - *analogic, analytic and iconic,*

[1](that is, try have your solutions to Item 3 questions reflect a suitable combination of these and the further attributes as covered in Sect. 4.3)

- *descriptive (if a domain specification) and prescriptive (if a requirements specification), and*
- *extensional and intentional.*

5. Chapter 5, Sect. 5.2:

   Narrate and formalise concrete syntax(es) as well as abstract syntax(es) for at least one syntactic type and at least one semantic type for chosen (however small) and respective phenomena (or concepts) of your chosen domain.

   That is, continue now towards further and further descriptions of your chosen domain.
6. Chapter 5, Sect. 5.3:

   Narrate and formalise one denotational and one behavioural semantics for chosen (however small) and respective phenomena (or concepts) of your chosen domain.

   As for Item 5: continue now towards further and further descriptions of your chosen domain.
7. Chapter 6, Sect. 6.3.1:

   Identify a small number of (hitherto not already identified) atomic and a small number of composite simple entities and narrate and formalise these,

   while

   emphasising, in your narration, which are attributes, which are sub-entities and which are mereologies of your described composite simple entities — as well as which are the attributes of atomic simple entities.

   Make sure that your enumerated narrative statements "fit, hand-in-glove" with similarly enumerated formulas.
8. Chapter 6, Sect. 6.4.1:

   Identify a small number of (hitherto not already identified) domain operations, narrate and formalise these, both by explicit model-oriented definitions and by pre/post conditions.

   Make sure that your enumerated narrative statements "fit, hand-in-glove" with similarly enumerated formulas.
9. Chapter 6, Sect. 6.4.2:

   Identify a small number of (hitherto not already identified) domain events.

   Later, as part of your answer to Item 10, you shall then also formalise these events.
10. Chapter 6, Sect. 6.3.2:

    Identify a small number of (hitherto not already identified) domain behaviours and narrate and formalise these

    while including the formalisation of identifier domain events as identified in Item 9.
11. Chapter 7, Sect. 7.2:

    Identify a small number of (hitherto not already identified) business processes and provide rough sketch narratives.
12. Chapter 7, Sect. 7.3:

    Identify a small number of (hitherto not already identified) domain intrinsic facets and narrate and formalise these.
13. Chapter 7, Sect. 7.4:

    Identify a small number of (hitherto not already identified) domain support technology facets and narrate and formalise these.
14. Chapter 7, Sect. 7.5:

    Identify a small number of (hitherto not already identified) domain management and organisation facets and narrate and formalise these.
15. Chapter 7, Sect. 7.6:

    Identify a small number of (hitherto not already identified) domain rules and regulations facets and narrate and formalise these.

16. Chapter 7, Sect. 7.7:

    Identify a small number of (hitherto not already identified) domain script (license or contract) facets and narrate and formalise these.

17. Chapter 7, Sect. 7.8:

    Identify a small number of (hitherto not already identified) human behaviour facets and narrate and formalise these.

18. Chapter 7, Sects. 7.1–7.9:

    Consolidate your narratives and formalisations of the answers to Items 11–17 into a nice, clean, well-structured domain description.

    It is OK to leave some operations, events and behaviours only “hinted at”, but the idea that a completion is straightforward should be convincing.

19. Chapter 8, Sect. 8.4:

    Rough sketch (i.e., narrate) two reasonably distinct business process re-engineerings.

20. Chapter 8, Sect. 8.6:

    Develop, based on the answer to Item 19, and in stages of development, a domain requirements: projection, instantiation, determination, extension and fitting.

21. Chapter 8, Sect. 8.8:

    Identify shared entities, operations, events and behaviours, and suggest, narrate and formalise

    shared entity initialisation and refreshment requirements,
    shared operation requirements and
    shared event and behaviour requirements.

22. Chapter 8, Sects. 8.1–8.10:

    Consolidate your narratives and formalisations of the answers to Items 19–21 into a nice, clean, well-structured domain description.

