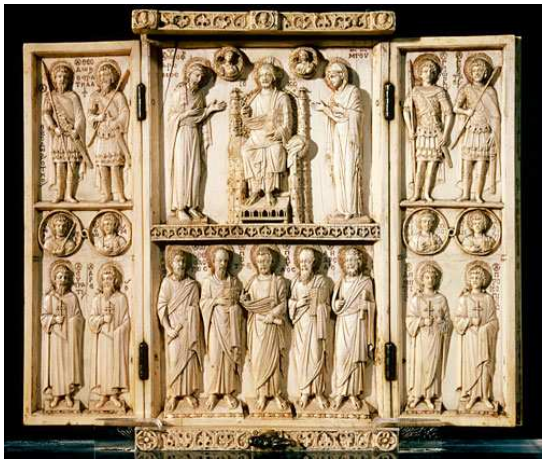
Dines Bjørner

SOFTWARE ENGINEERING

Volume I: The Triptych Approach

March 26, 2008. Compiled December 17, 2008, 15:56



Harbaville Altar, Constantinople; middle 10th Century

To be submitted, late 2008, for evaluation, to:

Springer
Berlin Heidelberg New York
Hongkong London
Milan Paris Tokyo

Document History

- Version 1 released March 26, 2008:
  - ⋆ A first "vastly incomplete" draft of this document was conceived March 26, 2008 and essential parts of Appendices F–K and N–P were "lifted" from [40].
- Version 2 released April 19, 2008:
  - ⋆ On Sunday April 6, 2008, a complete reorganisation of the material assembled and written and rewritten by then took place — resulting in basically the current structure.
  - ⋆ A week, April 6–11, 2008, was then spent on "fattening" the syntactic structure of this textbook. The Pre- and Postlude appendices were added to Parts V and VI.
  - ⋆ "Serious", tentatively "concluding draft" work on Chap. 1 started on Saturday April 12, 2008.
  - ⋆ Work on Chap. 1 and Appendix E progressed significantly during the week of April 12–19, 2008.
- Version 3 released May 7, 2008:
  - ⋆ Draft copy notice inserted.
  - ⋆ Cross-referecing between Vol. 1 text and Vol. 1 slides pages.
    So far no check has been made for "synchroneity".
  - ⋆ Thus Vol. 1 text margin numbers refer to Vol. 2 slide numbers.
  - ⋆ Notes on 'A Possible (12 week) Course Plan' inserted into Preface, pages 11–14.
  - ⋆ Lecture plan inserted as first four slides: 2–6 incl.
- Version 6 released July 20, 2008:
  - ⋆ Worked on Appendix H.
  - ⋆ I am, as of today, July 20, 2008, not happy with Sect. H.2.
  - ⋆ It's treatment of 'states' is too long-winded.
  - ⋆ I believe my ideas on Sect. H.4 will change the former sections.
  - ⋆ I am starting on Sect. H.4 on page 399 tomorrow, July 21, 2008.

Hieronymus Bosch, 1450-1516: Garden of Eden

Dedication

BEING CONTEMPLATED

Jørgen Haugen Sørensen, 1934 ...

The Verdun Altar, 1181, Master Nicolas

Hans Memling, ca. 1430/40 – 1494, The Last Judgement 1466-1473

Hieronymus Bosch, 1450-1516: The Crucifixion of Saint Julia

Preface

A Different Textbook !

This textbook shall teach you a modern, mathematics-based approach to software development, from earliest conception to basic outlines of software architectures: from domains via requirements to design.

It does so in a novel way: Vol. I of this book is a guide to the study of Vol. II of this book. Vol. II contains, over 14 appendices (Appendices E–R) a fairly large 'support' example of a software development. It is carried out according to the principles and techniques outlined in Vol. I.

For lecturers there are electronic (postscript and pdf) slides covering both volumes. One way of lecturing based on this book is to display lecture slides (i.e., Vol. I) on one screen and lecture support slides (i.e., Vol. II) on an adjacent screen. For readers (i.e., strudents) a CD ROM contains all texts and slides thus enabling several modes of study are made possible. On that CD ROM the text versions of Vols. I–II have cross-references to corresponding slide versions !

Background

I wrote [33, 34, 35] as "The Mother of all Books on Software Engineering" !

Since the 2006 publication of [33, 34, 35] a few clarifications of some domain and requirements engineering principles and techniques have come about — and been published [39, 36, 40, 37, 42, 38].

The book [33, 34, 35], with its 2414 pages, may not exactly be a most enticing way to be introduced to the wonders of how domain engineering precedes requirements engineering. This is despite the possibilities that subsets of each volume can be studied by themselves (first Vol. 1, then Vol. 2), and likewise subsets of Vol. 3 can be studied independent of the previous volumes.

Finally, an essence of [33, 34, 35] is the triptych of Vol. 3 [34].

The present book focuses on that triptych — but in a totally different way.

In fact, this book is "totally" different from previous textbooks and signals a new way of teaching.

The Essentials

I have therefore written this "two volume" book as such a hopefully enticing way into the related engineering of domains and requirements.

So, in two small volumes, one in paper format, the other probably as an enclosed CD ROM, you get the very essence of domain and requirements engineering.

We cover both informal and formal specifications. The formal specifications are in the RAISE specification language RSL. This language will be introduced "along the way" — as it is being used. Every "first time" formula will be explained, and an RSL Primer, Appendix S, summarises the syntactic aspects of the language.

Volume I: A 225 Page Guide to The Triptych Method

Volume I consists of

- Chaps. 1–5 (Pages 1–225) and
- Appendices A–D (Pages 227–314).

The two "volumes" are to be studied in companion: You put both volumes in front of you, perhaps Vol. I in paper form, as a booklet, and Vol. II you may then display on your PC screen. Vol. I makes numerous references to "this or that" section of Vol. II. So you read Vol. I, get referred to, and thus checks with "such and such" a section of Vol. II. In lecture form slides will be available for the entire book. The lecturer will have both volumes displayable on two "parallel" lecture hall overhead screens and can alternate between lecture parts from Vol. I and example (support) parts from Vol. II.

Volume I contains four appendices:

- Appendix A is the Bibliography. It lists some **??** entries.
- Appendix B contains a extensive Indexes.
- Appendix C contains a rather complete (and hence large) Glossary (Pages 259–310). You may wish to study it all by itself ! It explains some 500 terms.
- Appendix D contains a brief overview of two axiom systems, one for time, the other for time-space.

Volume II: A Supporting Software Development

Volume II consists of

- Appendices E–R: A Model Development (317–499);

- Appendix S: An RSL Primer (503–525);
- Appendix T: Solutions to Exercises of Chaps. 1–3 and Appendices F–R.

Volume I will exclusively consist of informal English text. That text explains the Triptych approach to software development. Volume II provides all supporting examples on pages 317 to 499. Hence Vol. I will make numerous references to sections and pages of Vol. II.

On Lecturing over this Book

This book is written for a basically 12 week 3rd year undergraduate or a 1st year graduate course. Students — and readers in general — need some experience in programming.

Knowledge accrued from a combination of passing 3–4 courses in functional programming [97], imperative programming (as in a suitable subset of C, Java or C#), logic programming [142, 112, 11, 12] and parallel programming using, for example CSP [66, 111] is always a winner. Short of that a subset of these "clean programming" courses and either Java or C# is OK. (Familiarity with object-oriented programming is not necessary.) In fact, just studying the delightful [221] might just be enough !

Both Vols. I and II are offered as colourful slides — covering almost all material. Slides are by chapter and appendix, and are organised around the concept of two sets of 35+35 minute lectures per week, that is, a total of 24 lectures of 70 minutes. In addition a weekly three hour tutoring afternoon is intended to go through the model development of Vol. II together with presenting solutions to exercises posed at previous tutoring sessions.

Two kinds of exercises are offered. The first class of exercises are directly related to the topic of the appendix at the end of which they are posed. The second class, instead of focusing on the domain of Vol. II, namely that of transportation, suggests that students work out term reports much in the style of the model development of Vol. II, but for different domains. Any domain could be chosen, but we offer guidance, also in Appendices E–R of Vol. II, to such domains as: the financial service industry and container line industry.

A Possible Course Plan

A course based on this 2-volume book, i.e., the 'formal' text and the extensive example of a model development, has three parts:

- formal lecture sessions,
- tutorial sessions and
- student ("at home") course project work.

Yes, we suggest, strongly, that students pursue a lecturer-guided term project. This course project is, likewise strongly, suggested to be that of a domain — or

possibly a domain and requirements — engineering project. In such a project students are typically collected in M groups of approximately n students each — where n typically is 3–5, with 4 being optimal. Each group focuses on a distinct domain (and possible requirements). Exercise sections of most chapters of Vol. I outline such group projects.

The following lecture plan can be "squeezed" into a 12 week, 2 sessions per week, course period:

"slide 1"
"slide 2"

- **Introduction:**
  Lecture 1 : The Triptych and Informative Documents (I): Week #1
  1 The Triptych Paradigm
  2 Phases, Stages and Steps
  3 Informative Documents (I)
    (a) Project Name and Date
    (b) Project Partners and Addresses Addr.
    (c) Current Situation
    (d) Needs and Ideas
    (e) Scope and Span
    (f) Assumptions and Dependencies
    (g) Implicit/Derivative Goals
    (h) Synopsis

    Lectures: Pages 3–13. The Model Development: Pages 317–326.

  Lecture 2 : Inform.Docs. (II) & Method.: Week #1
  1 Informative Docs. (II)
    (i) Software Development Graphs
    (j) Resource Allocation
    (k) Budget Estimate
    (l) Standards Compliance
    (m) Contracts and Design Briefs
    (n) Logbook
  2 Methodology

  Lectures: Pages 13–24. The Model Development: Pages 13–331.

  Lecture 3: Conceptual Framework (I): Week #2
  1 Modelling and Analysis
  2 Descriptions, Prescriptios, Specifications
  3 Informal and Formal Development
  4 Software

  Lectures: Pages 25–34.

  Lecture 4: Conceptual Framework (II): Week #2
  5 Entities, Functions, Events, Behaviours

    Lectures: Pages 34–42. The Model Development: Pages 343–364.
  6 Domain Modelling versus Operational Research

    Lectures: Pages 42–45.

"slide 3"

- **Domain Engineering:**
  Lecture 5 : Prelude Stages: Week #3

1 The Domain Concept
2 Stages of Domain Engineering
3 Domain Stakeholders
4 Domain Acquisition
5 Domain Analysis and Concept Formation
6 Business Processes
7 Terminology

Lectures: Pages 51–64. The Model Development: Pages 332–339.

Lectures 6–8 : Domain Modelling:

1 **Lecture 6**: **Week #4**
   (a) Intrinsics
   (b) Support Technologies

   Lectures: Pages 64–73. The Model Development: Pages 343–384.
2 **Lecture 7**: **Week #5**
   (a) Management and Organisation
   (b) Rules and Regulations

   Lectures: Pages 73–84. The Model Development: Pages 387–411.
3 **Lecture 8**: **Week #5**
   (a) Scripts
   (b) Human Behaviour

   Lectures: Pages 84–102. The Model Development: Pages 415–454.

Lecture 9 : Postlude Stages: **Week #6**

1 Verification
2 Validation
3 Theory Formation
4 Domain Engineering Process Graph
5 Domain Engineering Documents

Lectures: Pages 102–106. The Model Development: Pages 457–457. "slide 4"

- **Requirements Engineering:**

Lecture 10 : Prelude Stages: The Requirements Engineering Stages **Week #6**

Lectures: Pages 109–127. The Model Development: Pages 461–469.

Lectures 11-13 : Requirements Modelling:

Lecture 11 : Domain Reqs. Modelling: **Week #7**

1 Projection,
2 Instantiation,
3 Determination
4 Extension
5 Fitting
6 Composition

Lectures: Pages 127–132. The Model Development: Pages 471–481.

Lecture 12 : Interface Reqs. Modelling: **Week #8**

1 Shared Phenomena
2 Shared Entity Requirements
3 Shared Function Requirements
4 Shared Event Requirements

5 Shared Behaviour Requirements
Lectures: Pages 133–134. The Model Development: Pages 483–485.

"slide 5"

- **Continued: Main Stage: Requirements Modelling:** **Week #9**
  Lecture 13 : **Machine Reqs.:**
  1 Performance
  2 Dependability
  3 Maintainability
  4 Platform
  5 Documentability
  6 Etcetera
  Lectures: Pages 135–163. The Model Development: Pages 487–491.
  Lecture 14 : **Postlude Stages:** **Week #9**
  1 Verific., Valid.
  2 Feasibility, Satisfiability
  3 Requirements Engineering Process Graph
  4 Requirements Engineering Documents
  Lectures: Pages 163–166. The Model Development: Pages 493–493.

"slide 6"

- **Software Design:**
  Lecture 15 : **Architectural Design** **Week #10**
  Lectures: Pages 167–189. The Model Development: Pages 497–498.
  Lecture 16 : **Component Design &c.** **Week #11**
  1 Component Design
  2 Software Design Process Graph
  3 Software Design Documents
  Lectures: Pages 189–222. The Model Development: Pages 498–499.
- **Summary:**
  Lecture 17 : **Review of Phases, Stages and Steps:** **Week #12**
  1 Domains, Requirements, Software Design
  2 Process Graphs
  3 Documents
  4 Process Assessment and Improvement
  Lectures: Pages 225–225. The Model Development: Pages 225–225.

"slide 7"

An Explanation

- We assume a 12 week course period, that is, a total of 24 courses sessions.
- Each session is two times 35–45 minutes.
- By a 'formal' session we mean
  - ⋆ a possibly tiered auditorium session in which the lecturer
  - ⋆ lectures over Vol. 1 material
  - ⋆ while showing some Vol. 2 examples

  - ⋆ on two overhead projectors —
    - · one for Vol. 1 slides,
    - · the other, occasionally "blinded", for Vol. 2 slides.
- By a 'tutoring' session we mean a
  - ⋆ a, usually flat classroom, session in in which the lecturer
  - ⋆ only shows Vol. 2 slides
  - ⋆ while walking around the room, discussing the examples
  - ⋆ and their work on the course project with students.
- In weeks 1, 2, 4, 5 and 9 there are two formal lectures per week.
  We are taking into account that student project work is not yet generating sufficient classroom questions.
- All other weeks have one formal session and one tutoring per week.

Acknowledgements

Can't think of anyone at the moment (December 17, 2008) !

Contents

VOLUME I: THE TRIPTYCH METHOD

Part III A Review of The Triptych Approach to SE

Part IV Administrative Appendices

VOLUME II: A MODEL DEVELOPMENT

Part VIII RAISE

Part I

Opening

A brief introduction, Chap. 1, sets the stage for Part II, Chaps. 2–4. The introduction outlines what they cover and what they do not cover.

1

Introduction

"SLIDE 9"

In this chapter we shall overview the 'triptych' approach to software development. The paradigm, first proper section just below, motivates the triplet of 'domain', 'requirements' and software 'design' 'phases' covered briefly in Sect. 1.3. These phases can be pursued in a series of (usually sequentially ordered) 'stages' and the stages likewise in likewise 'steps'. Work on many steps and some stages can occur in parallel. The stage and step concepts are introduced in Sect. 1.4 and covered in detail in Chaps. 2–4. The software engineering of these phases, their stages and steps are focused on constructing 'documents' — and the nature of these is covered in Sects. 1.5–1.8. Section 1.6 is the first major study section of this chapter. The core part of phase documents are either 'descriptive' (i.e., 'indicative', as it is), 'prescriptive' (i.e., 'putative' in the form of properties of what ones wants) or specifies a software design (i.e., are 'imperative'). Sect. 1.9 briefly elaborates on these terms. The "slide 10" term 'software' is given a proper definition — one that most readers should find surprising — in Sect. 1.10. Section 1.11 covers the ideas behind pursuing software development both using informal and formal techniques. And Sect. 1.12 — another major study section of chapter — finally introduces the notions of entities, functions, events and behaviours.

1.1 What Is a Domain ?

"SLIDE 11"

1.1.1 An Attempt at a Definition

Characterisation 1 (Domain) By a *domain* we shall understand a universe of discourse, small or large, a structure of entities, that is, of "things", individuals, particulars some of which are designated as state components; of functions, say over entities, which when applied become possibly state-changing actions of the domain; of events, possibly involving entities, occurring in time and expressible as predicates over single or pairs of (before/after) states; and of behaviours, sets of possibly interrelated sequences of actions and events. ∎

1.1.2 Examples of Domains "slide 12"

We give some examples of domains. (i) A country's railways form a domain of the rail net with its rails, switches, signals, etc.; of the trains travelling on the net, forming the train traffic; of the potential and actual passengers, inquiring about train travels, booking tickets, actually travelling, etc.; of the railway staff: management, schedulers, train drivers, cabin tower staff, etc.; and so forth.

"slide 13"

(ii) Banks, insurance companies, stock brokers, traders and stock exchanges, the credit card companies, etc., form the financial services industry domain.

(iii) consumers, retailers, wholesalers, producers and the supply chain form "the market" domain.

There are many domains and the above have only exemplified "human made" domains, not, for example, those of the natural sciences We shall have more to say about this later. Essentially it is for domains like the 'human made' domains that this book will show you how to professionally develop the right software and where that software is right !

1.2 The Triptych Paradigm "slide 14"

> *Before software can be designed one must understand its requirements. Before requirements can be expressed one must understand the application domain.*

We assume that the reader understands the term 'software', but we shall, in Sect. 1.10, explain our definition of this term. By requirements we understand a document which prescribes the properties that are expected from the software (to be designed). By application domain we understand the business area of human activity and/or the technology area for which the software is to be applied. We shall, in the rest of this book, omit the prefix 'application' and just use the term 'domain'.

1.3 The Triptych Phases of Software Development "slide 15"

1.3.1 The Three Phases

As a consequence of the "dogma" we view software development as ideally progressing in three *phases*: In the first phase, 'Domain Engineering', a model is built of the application domain. In the second phase, 'Requirements Engineering', a model is built of what the software should do (but not how it should that). In the third phase, 'Software Design', the code that is subject to executions on computers is designed.

1.3.2 Attempts at Definitions

"SLIDE 16"

Characterisation 2 (Domain Engineering) By *domain engineering* we shall understand the processes of constructing a domain model, that is, a model, a description, of the chosen domain, as it is, "out there", in some reality, with no reference to requirements, let alone software. ∎

Characterisation 3 (Requirements Engineering) By *requirements engineering* we shall understand the processes of constructing a requirements model, that is, a model, a prescription, of the chosen requirements, as we would like them to be. ∎

"slide 17"

Characterisation 4 (Software Design) By *software design* we shall understand the processes of constructing software, from high level, abstract (architectural) designs, via intermediate abstraction level component and module designs, to concrete level, "executable" code. ∎

Characterisation 5 (Model) By a *model* we shall understand a mathematical structure whose properties are those described, prescribed or design specified by a domain description, a requirements prescription, respectively a software design specification. ∎

1.3.3 Comments on The Three Phases

"SLIDE 18"

The three phases are linked: the *requirements prescription* is "derived" from the *domain description*, and the *software design* is derived from the requirements prescription in such a way that we obtain a maximum trust in the software: that it meets customer expectations: that is, it is the right software, and that it is correct with respect to requirements: that is, the software is right.

"slide 19"

Characterisation 6 (Phase of Software Development) By a *phase of development* we shall understand a set of development stages which together accomplish one of the three major development objectives: a(n analysed, validated, verified) domain model, a(n analysed, validated, verified) requirements model, or a (verified) software design. These three "tasks": a domain model, a requirements model, and a software design will be defined below. ∎

"slide 20"

Characterisation 7 (Software Development) Collectively the three phases are included when we say 'software development'. ∎

Domain engineering is covered as follows: Chapter 2 outlines all the stages and steps of domain engineering. It does not bring examples. Instead the book provides for one large example, the 'Model Development' of most of Vol. II. Hence Appendices F–K provides in "excruciating" detail examples of

all the relevant aspects of domain engineering. These are then being referred to in Chap. 2.

Requirements engineering is covered as follows: Chapter 3 outlines all the stages and steps of requirements engineering. Like Chap. 2 Chap. 3 does not bring examples. Instead Appendices N–P provides in "excruciating" detail examples of all the relevant aspects of requirements engineering. These are then being referred to in Chap. 3.

Software design is not covered "in earnest" in this book. Chapter 4 overviews how one refines the requirements prescription, in stages and steps of development, into executable code. Appendix R, as a consequence, only offers some rudimentary examples.

1.4 Stages and Steps of Software Development "slide 21"

We make distinctions between phases of development (i.e., the domain engineering, the requirements engineering and the software design phases), stages of development — within a phase, and steps of development — within a stage.

1.4.1 Stages of Development "slide 22"

Characterisation 8 (Stage of Software Development) By a *stage of development* we mean a major set of logically strongly related development steps which together solves a clearly defined development task. ∎

We shall later define the stages of the major phases, and we shall then be rather loose as to what constitutes a development step. That is, Chaps. 2–3 shall define the specific stages relevant to those phases of development.

1.4.2 Steps of Development "slide 23"

Characterisation 9 (Step of Software Development) By a *step of development* we mean iterations of development within a stage such that the purpose of the iteration is to improve the precision or make the document resulting from the step reflect a more concrete description, prescription or specification. ∎

1.5 Development Documents "slide 24"

All we do, really, as software developers, can be seen as a long sequence of documenting, i.e., producing, writing, documents alternating with thinking and reasoning about and presenting and discussing these documents to and with other people: customers, clients and colleagues. Among the last documents to be developed in this series are those of the executable code. "slide 25"

In this section we shall take a look at the kind of documents that should result from the various phases, stages and steps of development, and for whose writing, i.e., as "input", aside from other documents, we do all the thinking, reasoning, and discussing.