    It is OK to leave some operations, events and behaviours only “hinted at”, but the idea that a completion is straightforward should be convincing.

23. Page XI: The Triptych Phases:

    Name the three phases of software engineering.

24. Chapter 7, Sect. 7.9: The Domain Engineering Stages:

    Name 7–9 of the stages of domain engineering.
    Give a 1-3 line characterisation of 5–6 of the domain engineering stages.

25. Chapter 7, Sect. 7.1 (Sects. 7.2–7.8): The Main Facets of Domain Modelling:

    Name 5–6 facets of domain modelling.
    Give a 1-3 line characterisation of the domain modelling facets.

26. Chapter 8, Sect. 8.10: The Requirements Engineering Stages:

    Name 7–9 of the stages of requirements engineering.
    Give a 1-3 line characterisation of 5–6 of the requirements engineering stages.

27. Chapter 8, Sect. 8.2: The Core Stages of Requirements Modelling:

    Name the core stages of requirements modelling.
    Give a 1-3 line characterisation of these stages of requirements modelling.

28. Chapter 8, Sect. 8.6: The Main Facets of Domain Requirements Modelling:

    Name 4–5 facets of domain requirements modelling.
    Give a 1-3 line characterisation of each the these facets of domain requirements modelling.

29. Chapter 8, Sect. 8.8: The Three Facets of Interface Requirements Engineering:

    Name the three facets of interface requirements modelling.
    Give a 1-3 line characterisation of each of the three facets of interface requirements modelling.

30. Chapter 6, Sects. 6.3–6.4: On A Specification Ontology:
    Identity the four aspects of the specification ontology carried forward by this book. Characterise these.
31. Chapter 6, Sect. 6.3: Atomic and Composite Simple, Discrete Entities
    Characterise the concepts of atomic and composite simple, discrete entities.
    Characterise the concepts of entity attributes, sub-entities of composite entities and their mereology.

B.2 Project Topics

We shall list only a few terms from "the languages" of the domains otherwise labelled below. We leave it to the project participants (i) to significantly extend this terminology, and to classify the individual terms as designating (ii) simple entities (whether continuous or discrete, and, if discrete, whether atomic or composite), operations, events and behaviours, or (iii) syntactic or semantic quantities.

B.2.1 Infrastructure Components

1. **Airports:** check-in, baggage, baggage handling, baggage conveyour belts, passenger, ticket, boarding card, security control, gate, aircraft, fuelling, cleaning, passenger unloading and loading, catering, etc.
   Airport management as the monitoring and control of flow of passengers, aircraft, baggage and information (including monitoring and control information) in airports with interfaces to air traffic control, air lines and passengers.
2. **Air Traffic:** aircrafts, ground control tower, terminal control tower, regional and continental control centers, approaching aircraft, landing and take-off, departing aircraft, holding patterns, communications between aircraft and control towers and centers, etc.
   Air Traffic, whether monitored and/or controlled. National and international air flight and aircraft guidance rules and regulations (GAO).
3. **Assembly Manufacturing:** products, product parts and sub-parts, (product) bill-of-material, parts-explosion, machine assembly, parts storage, product warehouse,
4. **The Consumer Market & Supply Chain:** Can be "decomposed" into sub-group projects:
   a) **Consumers:** being made aware of merchandise offerings, by retailers, through advertisement and "window-shopping", making inquiries, receiving offers, ordering merchandise, inspecting merchandise received, accepting and paying for merchandise, returning reject offers, returning received and paid-for merchandise for repair or replacement, etc.
   b) **Retailers:** advertising merchandise (aimed at consumers), ordering merchandise (aimed at wholesalers), replying to consumer inquiries, offering merchandise to consumers, delivering (packaging and//or sending/posting) merchandise to consumers, billing consumers, accepting returning, unaccepted goods, accepting returned merchandise for repiar or replacement (or refund), receiving ordered goods from wholesalers, warehousing these goods, updating sales catalogues, paying wholesaler bills, etc.
   c) **Wholesalers:**
   d) **Payment:**
   e) **Producers:**
   f) **Distribution Chain:**
5. **Container Line Industry:**
   a) **Containers:**
   b) **Container Vessels:**
   c) **Container Terminal Ports:**
   d) **Bill of Ladings / Way Bills:**
   e) **Container Lines:**