For any of the three phases of development, one can distinguish three classes of documents:

1.6 Informative Documents

"SLIDE 26"

An informative document 'informs'. An informative document is expressed in some national language.[1] Informative documents serve as a link between developers. clients and possible external funding agencies:

- *"What is the project name ?"* Item 1[2]
- *"When is the project carried out ?"* Item 1
- *"Who are the project partners ?"* Item 2
- *"Where is the project being done ?"* Item 2
- *"Why is the project being pursued ?"* Items 3(a)–3(b)
- *"What is the project all about ?"* Items 3(b)–3(g)
- *"How is the project being pursued ?"* Items 4–6

"slide 27"

And many other such practicalities. Legal contracts can be seen as part of the informative documents. We shall list the various kinds of informative documents that are typical for domain and for requirements engineering.

1.6.0 An Enumeration of Informative Documents

Instead of broadly informing about the aims and objectives of a development project we suggest a far more refined repertoire of information "tid-bits". A listing of the sixteen names of these "tid-bits" hints at these:

"slide 28"

[1] The fact that informative documents are informal displays a mere coincidence of two times 'inform'.

[2] The item numbers refer to the enumerated listing given on Page 7.

(e) Assumptions and Dependencies Sect. 1.6.7
(f) Implicit/Derivative Goals Sect. 1.6.8
(g) Synopsis Sect. 1.6.9
4 [Project Plan]
(a) Software Development Graph Sect. 1.6.10
(b) Resource Allocation Sect. 1.6.11
(c) Budget Estimate Sect. 1.6.12
(d) Standards Compliance Sect. 1.6.13
5 Contracts and Design Briefs Sect. 1.6.14
6 Logbook Sect. 1.6.15

For examples of 'information' modelling and resulting documents we refer to Appendix E, Sect. E.1 and to Appendix M, Sect. M.1.

"slide 29"

We shall now explain each of these kinds of informative documents.

1.6.1 Project Name and Dates

"SLIDE 30"

The first information are those of

- Project Name: the name of the endeavour;
- Project Dates: the dates of the project.

For examples of 'project name and dates' modelling and resulting documents we refer to Appendix E, Sect. E.1.1, Page 318 and to Appendix M, Sect. M.1.1, Page 461.

1.6.2 Project Partners and Places

"SLIDE 31"

The second information is that of

- Project Partners: who carries out the project.
  Full partner (collaborator) details are (eventually) to be given:
  - ⋆ Client(s): full names, addresses, and possibly names of contact persons, etc., of the people and/or companies and/or institutions who and which have 'ordered' the project and who and which shall receive its resulting documents.
  - ⋆ Developer(s): full names, addresses, and possibly names of contact persons, etc., of the people and/or companies and/or institutions who and which are primarily developing the deliverables of the project and who and which shall receive its main funding.

"slide 32"

  - ⋆ Project Consultant(s): full names, addresses, and possibly names of possible consultants, i.e., companies and/or individuals outside "the circle" of clients and developers.
  - ⋆ Project Funding Agencies: full names, addresses, possibly names of contact persons, etc., of the people and/or agencies who and which are possibly [co-]funding the project.

  - ⋆ Project Audience: full names, addresses, and possibly names of contact persons, etc., of the people and/or agencies who and which are possibly (also) interested in the project.

"slide 33"

- Project Places: where is the project carried out ? Full addresses: visiting and postal mailing addresses and electronic addresses.

For examples of 'project partners and places' modelling and resulting documents we refer to Appendix E, Sect. E.1.2, Page 318 and to Appendix M, Sect. M.1.2, Page 462.

1.6.3 Current Situation

"slide 34"

Usually a domain engineering project is started for some reason. Either the developer or the client, or both, have only scant knowledge of the domain, or, when they have it is not written down but is "inside" the heads of some or most of their (i.e., developer or client) staff. Similarly a requirements "slide 35" engineering project is started for some reason. A common reason is that of the current situation on the client side. Either no IT is used but there is a need for some IT, or current IT is outdated, or new demands are made by owners, management or employees in general at the client, demands that "translate" into altered or new IT; or customers of the client may have similar expectations — of better e-service etc., from the client, i.e., their provider. "slide 36" For a software design project ....

The 'Current Situation' document must outline this in succinct terms: say half to a full page.

For examples of 'current situation' modelling and resulting documents we refer to Appendix E, Sect. E.1.3, Page 319 and to Appendix M, Sect. M.1.3, Page 462.

1.6.4 Needs and Ideas

"slide 37"

Needs

Usually the current situation is paraphrased, i.e., accentuated, by expressions of specific 'needs' for a domain description, or for a requirements prescription, or for a completed software design, i.e., for software.

The need for a domain description could either be that it should form the basis for an orderly process of requirements development, or the basis for teaching and learning courses, say for new staff of the enterprise (of the domain), or both. "slide 38"

The need for a requirements prescription could either be that it should form the basis for an orderly process of requirements development, or the basis for a tender, i.e., an offer to develop some software, or both. "slide 39"

Usually can express needs while at the same time indicate how one might foresee an expressed need being possibly fulfilled, i.e., achieved.

A need for a software design may be that it must be based on an existing requirements prescription.

A need for a requirements prescription may be that it must be based on an existing domain description.

A need for a domain description may be that it must be just informal, another need may be that it be both informal and formal.

Ideas[3]

One thing are the 'needs'. Another thing are the 'ideas'. If there are needs but no ideas, or if there is no need but ideas, then "forget it": no reason to embark on a development !

"slide 41" By *ideas* we mean that there are some substantial concepts that, when properly deployed, can lead to a believable development, whether of a domain description, of a requirements prescription, or of a software design.

By *domain ideas* we mean such concepts "upon" or "around" which one can build, one can model, a domain description. Examples will be given in Sect. 2.3 on page 55

By *requirements ideas* we mean such concepts "upon" or "around" which one can build, one can model, a requirements prescription. Examples will be given in Sect. 3.3 on page 117

By *software design ideas* we mean such concepts "upon" or "around" which one can build, one can model, a software design. Examples will be given in Sect. 4.3 on page 170

• • •

For examples of 'needs and ideas' modelling and resulting documents we refer to Appendix E, Sect. E.1.4, Page 320 and to Appendix M, Sect. M.1.4, Page 462.

1.6.5 Concepts and Facilities

"slide 42"

The pragmatics of the 'concepts and facilities' section is to — ever so briefly — inform all parties to the contract of which are the most important ideas of the subject domain of the contract. A facility is a physical phenomenon (here embodied, for example, in the form of software tools) while a concept is a mental construction (covering, usually some physical phenomena or concepts of these).

"slide 43"

In the context of informing only about a domain description development project the concepts and facilities are intended, in the document section of that name, to be the most pertinent concepts and facilities on which the domain description should focus.

"slide 44"

[3] "slide 40"

In the context of informing only about a requirements prescription development project the concepts and facilities are intended, in the document section of that name, to be the most pertinent concepts and facilities of the requirements prescription: which are the novel ideas which the requirements should be based. For examples of 'concepts and facilities' modelling and resulting documents we refer to Appendix E, Sect. E.1.5, Page 321 and to Appendix M, Sect. M.1.5, Page 463.

1.6.6 Scope and Span

"SLIDE 45"

Characterisation 10 (Scope) By *scope* — in the context of informative software development documentation — we shall understand an outline of the broader setting of the problem, i.e., the universe of discourse at hand. . ■

Characterisation 11 (Span) By a *span* — in the context of informative software development documentation — we shall understand an outline of the more specific area and the nature of the problem that need be tackled. ■

"slide 46"

Let us examine a few generic cases of scope/span determination.

(i) "Pure" domain engineering scope and span: By ' "pure" domain engineering' we mean a project aimed at just producing a domain model. In such a case the scope should typically be chosen as wide as possible, while the span is a proper, but not too "small" subset of the scope. "slide 47"

(ii) Domain and requirements engineering scope and span: By 'domain and requirements engineering' we mean a project first aimed at producing a domain model and then, from it, "derive" a requirements model. In such a case the scope should typically be chosen to be comfortably wider than the scope of the requirements part of the project. "slide 48"

(iii) Requirements engineering and software design scope and span: By 'requirements engineering and software design' we mean a project first aimed at producing a requirements model and then, from it, "derive" a software design. In such a case the scope and span part of the requirements part of the project should be equal. Software design projects have their scope and span being set by the requirements part of the project.

For examples of 'scope and span' modelling and resulting documents we refer to Appendix E, Sect. E.1.6, Page 322 and to Appendix M, Sect. M.1.6, Page 464.

1.6.7 Assumptions and Dependencies

"SLIDE 49"

There are two kinds of assumptions and dependencies. One kind has to do with sources of knowledge. For domain development there needs to be the sources from which the domain engineer can learn about and develop the domain description. We assume and depend on that. For requirements development there needs to be a domain description as well as people from whom

the requirements engineer can elicit the requirements and thus develop the requirements prescription. We assume and depend on that. And for software design there needs to be a requirements prescription. We assume and depend on that. The other kind has to do with delineation of the domain.

"slide 50"

Usually a domain description (one upon which we base our (domain) requirements) leaves out what we might call the "fringes" of the domain, i.e., the environment of that domain. To also describe those parts might simply "be too much"! That environment is simply judged too large, too unwieldy, to describe.

"slide 51"

Yet, sooner or later, that environment will show up in the requirements prescription, if it is not already in the domain description. The requirements prescription eventually, thus, comes to depend — maybe not exactly crucially, but anyway — on events originating in the environment, or the ability of the computing system to dispose of some output to that environment.

In the 'assumptions and dependencies' project document the project responsible must clearly express these assumptions and dependencies.

For examples of 'assumptions and dependencies' modelling and resulting documents we refer to Appendix E, Sect. E.1.7, Page 323 and to Appendix M, Sect. M.1.7, Page 465.

1.6.8 Implicit/Derivative Goals

"slide 52"

Usually computing systems provide, or offer, a large number of entities, functionalities, events and behaviours, and it is those requirements we prescribe. But those entities, functionalities, events and behaviours really do not themselves reveal why they are or were prescribed. Usually their prescription serves "ulterior" goals which cannot be quantified in a way that indicates what the prescribed computing system should offer.

"slide 53"

Typical meta-goals are such as: (i) *"Deployment of the computing system should result in greater profits for the company."* (ii) *"Deployment of the computing system should result in the company attaining a larger market share for its products."* (iii) *"Deployment of the computing system should result in fewer worker accidents."* (iv) *"Deployment of the computing system should result in more satisfied customers (and staff)."*

"slide 54"

Other kinds of meta-goals are: (v) "The existence of a domain description will have led or should lead to better understanding of the domain, hence to improved performance of domain staff trained in the domain based on such domain descriptions." (vi) "The existence of a requirements prescription will have led or should lead to more appropriately targeted software."

"slide 55"

In the 'implicit/derivative goals' project document the project responsible must clearly express these implicit/derivative goals.

For examples of 'implicit/derivative goals' modelling and resulting documents we refer to Appendix E, Sect. E.1.8, Page 324 and to Appendix M, Sect. M.1.8, Page 465.

1.6.9 Synopsis

"SLIDE 56"

The four sub-groups of informative document parts: current situation, needs and ideas, scope and span, and concepts and facilities, form an introductory "whole" that now need be "solidified". They need to be brought together in a more coherent fashion — in what we shall call the synopsis document "slide 57"

Characterisation 12 (Synopsis) By a *synopsis*[4] — in the context of informative software development documentation — we shall understand the same as a resumé, a summary, that is, a comprehensive view, that is, an extract of a combination of current situation, needs and ideas, concepts, and scope and span documentation informing about a universe of discourse for which some development work is desired, for example: (i) the construction of a domain "slide 58" description, (ii) or the construction of a requirements prescription based on an existing domain description, or both; (iii) or the construction of a software design based on existing requirements prescription; (iv) or both (requirements and software design), (v) or all (domain, requirements and software design); (vi) or the first two (domain and requirements). ∎

For examples of 'synopsis' modelling and resulting documents we refer to Appendix E, Sect. E.1.9, Page 325 and to Appendix M, Sect. M.1.10, Page 465. "slide 59" "slide 60"

1.6.10 Software Development Graphs

"SLIDE 61"

Development projects need be managed. This is true also for single person projects. Management of domain engineering projects must take into account that these are normally research projects: little is objectively known about the domain before it is properly described; hence one must be prepared for "unforeseen" resource usage. Software development graphs are a means of "slide 62" capturing, either beforehand, during, or after the project how that project is to be done, is being done, or was done, respectively !

Graphs[5]

Characterisation 13 (Software Development Graph) By a *software development graph* we shall *syntactically* understand a labelled graph whose distinctly labelled *nodes* (*vertexes*) designate development activities (phases, stages or steps), and whose distinctly labelled, directed *edges* (*arcs*) designate *precedence relations* between (node designated) activities. "slide 64"

Semantically a software development graph designate a set of project behaviour designators. A *project behaviour designator* is a sequence of *phase*, *stage* or *step state designators* and *state transition designators*.

[4] Synopsis: Greek, comprehensive view, from *synopsis:* to be going to see together.
[5] "SLIDE 63"

A *phase, stage* or *step state designator* is a node label such that the node is that of a phase part, or a stage part, or a step part of a software development graph.

A *state transition designator* is an edge label such that the edge is that of an edge of a development graph. ∎

A Conceptual Software Development Graph[6]

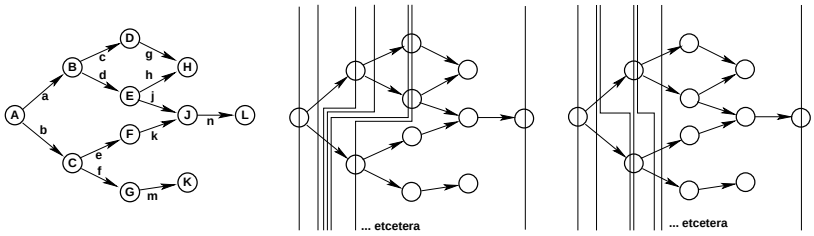


Fig. 1.1. A software development graph (left)
and two (incomplete) project behaviour designators (center and right)

The center graph of Fig. 1.1 portrays the following incompletely listed project behaviour designator:

<{A},{a,b},{B,b},{c,d,b},{D,E,b},{D,E,C},...,{L}>

"slide 66"

The "abstracted" software development graph of Fig. 1.1 denotes a very large number of project behaviours, that is, a very large number of project behaviour designators, and, for each of these, depending on the states of phase, stage or step, as represented, for example, by the states of the documents related to each of the nodes, a very large number of (dynamic) behaviours.

Who Sets Up the Graphs ?[7]

Management is responsible for setting up an appropriate software development graph (for each project). A software development graph shows how management intends to pursue the project: which phases, stages and steps to conduct, that is, to which depth of adherence to the triptych principles management wishes to achieve its aims.

Chapters 2 and 3 will illustrate "abstracted", i.e., generic, software development graph reflecting phases, stages and steps. Sections 2.14 and 3.14 summarize these (Figs. 2.3 on page 103 and 3.17 on page 164).

[6] "SLIDE 65"

[7] "SLIDE 67"

How Do Software Development Graphs Come About ?[8]

For a given, specific project, its software development graph comes about in any number of ways. (i) Either the project is a "repeat" project, that is, is developing a kind of software which has been developed before. In that case one simply uses the software development graph used in those earlier projects. But since there probably are some small, or perhaps not even that small, difference between the current project and the previous ones, the currently chosen software development graph may be modified. Thus every software development graph will be recorded for possible re-use in future. It becomes part of the "corporate assets" of the software house. "slide 69"

(ii) Or the project is a "research" project, that is, is developing a new kind of software which has not been developed before. In that case one starts with the process diagram most appropriate for the project.

If it is a domain engineering project then one starts with the domain engineering process graph of Fig. 2.3 on page 103 as the software development graph; modifies this graph to suit the specific domain at hand, all the while recalling that development of domain descriptions are really research rather than engineering tasks, hence accepting that the software development graph need be modified along the way: clear resource estimates of time and effort cannot be assured. "slide 70"

If it is a requirements engineering projectthen one starts with the domain engineering process graph of Fig. 3.17 on page 164 as the software development graph; and modifies this graph to suit the specific requirements at hand. One must always be prepared to modify the software development graph along the way.

For examples of 'concrete software development graph' modelling and resulting documents we refer to Appendix E, Sect. 1.6.10, Page 13 and to Appendix M, Sect. M.1.11, Page 465.

1.6.11 Resource Allocation "SLIDE 71"

Characterisation 14 (Software Development Graph Attribute) An *attributed software development graph* is a software development graph whose nodes and edges have been assigned development attributes. ∎ "slide 72"

Usually *node development attributes* include whether the node is a domain, a requirements or a software design development node; whether the node is a phase, stage, or step node; of what specific kind the node — when not just a phase node — is: any one of the stages of the three triptych phases[9];

[8] "SLIDE 68"

[9] The phases of domain and requirements modelling and analysis will first be "revealed" in Chaps. 2 and 3 — the only stage of the information document development is just that stage.

any one of the 16 kinds of information document development steps enumerated on Page 7; or any one of the many stages or steps of the domain and requirements modelling and analysis to be "revealed" in Chaps. 2 and 3.

"slide 73"

Given an attributed software development graph and given experience from projects "similar" to the one described by the graph one can now estimate resources to be allocated to each task, that is, to the carrying out the actions implied by each of its nodes. These resource estimates are of the following kinds: number and qualifications of project staff; when, i.e., during which periods each individual, but not yet named staff, is to be available for the action denoted by the box being attributed; tools (office space, equipment (incl. IT equipment), software — by allocated staff members — to be available for that action; 'begin' and 'end time'; etcetera.

"slide 74"

These estimates can be affixed to the nodes (boxes); thus augmenting its set of attributes.

For examples of 'resource allocation' modelling and resulting documents we refer to Appendix E, Sect. E.1.11, Page 327 and to Appendix M, Sect. M.1.12, Page 465.

1.6.12 Budget (and Other) Estimates

"SLIDE 75"

From the augmented (i.e., extended attributed) software development graph one can now derive a number of estimates:

- (i) a budget estimate, per phase and stage, and thus for the entire (software development graph [SDG] designated) project;
- (ii) a time estimate, per phase and stage, and thus for the entire (SDG designated) project;
- (iii) a staff estimate, per phase and stage, and thus for the entire (SDG designated) project (here it must be analysed which activities can occur in parallel) and usually in the form of a histogram;
- (iv) an equipment estimate, per phase and stage, and thus for the entire (SDG designated) project;
- etcetera.

For examples of 'budget estimate' modelling and resulting documents we refer to Appendix E, Sect. E.1.12, Page 328 and to Appendix M, Sect. M.1.13, Page 466.

1.6.13 Standards Compliance

"SLIDE 76"

A distinction is made between development standards and documentation standards.

Development Standards

Usually development occurs in the context of following some development standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [115]) has established a number of standards for the development of a various kinds of software. Other national and international organisations, including the International Organization for Standardization (ISO [117]) and the International Telecommunication Union (ITU [121]), have established similar standards.

Documentation Standards[10]

Usually documentation occurs in the context of following some documentation standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [115]) has established a number of standards also for the documentation of a various kinds of software. Other national and international organisations, including the International Organization for Standardization (ISO [117]) and the International Telecommunications Union (ITU [121]), have also established similar standards.

Standards Versus Recommendations[11]

Some standards are binding, some are recommendations. Reference to specific standards and recommendations can be written into project contracts with the meaning that the project must comply with these standards and recommendations. Some standards mandate or recommend the use — and hence the documentation style — of certain development practices. Other standards mandate or recommend the use of specific spelling forms, mnemonics, abbreviations, etc.

Specific Standards[12]

There are very many standards for software development and for its documentation. Some standards come and go. Others are quite stable. A study of more specialised standards reveals the following acronyms: MIL-STD-498, DOD-STD-2167A, RTCA/DO-178B, JSP188 and DEF STAN 05-91. The reader is invited to search for these on the Internet. It therefore makes little sense for us to list other than a few clusters of seemingly more stable and trustworthy standards. "slide 80"

- *International Organization for Standardization (ISO):* `http://www.iso.ch/`

[10] "SLIDE 77"
[11] "SLIDE 78"
[12] "SLIDE 79"

  - ⋆ ISO 9001: Quality Systems Model for quality assurance in design, development, production, installation and servicing
  - ⋆ ISO 9000-3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software
  - ⋆ ISO 12207: Software Life Cycle Processes `http://www.12207.com/`

"slide 81"

- IEEE Standards: `http://standards.ieee.org/`
  - ⋆ IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology
    This standard contains definitions for more than 1000 terms, establishing the basic vocabulary of software engineering.
  - ⋆ IEEE Std 1233-1996, Guide for Developing System Requirements Specifications
    This standard provides guidance for the development of a set of requirements that, when realized, will satisfy an expressed need.

"slide 82"

  - ⋆ IEEE Std 1058.101987, Standard for Software Project Management Plans
    This standard specifies the format and contents of software project management plans.
  - ⋆ IEEE Std 1074.1-1995, Guide for Developing Software Life Cycle Processes
    This guide provides approaches to the implementation of IEEE Std 1074. (This standard defines the set of activities that constitute the mandatory processes for the development and maintenance of software.)
  - ⋆ IEEE Std 730.1-1995, Guide for Software Quality Assurance Plans
    The purpose of this guide is to identify approaches to good Software Quality Assurance practices in support of IEEE Std 730. (The standard establishes a required format and a set of minimum contents for Software Quality Assurance Plans. The description of each of the required elements is sparse and thus provides a template for the development of further standards, each expanding on a specific section of this document.)