   f) **Sea Routes:**
   g) **Senders and Receivers:**
   h) **Interface to Logistics Firms:**
6. **The Financial Service Industry:**
   a) **Banks:**
   b) **Commodities Exchange:**
   c) **Portfolio Management:**
   d) **Insurance:**
   e) **Credit Cards:**
7. **Harbours:**
8. **Hospitals:**
9. **Logistics:**
10. **Pipe Lines:**
11. **Railways:**

B.2.2 Components of Components of ... Infrastructure Components

The emphasis of the below-enumerated project topics is that they reflect the following 'problem frames':

- **Reactive systems:** Items 12, 13, 18, 19 and 20.
  These projects are best pursued in conjunction with a course on *real-time, embedded, safety critical systems.*
- **Workpiece systems:** Items 16, 14, ...
  These projects are best pursued in conjunction with a course on *computer human interface systems.*
- **Transformation systems:** Items ...
  These projects are best pursued in conjunction with a course on *compiler and interpreter systems.*
- **Information systems:** Items 14.15, 17.
  These projects are best pursued in conjunction with a course on *database systems.*

MORE TO COME

Here is the list:

12. **Airport Baggage Handling:**
13. **Automatic Teller Machine:**
14. **Container Stowage:**
15. **Credit Slip Clearance:**
16. **Document Handling:**
17. **Library Item Management:**
18. **Rail-Road Level Crossing:**
19. **Railway Track Interlocking:**
20. **Road Intersection Semaphore Monitoring and Control:**

MORE TO COME

B.3 Project Groups

B.4 Weekly Project Reports

Section B.1 outlined textbook exercises, section-by-section, that is, the lecture-by-lecture assignments that are proposed for the students.

"Answers" to these assignments are to be recorded in a group report. Such a report could, for example. be structured by chapter and section headings according to those of the itemised assignments (of Sect. B.1).

This would mean that answers to Items 11-18 and 19-22 together constitute more-or-less proper domain descriptions, respectively requirements prescriptions.

B.5 Project Tutoring

B.5.1 Weekly "Class" Tutoring Sesssion

B.5.2 Individual Project Group Tutoring Sesssion

B.6 Project Report Format

B.7 Course Project Phases

B.7.1 Introductory Concepts Phase: Lectures 1–7

B.7.2 Domain Engineering Phase: Lectures 8–12

B.7.3 Requirements Engineering Phase: Lectures 13–16

B.7.4 Final Course Phase

B.8 Course Evaluation

B.8.1 Course Exam

B.8.2 Project Evaluation

s922

C

An RSL Primer

s923

This is an ultra-short introduction to the RAISE Specification Language, RSL.

C.1 Types

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

C.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.

s924

Basic Types

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

Composite Types

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".

s925

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:

Composite Type Expressions

[7] A-**set**
[8] A-**infset**
[9] $A \times B \times ... \times C$
[10] A^*
[11] A^ω
[12] $A \xrightarrow[m]{} B$
[13] $A \to B$
[14] $A \stackrel{\sim}{\to} B$
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian $B \times C \times ... \times D$, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., ($A \xrightarrow[m]{} B$), 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.