"slide 83"

  - ⋆ IEEE Std 1008-1987 (reaffirmed 1993), Standard for Software Unit Testing
    The standard describes a testing process composed of a hierarchy of phases, activities, and tasks. Further, it defines a minimum set of tasks for each activity.
  - ⋆ IEEE Std 1063-1987 (reaffirmed 1993), Standard for Software User Documentation
    This standard provides minimum requirements for the structure and information content of user documentation.
  - ⋆ IEEE Std 1219-1992, Standard for Software Maintenance
    This standard defines a software maintenance process.

"slide 84"

- Software Engineering Institute (SEI): http://www.sei.cmu.edu
  - ⋆ Software Process Improvement Models and Standards, including SEI's various Capability Maturity Models
- *UK Ministry of Defence Standards* http://www.dstan.mod.uk/

- ⋆ 00-55: Requirements for Safety Related Software in Defence Equipment
  `http://www.dstan.mod.uk/data/00/055/02000200.pdf`
- ⋆ 00-56: Safety Management Requirements for Defence Systems
  `http://www.dstan.mod.uk/data/00/056/01000300.pdf`

So, please, use the Internet for latest on standards relevant to your project.

For examples of 'standards compliance' modelling and resulting documents we refer to Appendix E, Sect. E.1.13, Page 329 and to Appendix M, Sect. M.1.14, Page 466.

1.6.14 Contracts and Design Briefs

"SLIDE 85"

Contracts

The current situation, needs and ideas, concepts and facilities, scope and span and synopsis document parts set the stage for, and are a necessary background for contractual documents. Usually one contract document is sufficient for small projects. And usually several related contract documents are needed for larger projects.

Characterisation 15 (Contract) By a *contract* — in the context of informative software development documentation — we shall understand a separate, clearly identifiable document (i) which is legally binding in a court of law, (ii) which identifies parties to the contract, (iii) which describes what is being contracted for, possibly mutual deliveries, by dates, by contents, by quality, etc., (iv) which details the specific development principles, techniques, tools and standards to be used and followed, (v) which defines price and payment conditions for the deliverables, (vi) and which outlines what is going to happen if delivery of any one deliverable is not made on time, or does not have the desired contents, or does not have the desired quality, etc. ∎

"slide 86"

Items (iii–iv) constitute the main part of a *design brief*. (See below.)

"slide 87"

For national and for international contracts predefined forms which make more precise what the contracts must contain are usually available. We will not bring in an example. Such an example would have to reflect the almost 'formal' status of 'legal binding', and would thus have to be extensive and very carefully worded, hence rather long. Instead we refer to national and international contract forms.

The software development field is undergoing dramatic improvements. Clients are entitled to have legally guaranteed quality standards (incl. correctness verification). Hence contracts will have to refer to(i)the broader domain and give specific references to named domain stakeholders, if the development of a domain description is (to be) contracted; or (ii)existing domain descriptions and give specific references to named stakeholders, if the development of a requirements prescription is (to be) contracted; or (iii)existing requirements prescriptions and give specific references to named stakeholders, if the development of software is (to be) contracted.

"slide 88"

Therefore contracts should name "the methods" by means of which the deliveries will be developed — as we have indicated in item (iv) of the characterisation.

Contract Details[13]

"slide 90"
1 **Overview:** Contracts between an organization and a software vendor should clearly describe the rights and responsibilities of the parties to the contract. The contracts should be in writing with sufficient detail to provide assurances for performance, source code accessibility, software and data security, and other important issues. Before management signs the contracts, it should submit them for legal counsel review.
Organizations may encounter situations where software vendors cannot or will not agree to the terms an organization requests. Under these circumstances, organizations should determine if they are willing to accept or able to mitigate the risks of acquiring the software without the requested terms. If not, consideration of alternative vendors and software may be appropriate.

"slide 91"
2 **General Issues of Licensing:**
Software is usually licensed, not purchased; and under licensing agreements, organizations obtain no ownership rights, even if the organization paid to have the software developed. In general, for domain descriptions and requirements prescriptions, a license should clearly define permitted users and sites.

"slide 92"
3 **Copyright:**
Proprietary as well as open-source software are protected by copyright laws. If need be then clients and vendors must make sure that also their domain descriptions and requirements prescriptions are protected by being proprietary.

"slide 93"
4 **Domain, Requirements and Software Development Specifications:**
Contracts for the development of custom domain descriptions, requirements prescriptions, and software design must be very specific about the scope and span of domain descriptions and requirements prescriptions, that requirements prescriptions build on accepted domain descriptions, that requirements prescriptions are feasible and satisfiable, and that software designs build on accepted requirements prescriptions.

"slide 94"

"slide 95"
5 **Performance Standards:**
This issue relates to requirements and software. When the requirements prescriptions are claimed feasible and satisfiable, then there must be software that satisfies the requirements. These requirements also include performance requirements, part of the machine requirements to be covered in Chap. 3.

"slide 96"
6 **Documentation, Modification, Updates and Conversion:**

[13] "SLIDE 89"

A licensing or development agreement should require vendors to deliver appropriate documentation. This should include all kinds of documentation — such as defined later. A license or separate maintenance agreement should address the availability and cost of document updates and modifications.

7 **Bankruptcy:** "slide 97"
In addition to escrow agreements, organizations should consider the need for other clauses in licensing agreements to protect against the risk of a vendor bankruptcy. For mission-critical software, organizations should consult with their legal counsel on how best to deal with the Bankruptcy laws, which typically gives a bankrupt vendor discretion to determine which of its executory contracts it will continue to perform and which it will reject. Proper structuring of the contract can help an organization protect its interests if a vendor becomes insolvent.

8 **Regulatory Requirements:** "slide 98"
Domain descriptions, requirements prescriptions and software designs must individually often have to comply with national (state and federal), regional (NAFTA, EU, etc.), and/or international (ICAO, IMO, etc.) regulatory agency requirements. These compliance requirements must be clearly stated in the contract.

9 **Payments:** "slide 99"
Software development contracts normally call for partial payments at specified milestones, with final payment due after completion of acceptance tests. Organizations should structure payment schedules so developers have incentives to complete the project quickly and properly. Properly defined milestones can break development projects into deliverable segments so an organization can monitor the developer's progress and identify potential problems.
Contracts should detail all features and functions the delivered software will provide. If a vendor fails to meet any of its express requirements, organizations should retain the right to reject the tendered product and to withhold payment until the vendor meets all requirements.

10 **Representations and Warranties:** "slide 100"
Organizations should seek an express *representation and warranty* — this is a statement by which one party gives certain assurances to the other, and on which the other party may rely — in the document deliverables, that the licensed documentation whether a domain description a requirements prescriptions, or a software design (incl. code) does not infringe upon the intellectual property rights of any third parties.

11 **Dispute Resolution:** "slide 101"
Organizations should consider including dispute resolution provisions in contracts and licensing agreements. Such provisions enhance an organization's ability to resolve problems expeditiously and may provide for continued software development during a dispute resolution period.

12 **Agreement Modifications:** "slide 102"

Organizations should ensure software licenses clearly state that vendors cannot modify agreements without written signatures from both parties. This clause helps ensure there are no inadvertent modifications through less formal mechanisms some states may permit.

"slide 103"

13 **Vendor Liability Limitations:**
Some vendors may propose contracts that contain clauses limiting their liability.They may add provisions that disclaim all express or implied warranties or that limit monetary damages to the value of the product itself, specific liquidated damages, etc.. Generally, courts uphold these contractual limitations on liability in commercial settings unless they are unconscionable. Therefore, if organizations are considering contracts, they should consider whether the proposed damage limitation bears an adequate relationship to the amount of loss the financial organization might reasonably experience as a result of the vendor's failure to perform its obligations. Broad exculpatory clauses that limit a vendor's liability are a dangerous practice that could adversely affect the soundness of an organization because organizations could be injured and have no recourse.

"slide 104"

14 **IT Security:**
We interpret this contract aspect only in the light of software. There is an ISO recommendation of IT Security:INTERNATIONAL ISO/IEC STANDARD 17799 Reference number ISO/IEC 17799:2005(E), ISO/IEC 2005, ISO/IEC 17799:2005(E), Information technology, Security techniques: Code of practice for information security management, ISO copyright office, Case postale 56, CH-1211 Geneva 20, Switzerland. E-mail copyright@iso.org, Web www.iso.org. Published in Switzerland. Second edition, 2005-06-15. We advice clients and developers to carefully adhere to that ISO recommendation.

"slide 105"

15 **Subcontracting and Multiple Vendor Relationships:**
Some software vendors may contract third parties to develop software for their clients. To provide accountability, it may be beneficial for organizations to designate a primary contracting vendor. Organizations should include a provision specifying that the primary contracting vendor is responsible for the software regardless of which entity designed or developed the software. Organizations should also consider imposing notification and approval requirements regarding changes in vendor's significant subcontractors.

"slide 106"

16 **Restrictions and Adverse Comments:**
Some software licenses include a provision prohibiting licensees from disclosing adverse information about the performance of the software to any third party. Such provisions could inhibit an organization's participation in user groups, which provide useful shared experience regarding software packages. Accordingly, organizations should resist these types of provisions.

Design Briefs[14]

Characterisation 16 (Design Brief) By a *design brief* we understand a clearly delineated subset text of the contract. To recall (from the characterisation): This text (item (iii)) describes what is being contracted for possibly mutual deliveries, by dates, by contents, by quality, etc., and ((iv)) it details the specific development principles, techniques and tools; that is, the design brief directs the developers, the providers of what the contract primarily designates, as to what, how and when to develop what is being contracted. ∎

For examples of 'contract and design brief' modelling and resulting documents we refer to Appendix E, Sect. E.1.14, Page 330 and to Appendix M, Sect. M.1.15, Page 466.

1.6.15 Logbook

"SLIDE 108"

Characterisation 17 (Logbook) By a *logbook* we understand a record, a set of notes, which as correctly as is humanly feasible, lists the development, release, installation, use, maintenance, etc., history of a project. ∎

A logbook serves as a necessary reference in innumerable, usually unforeseeable instances of development.

"slide 109"

Example 1 (Logbook) An "abstracted" ... (dot, dot, dot) example is:

2 Jan. 1991: Initial meeting between partners *&c.*
...
31 May 1993: Acceptance of domain model *&c.*
...
24 October 1994: Acceptance of requirements model *&c.*
...
3 June 1996: Acceptance of software delivery *&c.*
...

The *&c.* signify reports, and the ... signify other logbook entries. . ∎

1.6.16 Discussion of Informative Documentation

"SLIDE 110"

General

We have identified some useful components of informative document parts. There may be other such informative parts. It all may depend on the universe of discourse, i.e., the problem at hand. We thus encourage the software developer to carefully reflect on which are the necessary and sufficient informative document parts.

There is usually a separate set of informative documents to be worked out for each phase of development: (i) the domain phase, (ii) the requirements phase, and (iii) the software design phase. "slide 111"

The current situation, needs, ideas, concepts, scope, span, synopsis and contract document parts differ in content between these phases. Usually the informative document parts, although crucially important, need not require excessive resources to develop, but their development must still be very careful!

In general, the informative document parts are concerned with the socioeconomic, even geopolitical, and hence pragmatic context of the projects about which they inform. As such they are "fluid", i.e., less precise, in what they aim at and what their objectives are. The next two documentation kinds are, in that respect, much more precise, and much more focused.

Methodological Consequences: Principle, Techniques and Tools[15]

Principle 1 (Information Document Construction) When first contemplating a new software development project, make sure — as the very first thing — to establish a proper complement of (all) informative documents. Throughout the entire development and after — during the entire lifetime of the result, whether a domain model, or a requirements model, or a software system — maintain this set of informative documents. ∎ "slide 113"

Principle 2 (Information Documents) The informative documents must be authoritative, definitive and interesting to read. ∎ "slide 114"

Technique 1 (Information Document Construction) First establish a document embodying the fullest possible table of contents, whether for just a domain development, or a requirements development, or a software design project, or for a combination of these. Then fill in respective document parts, "little by little", just a few sentences, using terse, precise, i.e., concise language, while avoiding descriptions (prescriptions and specifications) and analyses. Throughout maintain clear monitoring and control of all versions of these documents. ∎ "slide 115" "slide 116"

Tool 1 (Information Document Construction) A text processing system, preferably LaTeX, but MS Word will do, with good cross-referencing facilities, even between separately 'compilable' documents, provides a 'minimum' tool of documentation. Add to this a reasonably capable version monitoring and control system (such as CVS [59]) and you have a workable system. ∎

The subject of document version monitoring and control will not be dealt with in this volume. "slide 117" "slide 118"

[14] "SLIDE 107"

[15] "SLIDE 112"

1.7 Modelling Documents

"slide 119"

Documents which describe, prescribe or specify something, such document are intended to model those things. They, the document, are not those things, just conceptualisations, i.e., models of them. In this book we shall only seriously cover the modelling of domains and of requirements.

1.7.1 Domain Modelling Documents[16]

"slide 120"

Chapter 2 covers domain engineering in general and Sect. 2.9 covers domain modelling in particular.

Domain descriptions are documents. They are usually rather substantial. They usually include the following kinds of documents:

1	stakeholder identification and liaison records,	Sect. 2.4
2	acquisition sketches,	Sect. 2.5
3	business process rough sketches,	Sect. 2.7
4	terminologies,	Sect. 2.8
5	and domain models proper.	Sect. 2.9

"slide 121"

Chapter 2 will cover the domain engineering phase with its

- (i) stakeholder identification, Sect. 2.4
- (ii) domain acquisition, Sect. 2.5
- (iii) domain analysis and concept formation, Sect. 2.6
- (iv) business process rough sketching, Sect. 2.7
- (v) terminology, Sect. 2.8
- (vi) domain modelling, Sect. 2.9
- (vii) domain model verification, Sect. 2.10
- (viii) domain model validation, Sect. 2.11
- and (ix) domain theory formation Sect. 2.13

stages. Documents emerge from each of these stages.

Documents 1, 2, 3, 4 and 5 correspond to (i), (ii), (iv), (v) and (vi). The other activities are analytic.

1.7.2 Requirements Modelling Documents[17]

"slide 122"

Chapter 3 covers requirements engineering in general and Sect. 3.9 covers requirements modelling in particular.

Requirements prescriptions are documents. They are usually rather substantial. They usually include the following kinds of documents:

[16] By 'Domain Modelling Documents' we mean the same as by 'Domain Description Documents'.

[17] By 'Requirements Modelling Documents' we mean the same as by 'Requirements Prescription Documents'.

1 stakeholder identification and liaison records,
2 acquisition sketches,
3 business process re-engineering rough sketches,
4 terminologies, and
5 requirements models proper.

"slide 123"

Chapter 3 will cover the requirements engineering phase with its

- (i) stakeholder identification, Sect. 3.4
- (ii) requirements acquisition, Sect. 3.5
- (iii) requirements analysis and concept formation, Sect. 3.6
- (iv) business process re-engineering rough sketching, Sect. 3.7
- (v) terminology, Sect. 3.8
- (vi) requirements modelling, Sect. 3.9
- (vii) requirements model verification, Sect. 3.10
- (viii) requirements model validation, Sect. 3.11
- (ix) requirements feasibility and satisfiability analysis, Sect. 3.12
- and (x) requirements theory formation. Sect. 3.13

stages. Documents emerge from each of these stages.

Correspondence between Items 1–5 and Items (i–x) are as for corresponding domain stages and documents.

1.8 Analysis Documents

"slide 124"

1.8.1 Verification, Model Checks and Tests

Characterisation 18 (Analysis) By analysis we mean a process which results in a document and which analyses another document: a domain description, a requirements prescription, or a software design, and where the analysis is either a verification (in the sense of formally proving a property), or a model check (in the sense of writing another, mechanically analysable, document which "models" the former and checks whether it possesses a given property), or a formal (or even informal) test (in the sense of subjecting the former document to a form of "execution" to observe whether that execution yields a given result). ∎

1.8.2 Concept Formation

"slide 125"

Yet there is also another form of analysis. One that results in the analysing engineer forming a concept.

Characterisation 19 (Concept Formation) By concept formation we mean an analysis process in which the analysing engineer from analysed phenomena or analysed concrete concepts form a concept, respectively a "more" abstract, i.e., less concrete concept. ∎

1.8.3 Domain Analysis Documents "slide 126"

Chapter 2 covers domain engineering in general and Sects. 2.6 and 2.10–2.13 covers domain analysis in particular.

Stages (iii, vii, viii, ix) listed in Sect. 1.7.1 are analytic. They result in the following kinds of documents:

1	domain analysis (and concept formation)	see Sect. 2.6
2	domain model verification,	see Sect. 2.10
3	domain model validation,	see Sect. 2.11
4	and domain theory formation.	see Sect. 2.13

1.8.4 Requirements Analysis Documents "slide 127"

Chapter 3 covers domain engineering in general and Sects. 3.6 and 3.10–3.13 covers requirements analysis in particular.

Stages (iii, vii, viii, ix, x) listed in Sect. 1.7.2 are analytic. They result in the following kinds of documents:

1	requirements analysis (and concept formation),	Sect. 3.6
2	requirements model verification,	Sect. 3.10
3	requirements model validation,	Sect. 3.11
4	requirements feasibility and satisfiability,	Sect. 3.12
5	and requirements theory formation.	Sect. 3.13

1.9 Descriptions, Prescriptions, Specifications "slide 128"

1.9.1 Characterisations

We have, so far, used the terms descriptions, prescriptions and specifications — and we shall continue to use these terms — with the following meanings.

(A) *Descriptions* are of "what there is", that is, descriptions are, in this book, of domains, "as they are";

(B) *Prescriptions* are of "what we would like there to be", that is, prescriptions are, in this book, of requirements to software; and

(C) *Specifications* are of "how it is going to be", that is, specifications are, in this book, of software.

1.9.2 Reiteration of Differences "slide 129"

Descriptions are intended to state objective facts, i.e., are *indicative*. Prescriptions are intended to state commonly supposed and assumed to exist facts, i.e., are *putative* which we here take to be the same as *optative*: expressive of wish or desire. Specifications are intended to be expressive of a command, not to be avoided or evaded, i.e., are *imperative*. "slide 130"

Descriptions are intended to state objective facts, i.e., are *indicative*. Prescriptions are intended to state commonly supposed and assumed to exist facts, i.e., are *putative* which we here take to be the same as *optative*: expressive of wish or desire. Specifications are intended to be expressive of a command, not to be avoided or evaded, i.e., are *imperative*.

"slide 131"

(i) Software shall satisfy requirements.

(ii) Requirements defines properties of software.

(iii) Requirements must be commensurate with "their domain"; that is, requirements must satisfy all the properties of the domain insofar as these have not been re-engineered.

(iv) Requirements prescriptions denote requirements models.

(v) Requirements models are not the software, only abstractions of software.

(vi) Requirements models are computable adaptations of subsets of domain models.

(vii) Domains satisfy a number of laws.

(viii) Domain laws should be expressed by or derivable from domain descriptions.

(ix) Domain descriptions denote domain models.

(x) Domain models are not the domain, only abstractions of domains.

1.9.3 Rôle of Domain Descriptions

"slide 132"

Domain descriptions for common computing system (colloquially: IT) applications relate to requirements prescriptions and software specifications (incl. code) as physics relate to classical engineering artifacts: (a) electricity, plasma physics, etc., relate to electronics; (b) mechanics, aerodynamics, etc., relate to aeronautical engineering; (c) nuclear physics, thermodynamics, etc., relate to nuclear engineering; etcetera.

"slide 133"

Domain engineering relate to IT applications as follows: (d) transport domains to software (engineering) for road, rail, shipping and air traffic applications; (e) financial service industry domains to software (engineering) for banking, stock trading; portfolio management, insurance, credit card, etc., applications; (f) market trading ("the market") domains to software (engineering) for consumer, retailer, wholesaler, supply chain, etc., applications (aka "e-business"); etcetera.

The Sciences of Human and Natural Domains[18]

The 'Human Domains'

The domains for which most software systems are at play are — what we shall call — the human domains of financial service industries banks, insurance companies, stock

[18] "slide 134"

(etc.) trading brokers, traders, exchanges, etcetera; transportation industries roads, rails, shipping and air traffic; "the market" of consumers, retailers, wholesalers, product originators, and their distribution chains; etcetera,

The Natural Sciences[19]

In contrast the natural sciences includes physics: classical mechanics: statics, kinematics, dynamics, continuum mechanics: solid mechanics and fluid mechanics, mechanics of liquids and gases: hydrostatics, hydrodynamics, pneumatics, aerodynamics, and other fields; electromagnetism, relativity, thermodynamics and statistical mechanics, quantum mechanics, etcetera "slide 136"

The above listing is of disciplines within the natural sciences. It is not to be confused with a listing of research areas such as: condensed matter physics, atomic, molecular, and optical physics, high energy/particle physics, astrophysics and physical cosmology, etc.

Research Areas of the Human Domains[20]

To establish a domain description for an area within the human domain — for which there was no prior domain description — is a research undertaking — just as it is for establishing a domain description for an area within the domain of natural sciences. There are thus as many[21] human domain research areas as there are reasonably clearly separable such areas within the human domain.

Rôle of Domain Descriptions — Summarised[22]

That then is the rôle of domain descriptions to gain understanding, through research, and, independently, to obtain the right software: software that meet client expectations.

1.9.4 Rôle of Requirements Prescriptions "SLIDE 139"

A main rôle of a requirements prescription is to prescribe "the machine" !

The Machine

Characterisation 20 (Machine) By 'the machine' we shall mean a combination of hardware and software. ∎

[19] "SLIDE 135"
[20] "SLIDE 137"
[21] and we think: exciting research areas
[22] "SLIDE 138"

Machine Properties

The purpose of developing a requirements prescription is to prescribe properties of a machine.