C.1.2 Type Definitions s926

Concrete Types

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

Type Definition

```
type
   A = Type_expr
```

s927

Some schematic type definitions are:

Variety of Type Definitions

```
[1] Type_name = Type_expr /* without |s or subtypes */
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
        mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
        ... |
        mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

s928

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

Record Types

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

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

```
axiom
   ∀ 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
```

s929

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:

Subtypes

```
type
   A = {| b:B • 𝒫(b) |}
```

s930

Sorts — Abstract Types

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

Sorts

type
A, B, ..., C

C.2 The RSL Predicate Calculus s931

C.2.1 Propositional Expressions

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

Propositional Expressions

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

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives
s932 (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

C.2.2 Simple Predicate Expressions

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

Simple Predicate Expressions

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

s933 are simple predicate expressions.

C.3 Quantified Expressions

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

Quantified Expressions

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

are quantified expressions — also being predicate expressions.

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

C.4 Concrete RSL Types: Values and Operations

s934

C.4.1 Arithmetic

Arithmetic

type
  Nat, **Int**, **Real**
value
  $+,-,*$: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\stackrel{\sim}{\to}$**Nat** | **Int**×**Int**$\stackrel{\sim}{\to}$**Int** | **Real**×**Real**$\stackrel{\sim}{\to}$**Real**
  $<,\leq,=,\neq,\geq,>$ (**Nat**|**Int**|**Real**) → (**Nat**|**Int**|**Real**)

s935

C.4.2 Set Expressions

Set Enumerations

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

Set Enumerations

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

s936

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.

Set Comprehension

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

s937

C.4.3 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:

Cartesian Enumerations

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

s938

C.4.4 List Expressions

List Enumerations

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

List Enumerations

```
    {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
    {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

    ⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the
s939 former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

List Comprehension

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

s940

C.4.5 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:

Map Enumerations

```
type
```

```
    T1, T2
    M = T1 →m T2
value
    u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
    [], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M
```

s941

Map Comprehension

The last line below expresses map comprehension:

Map Comprehension

```
type
    U, V, X, Y
    M = U →m V
    F = U ~→ X
    G = V ~→ Y
    P = U → Bool
value
    comprehend: M×F×G×P → (X →m Y)
    comprehend(m,F,G,P) ≡
        [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

s942

C.4.6 Set Operations

Set Operator Signatures

Set Operations

```
value
    19  ∈: A × A-infset → Bool
    20  ∉: A × A-infset → Bool
    21  ∪: A-infset × A-infset → A-infset
    22  ∪: (A-infset)-infset → A-infset
    23  ∩: A-infset × A-infset → A-infset
    24  ∩: (A-infset)-infset → A-infset
    25  \: A-infset × A-infset → A-infset
    26  ⊂: A-infset × A-infset → Bool
    27  ⊆: A-infset × A-infset → Bool
    28  =: A-infset × A-infset → Bool
    29  ≠: A-infset × A-infset → Bool
    30  card: A-infset ~→ Nat
```

s943

Set Examples

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** $\{a,b,c\} = 3$

s944

Informal Explication

19. $\in$: The membership operator expresses that an element is a member of a set.
20. $\notin$: The nonmembership operator expresses that an element is not a member of a set.
21. $\cup$: 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.
22. $\cup$: 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.
23. $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24. $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator
s945 gives the set whose members are in some of the operand sets.
25. $\setminus$: 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.
26. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. $\subset$: 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.
28. $=$: The equal operator expresses that the two operand sets are identical.
29. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

s946

Set Operator Definitions

The operations can be defined as follows ($\equiv$ is the definition symbol):

Set Operation Definitions

value
$s' \cup s'' \equiv \{\ a \mid a{:}A \bullet a \in s' \vee a \in s''\ \}$
$s' \cap s'' \equiv \{\ a \mid a{:}A \bullet a \in s' \wedge a \in s''\ \}$
$s' \setminus s'' \equiv \{\ a \mid a{:}A \bullet a \in s' \wedge a \notin s''\ \}$
$s' \subseteq s'' \equiv \forall\ a{:}A \bullet a \in s' \Rightarrow a \in s''$
$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists\ a{:}A \bullet a \in s'' \wedge a \notin s'$
$s' = s'' \equiv \forall\ a{:}A \bullet a \in s' \equiv a \in s'' \equiv s{\subseteq}s' \wedge s'{\subseteq}s$
$s' \neq s'' \equiv s' \cap s'' \neq \{\}$
card $s \equiv$
  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 ∗/

s947

C.5 Cartesian Operations

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
```