1.9.5 Rough Sketches

"slide 140"

Characterisation 21 (Rough Sketch) By a *rough sketch* we mean an informal text which does not claim to be consistent or complete, and which attempts, perhaps in an unstructured manner, to outline a phenomenon or a concept. ∎

Rough sketches are useful "starters" towards narratives, and are used in acquired domain or requirements knowledge, and in outlining business processes and re-engineered such.

We refer to the rough sketch example of Sect. F.2 on page 343.

1.9.6 Narratives

"slide 141"

Characterisation 22 (Narrative) By a *narrative* we mean an informal text which is structured, which is claimed consistent and relative complete, and which informally defines a phenomenon or a concept. ∎

"slide 142"

Narratives will be our main "work horse", our chief means, at communicating domain descriptions and requirements prescriptions to all stakeholders.

"slide 143"

We refer to the narrative example of Page 343 of Sect. F.3.1 on page 343.

Characterisation 23 (Annotation) By an *annotation* we mean an informal text which is structured so as to follow, usually line-by-line a formal (mathematical) text which it aims at explaining to a lay reader not familiar with the mathematical formulas. ∎

"slide 144"

We usually mandate that all formulas be annotated. But we do not mandate a specific "formal" way of structuring the annotations.

We refer to the annotations example of Page 344 of Sect. F.3.1 on page 343 (which annotates the formalisation of Page 344 of Sect. F.3.1 on page 343).

1.10 Software

"slide 145"

1.10.1 What is Software ?

Characterisation 24 (Software) By software we understand: a set of documents: the domain development (incl. verification and validation) documents, the requirements development (incl. verification and validation) documents, and the software design development (incl. verification) documents. ∎

1.10.2 Software is Documents !

"SLIDE 146"

Domain Documents

The domain development documents include the informative documents and the documents which record stakeholder identification and relations, domain acquisition, domain analysis and concept formation, rough sketches of the business (i.e., domain) processes, terminologies, domain description, domain verification (incl. model check and test), domain validation and domain theory formation.

Requirements Documents[23]

The requirements development documents include the informative documents and the documents which record stakeholder identification and relations, requirements acquisition, requirements analysis and concept formation, rough sketches of the re-engineered business (i.e., new, revised domain) processes, terminologies, requirements description, requirements verification (incl. model check and test), requirements validation and requirements theory formation.

Software Design Documents[24]

And the software design development documents include the informative documents, the documents which record architectural designs ("how derived from requirements") and verifications (incl. model checks and tests), component designs and verifications (incl. model checks and tests), module designs and verifications (incl. model checks and tests), code designs and verifications (incl. model checks and tests), and the actual executable code documents.

Sections 2.15, 3.15 and 4.14 shall detail the above documents.

Software System Documents[25]

Characterisation 25 (Software System) By a *software[-based] system* we shall understand a set of software system documents (see below) as well as the hardware, the IT equipment for which the software is oriented: computers, their peripherals, data communication equipments, etcetera. ▪

"slide 150"

The *software system documents* include: the actual executable code documents, as well as ancillary documents: demonstration (i.e., demo) manuals, training manuals, installation manuals, user manuals, maintenance manuals, and development and maintenance logbooks.

[23] "SLIDE 147"

[24] "SLIDE 148"

[25] "SLIDE 149"

1.11 Informal and Formal Software Development "SLIDE 151"

In this book we shall advocate a combination of informal and formal development. And in this section we shall use the term specification (specify) to also cover description (describe) and prescription (prescribe), etc.

1.11.1 Characterisations

Informal Development

Characterisation 26 (Informal Development) By *informal development* we understand, in this book, a software development which does not use formal techniques, see below; instead it may use UML and an executable programming language. ∎

Formal Development[26]

Characterisation 27 (Formal Development) By *formal development* we mean, in this book, a software development which uses one or more formal techniques, see below, and it may then use these in a spectrum from systematically via rigorously to formally. ∎

"slide 153"

For characterisations of systematically, rigorously and formally we refer to charaterisations below.

Formal Software Development

Characterisation 28 (Formal Software Development Technique) By a *formal development technique* we mean, in this book, a software development in which specifications are expressed in a formal language, that is, a language with a formal syntax so that all specifications can be judged well-formed or not; a formal semantics so that all well-formed specifications have a precise meaning; and a (relatively complete) proof system such that one may be able to reason over properties of specifications or steps of formally specified developments from a more abstract to a more concrete step. Additionally a formal technique may be a calculus which allows developers to calculate, to refine "next", formally specified development steps from a preceding, formally specified step. ∎

Formal techniques are usually supported by software tools that check for syntactic and helps check for semantic correctness.

Examples of formal techniques, sometimes referred to as formal methods, are Alloy [122], ASM (Abstract State Machines) [188], B and event-B [4, 52], DC (Duration Calculus) [228], MSC and LSC (Message and Live Sequence

[26] "SLIDE 152"

Charts) [102, 118, 119, 120], Petri Nets [173, 124, 185, 184, 186], Statecharts [98, 99, 101, 103, 100], RAISE (Rigorous Approach to Industrial Software Engineering) [33, 34, 35, 85, 87, 86], TLA+ (Temporal Logic of Actions) [131, 132, 155], VDM (Vienna Development Method) [44, 45, 78] and Z [203, 204, 226, 108]. The EATCS[27] Monograph [43] arose from [188, 52, 69, 163, 86, 155, 108] and covers ASM, B and event-B, CafeOBJ, CASL, DC, RAISE, TLA+, VDM and Z.

This book will, in Vol. II, primarily feature the `RAISE` approach and thus use its `S`pecification `L`anguage `RSL`. For a more comprehensive introduction to formal techniques we refer to [33, 34, 35].

Systematic (Formal) Development ![28]

Characterisation 29 (Systematic (Formal) Development) By a *systematic use of a formal technique* we mean, in this book, a software development which which formally specifies whenever something is specified, but which does not (at least only at most in a minor of cases) reason formally over steps of development. ∎

Rigorous (Formal) Development ![29]

Characterisation 30 (Rigorous (Formal) Development) By a *rigorous use of formal techniques* we mean, in this book, a software development which which formally specifies whenever something is specified, and which formally express (some, if not all) properties that ought be expressed, but which does not (at least only at most in a minor number of cases) reason formally over steps of development, that is, verify these to hold, either by theorem proving, or by model checking, or by formally based tests. ∎

Formal (Formal) Development ![30]

Characterisation 31 (Formal (Formal) Development) By *formal use of a formal techniques* we mean, in this book, a software development which which formally specifies whenever something is specified, which formally expresses (most, if not all) properties that ought be expressed, and which formally verifies these to hold, either by theorem proving, or by model checking, or by formally based tests. ∎

[27] EATCS: European Association for Theoretical Computer Science

[28] "SLIDE 154"

[29] "SLIDE 155"

[30] "SLIDE 156"

1.11.2 Recommendations

"slide 157"

This book advocates that software development be pursued according to the triptych paradigm, and that the phases, stages and steps, as outlined in Chaps. 2–4, be pursued in a combination of both informal and formal descriptions, prescriptions and specifications, in a systematic to rigorous fashion.

"slide 158"
"slide 159"

1.12 Entities, Functions, Events and Behaviours

"slide 160"

So what is it that we describe, prescribe and specify, informally or formally ? The answer is: simple entities, operations, events and behaviours We shall, in this section, survey these concepts of of domains, requirements and software designs. In the domain we observe phenomena. "slide 161" From usually repeated such observations we form (immediate, abstract) concepts. We may then "lift" such immediate abstract concepts to more general abstract concepts. Phenomena are manifest. They can be observed by human senses (seen, heard, felt, smelled or tasted) or by physical measuring instruments (mass, length, time, electric current, thermodynamic temperature, amount of substance, luminous intensity). Concepts are defined.

"slide 162"

We shall analyse phenomena and concepts according to the following simple, but workable classification: *simple entities*, *functions* (over entities), *events* (involving changes in entities, possibly as caused by function invocations, i.e., *actions*, and/or possibly causing such), and *behaviours* as (possibly sets of) sequences of actions (i.e., function invocations) and events.

1.12.1 Simple Entities

"slide 163"

Characterisation 32 (Simple Entity) By a *simple entity* we mean something we can point to, i.e., something manifest, or a concept abstracted from, such a phenomenon or concept thereof. ∎

Simple entities are either atomic or composite. The decision as to which simple entities are considered atomic or composite is a decision solely taken by the describer.

Atomic Entities[31]

Characterisation 33 (Atomic Entity) By an *atomic entity* we intuitively understand a simple entity which "cannot be taken apart" (into other, the sub-entities) and which possess one or more attributes. ∎

[31] "slide 164"

Attributes — Types and Values:[32]

With any entity we can associate one or more attributes.

Characterisation 34 (Attribute) By an *attribute* we understanda pair of a **type** and a **value**. ▪

"slide 166"

Example 2 (Atomic Entities)

Entity: **Person**		Entity: **Bank Account**	
Type	**Value**	**Type**	**Value**
Name	Dines Bjørner	number	212 023 361 918
Weight	118 pounds	balance	1,678,123 Yen
Height	179 cm	interest rate	1.5 %
Gender	male	credit limit	400,000 Yen

. ▪

"Removing" an attribute from an entity destroys its "entity-hood".

Composite Entities[33]

Characterisation 35 (Composite Entity) By a *composite entity* we intuitively understand an entity (i) which "can be taken apart" into sub-entities, (ii) where the composition of these is described by its *mereology*, and (iii) which, apart from the attributes of the sub-entities, further possess one or more attributes. ▪

Sub-entities are entities.

Mereology[34]

Characterisation 36 (Mereology) By *mereology* we understanda theory of part-hood relations. That is, of the relations of part to whole and the relations of part to part within a whole. ▪

The term mereology seems to have been first used in the sense we are using it by the Polish mathematical logician Stanisław Leshniewski [145, 159, 206, 207, 214].