s948

C.5.1 List Operations

List Operator Signatures

List Operations

```
value
    hd: Aω ~→ A
    tl: Aω ~→ Aω
    len: Aω ~→ Nat
    inds: Aω → Nat-infset
    elems: Aω → A-infset
    .(.): Aω × Nat ~→ A
    ⌢: A∗ × Aω → Aω
    =: Aω × Aω → Bool
    ≠: Aω × Aω → Bool
```

s949

List Operation Examples

List Examples

```
examples
    hd⟨a1,a2,...,am⟩=a1
    tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
    len⟨a1,a2,...,am⟩=m
    inds⟨a1,a2,...,am⟩={1,2,...,m}
    elems⟨a1,a2,...,am⟩={a1,a2,...,am}
```

```
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

s950

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
s951 larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

s952 The operations can also be defined as follows:

List Operator Definitions

List Operator Definitions

```
value
   is_finite_list: A^ω → Bool

   len q ≡
      case is_finite_list(q) of
         true → if q = ⟨⟩ then 0 else 1 + len tl q end,
         false → chaos end

   inds q ≡
      case is_finite_list(q) of
         true → { i | i:Nat • 1 ≤ i ≤ len q },
         false → { i | i:Nat • i≠0 } end

   elems q ≡ { q(i) | i:Nat • i ∈ inds q }

   q(i) ≡
      if i=1
         then
            if q≠⟨⟩
               then let a:A,q′:Q • q=⟨a⟩⌢q′ in a end
               else chaos end
         else q(i−1) end

   fq ⌢ iq ≡
         ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
          | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
```

pre is_finite_list(fq)

$iq' = iq'' \equiv$
inds $iq' =$ **inds** $iq'' \wedge \forall$ i:**Nat** • $i \in$ **inds** $iq' \Rightarrow iq'(i) = iq''(i)$

$iq' \neq iq'' \equiv \sim(iq' = iq'')$

s953

C.5.2 Map Operations

Map Operator Signatures and Map Operation Examples

Map Operations

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

dom: M → A-**infset** [domain of map]
dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

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

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

∪: 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**

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

s954

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. s955
- \: 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.

s956

Map Operation Redefinitions

The map operations can also be defined as follows:

Map Operation Redefinitions

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
     [ a↦b | a:A,b:B •
        a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
               a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
     dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
     [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
     pre rng m ⊆ dom n
```

C.6 λ-Calculus + Functions s957

C.6.1 The λ-Calculus Syntax

λ-Calculus Syntax

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

s958

C.6.2 Free and Bound Variables

Free and Bound Variables

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

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

s959

C.6.3 Substitution

In RSL, the following rules for substitution apply:

Substitution

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,
  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λ*x*•*P*)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λ*y*• **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)).

s960

C.6.4 α-Renaming and β-Reduction

α and β Conversions

- α-renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

s961

C.6.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
    A, B, C
value
    obs_B: A → B,
    obs_C: A → C,
    gen_A: B×C → A
```

s962

C.6.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions

value
  f: Arguments $\rightarrow$ Result
  f(args) $\equiv$ DValueExpr

  g: Arguments $\stackrel{\sim}{\rightarrow}$ Result
  g(args) $\equiv$ ValueAndStateChangeClause
  pre P(args)

s963 Or functions can be defined implicitly:

Implicit Function Definitions

value
  f: Arguments $\rightarrow$ Result
  f(args) **as** result
  post P1(args,result)

  g: Arguments $\stackrel{\sim}{\rightarrow}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

The symbol $\stackrel{\sim}{\rightarrow}$ 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.

C.7 Other Applicative Expressions s964

C.7.1 Simple let Expressions

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

Let Expressions

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

is an "expanded" form of:

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

s965

C.7.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive **let** Expressions

  let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let $f = \mathbf{Y}F$ **in** $B(f,a)$ **end**

where:

$F \equiv \lambda g \bullet \lambda a \bullet (E(g))$ and $YF = F(YF)$

s966

C.7.3 Predicative let Expressions

Predicative **let** expressions:

Predicative **let** Expressions

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

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

s967

C.7.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

Patterns

let $\{a\} \cup s = set$ **in** ... **end**
let $\{a,\_\} \cup s = set$ **in** ... **end**

let $(a,b,...,c) = cart$ **in** ... **end**
let $(a,\_,...,c) = cart$ **in** ... **end**

let $\langle a\rangle ^\frown \ell = list$ **in** ... **end**
let $\langle a,\_,b\rangle ^\frown \ell = list$ **in** ... **end**

let $[a \mapsto b] \cup m = map$ **in** ... **end**
let $[a \mapsto b,\_] \cup m = map$ **in** ... **end**

s968

C.7.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals

if b_expr **then** c_expr **else** a_expr
end

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

if b_expr_1 **then** c_expr_1

elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...
elsif b_expr_n **then** c_expr_n **end**

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

s969

C.7.6 Operator/Operand Expressions

Operator/Operand Expressions

⟨Expr⟩ ::=
  ⟨Prefix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
  | ⟨Expr⟩ ⟨Suffix_Op⟩
  | ...
⟨Prefix_Op⟩ ::=
  − | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
  = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
  | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ^ | † | ∘
⟨Suffix_Op⟩ ::= !

C.8 Imperative Constructs s970

C.8.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Statements and State Change

  Unit
value
  stmt: **Unit** → **Unit**
  stmt()

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, **stmt**, denote state-to-state changing functions.
- Writing () as “only” arguments to a function “means” that () is an argument of type **Unit**.

s971

C.8.2 Variables and Assignment

Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

C.8.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and **skip**

2. **skip**
3. stm_1;stm_2;...;stm_n

C.8.4 Imperative Conditionals

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**

s972

C.8.5 Iterative Conditionals

Iterative Conditionals

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

C.8.6 Iterative Sequencing

Iterative Sequencing

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

C.9 Process Constructs s973

C.9.1 Process Channels

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

Process Channels

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the
s974 designated types (A and B).

C.9.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

Process Composition

P ‖ Q	Parallel composition
P ⌷ Q	Nondeterministic external choice (either/or)
P ⌈⌉ Q	Nondeterministic internal choice (either/or)
P ╫ Q	Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external (⌷) or internal (⌈⌉). The interlock (╫) composition expresses that the two processes
s975 are forced to communicate only with one another, until one of them terminates.

C.9.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events

c ?, k[i] ?	Input
c ! e, k[i] ! e	Output

expresses the willingness of a process to engage in an event that "reads" an input, respectively
s976 "writes" an output.

C.9.4 Process Definitions

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.

Process Definitions

value
P: **Unit** → **in** c **out** k[i]
Unit
Q: i:KIdx → **out** c **in** k[i] **Unit**

$P() \equiv ... \, c \, ? \, ... \, k[\,i\,] \, ! \, e \, ...$
$Q(i) \equiv ... \, k[\,i\,] \, ? \, ... \, c \, ! \, e \, ...$

The process function definitions (i.e., their bodies) express possible events.

C.10 Simple RSL Specifications

s977

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

Simple RSL Specifications

type
...
variable
...
channel
...
value
...
axiom
...

s978

D

Indexes

D.1 Concept Index

D.2 Definition Index

D.3 Example Index

D.4 Symbol Index