[32] "SLIDE 165"
[33] "SLIDE 167"
[34] "SLIDE 168"

Composite Entities — Continued[35]

Example 3 (Transport Net, A Narrative)

Entity: **Transport Net**		
Subentities:	Segments Junctions	
Mereology:	"set" of one or more s(egment)s and "set" of two or more j(unction)s such that each s(egment) is delimited by two j(unctions) and such that each j(unction) connects one or more s(egments)	
Attributes		
	Types:	**Values:**
	Multimodal	Rail, Roads
	Transport Net of	Denmark
	Year Surveyed	2006

. ∎

"slide 170"

"slide 171"

To put the above example of a composite entity in context we give and example of both an informal narrative and a corresponding formal specification:

Example 4 (Transport Net, A Formalisation) A transport net consists of one or more segments and two or more junctions. With segments [junctions] we can associate the following attributes: segment [junction] identifiers, the identifiers of the two junctions to which segments are connected [the identifiers of the one or more segments connected to the junction], the mode of a segment [the modes of the segments connected to the junction]

"slide 172"

type
  N, S, J, Si, Ji, M
value
  obs_Ss: N → S-**set**, obs_Js: N → J-**set**
  obs_Si: S → Si, obs_Ji: J → Ji
  obs_Jis: S → Ji-**set**, obs_Sis: J → Si-**set**
  obs_M: S → M, obs_Ms: J → M-**set**
axiom
  ∀ n:N • **card** obs_Ss(n) ≥ 1 ∧ **card** obs_Js(n) ≥ 2
  ∀ n:N • **card** obs_Ss(n) ≡ **card** {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
  ∀ n:N • **card** obs_Js(n) ≡ **card** {obs_Ji(c)|j:J • j ∈ obs_Js(n)} ...
type
  Nm, Co, Ye
value
  obs_Nm: N → Nm, obs_Co: N → Co, obs_Ye: N → Ye

Si, Ji, M, Nm, Co, Ye are not entities. They are names of attribute types and, as such, designate attribute values. N is composite, S and J are considered atomic . ∎

[35] "SLIDE 169"

States[36]

Characterisation 37 (State) By a domain *state*we shall understand a collection of domain entities chosen by the domain engineer. ∎

The pragmatics of the notion of state is that states are recurrent arguments to functions and are changed by function invocations.

Formal Modelling of Entities[37]

How do we model entities ? The answer is: by selecting a name for the desired "set", that is, type of entities; by defining that type to be either an abstract type, i.e., a sort,

```
type
  A
```

or a concrete type, i.e., with defined, concrete values.

```
type
  A = Type_Expression
```

"slide 175"

Values of the type are then expressed as:

```
value
  a:A
```

As our main support example unfolds in Vol. II we shall illustrate sorts with their observer functions and concrete types over either basic types (Booleans, integers, natural numbers, reals, etc., or over composite types (sets, Cartesians, records, lists, maps, functions). Appendix Sect. S.1 (Pages 503–506) gives a terse introduction to the type system of our main formal specification language RSL.

1.12.2 Functions

"SLIDE 176"

Characterisation 38 (Function) By a *function* we shall understand something which when *applied* to what we shall call *arguments* (i.e., entities) *yield* some entities called the *result* of the function (application). ∎

Actions

Characterisation 39 (Action) By an *action* we shall understand the same thing as applying a state-changing function to its arguments (including the state). ∎

[36] "SLIDE 173"

[37] "SLIDE 174"

Functions — Resumed[38]

The observer functions of the formal example above are not the kind of functions we are (later) seeking to identify in domains and requirements. These observer functions are mere technicalities: needed, due to the way in which we formalise — and are deployed in order to express sub-entities, mereologies and attributes.

Function Signatures[39]

Characterisation 40 (Function Signature) By a *function signature* we mean the *name and type* of a function.

```
type
  A, B, ..., C, X, Y, .., Z
value
  f: A × B × ... × C → X × Y × ... × Z
```

The last line above expresses a schematic function signature. ∎

Function Descriptions[40]

Characterisation 41 (Function Description) By a function description we mean a function signature and something which describes the relationship between function arguments (the a:A's, b:B's, ..., c:C's and the x:X's, y:Y's, ..., z:Z's). ∎

"slide 180"

Example 5 (Well Formed Routes)

```
type
  P = Ji × Si × Ji          /∗ path: triple of identifiers ∗/
  R′ = P∗                   /∗ route: sequence of connected paths ∗/
  R = {| r:R′ • wf_R(r) |}  /∗ subtype of R′: those r′s satisfying wf_R(r) ∗/
value
  wf_R: R′ → Bool
  wf_R(r) ≡
    ∀ i:Nat•{i,i+1}⊆inds r⇒let (,,ji′)=r(i),(ji″,,)=r(i+1) in ji′=ji″ end
```

The last line above describes the route wellformedness predicate. The meaning of the "(,," and ",,)" is that the omitted path components "play no rôle" ∎

[38] "SLIDE 177"
[39] "SLIDE 178"
[40] "SLIDE 179"

1.12.3 Events

"SLIDE 181"

Characterisation 42 (Event)

- An event can be characterised by
  - ⋆ a predicate, p and
  - ⋆ a pair of ("before") and ("after") of pairs of
    - · states and
    - · times:
    - · $p((t_b, \sigma_b), (t_a, \sigma_a))$.
  - ⋆ Usually the time interval $t_a - t_b$
  - ⋆ is of the order $t_a \simeq \text{next}(t_b)$

. ∎

"slide 182"

Sometimes the event times coincide, $t_b = t_a$, in which case we say that the event is instantaneous. The states may then be equal $\sigma_b = \sigma_a$ or distinct !

We call such predicates as p for event predicates.

By an *event* we shall thus, to paraphrase, understand an instantaneous change of state not directly brought about by some explicitly willed action in the domain, but either by "external" forces. or implicitly as a non-intended result of an explicitly willed action.

Events may or may not lead to the initiation of explicitly issued operations. "slide 183"

Example 6 (Events) A 'withdraw' from a positive balance bank account action may leave a negative balance bank account. A bank branch office may have to temporarily stop actions, i.e., close, due to a bank robbery. ∎

Internal events: The first example above illustrates an internal event. It was caused by an action in the domain, but was not explicitly the main intention of the "withdraw" function.

External events: The second example above illustrates an external event. We assume that we have not explicitly modelled bank robberies!

1.12.4 Behaviours

"SLIDE 184"

Simple Behaviours

Characterisation 43 (Simple Behaviour) By a *simple behaviour*

- we understand a sequence, q, of zero, one or more
  - ⋆ actions
  - ⋆ and/or events
  - ⋆ $q_1, q_2, \ldots, q_i, q_{i+1}, \ldots, q_n$
- such that the state
  - ⋆ resulting from one such action, q_i,
  - ⋆ or in which some event, q_i, occurs,
- becomes the state in which the next action or event, q_{i+1},

- ⋆ if it is an action, is effected,
- ⋆ or, if it is an event, is the event state

. ∎

"slide 185"

Example 7 (Simple Behaviours) The opening of a bank account, followed by zero, one or more deposits into that bank account, and/or withdrawals from the bank account in question, ending with a closing of the bank account. Any prefix of such a sequence is also a simple behaviour. Any sequence in which one or more events are interspersed is also a simple behaviour. ∎

General Behaviours[41]

A *behaviour* is either a *simple behaviour*, or is a *concurrent behaviour*, or, if the latter, can be either a *communicating behaviour* or not (i.e., just a concurrent behaviour).

Concurrent Behaviours[42]

Characterisation 44 (Concurrent Behaviour) By a concurrent behaviour we shall understand a set of behaviours (simple or otherwise). ∎

"slide 188"

Example 8 (Concurrent Behaviours) A set of simple behaviours, that may result from two or more distinct bank clients, each operating of their own, distinct, that is, non-shared accounts, forms a concurrent behaviour. ∎

Communicating Behaviours[43]

Characterisation 45 (Communicating Behaviour) By a *communicating behaviour* we shall understand a set of two or more behaviours where otherwise distinct elements (i.e., behaviours) share events. ∎

"slide 190"

Sometimes we do not model the behaviour from which external events are incident (i.e., "arrive") or to which events emanate (i.e., "depart"). But such an environment is nevertheless a behaviour.

"slide 191"

Example 9 (Communicating Behaviours) Consider a bank.To model that two or more clients can share the same bank account one could model the bank account as one behaviour and each client as a distinct behaviour. Let us assume that only one client can open an account and that only one client can close an account. Let us further assume that sharing is brought about by one client, say the one who opened the account, identifying the sharing clients. Now, in order to make sure that at most one client accesses the shared account

[41] "SLIDE 186"
[42] "SLIDE 187"
[43] "SLIDE 189"

at one one time (in any one "smallest" transaction interval) one may model "client access to account" as a pair of events such that during the interval between the first (begin transaction) and the second (end transaction) event no other client can share events with the bank account behaviour. Now the set of behaviours of the bank account and one or more of the client behaviours is an example of a communicating behavior. ∎

Formal Modelling of Behaviours[44]

Communicating behaviours, the only really interesting behaviours, can be modelled in a great variety of ways: from set-oriented models in B [4, 53], RSL [85, 87, 33, 34, 35, 84, 41], VDM [44, 45, 78, 77], or Z [203, 204, 226, 108, 107], to models using for example CSP [110, 194, 196, 111],(as for example "embedded" in RSL [85],), or, to diagram models using, for example, Petri nets [124, 173, 185, 184, 186], message [118, 119, 120] or live sequence charts [63, 102, 126], or state-charts [98, 99, 101, 103, 100].

1.12.5 Discussion "SLIDE 193"

The main aim of Sect. 1.12 is to ensure that we have a clear understanding of the modelling concepts of entities, functions, events and behaviours. To "reduce" the modelling of phenomena and concepts to these four kinds of phenomena and concepts is, of course, debatable. Our point is that it works, that further classification, as is done in for example John F. Sowa's [202], is not necessary, or rather, is replaced by how we model attributes of, for example, entities, and how we model facets (Sect. 2.9.2, next chapter).

1.12.6 Functions, Events and Behaviours as Entities

Review of Entities

In the example of Chap. F we identify the following as being entities: (i) nets (Item 6 on page 344), (ii) links (Item 5 on page 343), (iii) hubs (Item 5), (iv) insert commands (Item 18 on page 349), (v) remove commands (Item 19 on page 349), (vi) time (Item 31 on page 358), (vii) time intervals (Item 35 on page 359), (viii) vehicles (Item 42 on page 359), (ix) positions (Item 43 on page 359) and (x) traffic (Item 44 on page 360).

It may surprise some that we designate the insert and remove commands as entities. They are certainly of conceptual nature, but can be given manifest representations in the form of documents (that, for example order the building of a link and its eventual inclusion in the net).

It may surprise some that we designate time and time intervals as entities. They are certainly of conceptual and very abstract nature, but so is our choice.

[44] "SLIDE 192"

It may surprise some that we designate positions as entities. They are certainly manifest: one can point to a position.

And it may finally surprise some that we designate traffics as entities. It is certainly manifest, and can be recorded, say by video-recording the traffic. So that is also our choice.

Functions as Entities

TO BE WRITTEN

Events as Entities

TO BE WRITTEN

Behaviours as Entities

TO BE WRITTEN

1.13 Domain vs. Operational Research Models "SLIDE 194"

1.13.1 Operational Research (OR)

Since World War II, as a result of research and application of what became known as OR models (OR for Operational Research), these have won a significant position also within the transportation infrastructure. But domain models are not OR models. OR models usually use classical applied mathematics: calculus ([partial] differential equations), statistics, probability theory, graph theory, combinatorics, signal analysis, theory of flows in networks, etcetera where domain engineering use formal specification languages emphasising applied mathematical logic and modern algebra.

1.13.2 Reasons for Operational Research Analysis "SLIDE 195"

OR models are established, that is, OR analysis is performed, for the following reasons: to solve a particular problem, usually a resource allocation and/or scheduling problem, but also, less often, the problem is one of taking advice: should an investment be made, should one form of resource be "converted" into another form, etc. Once solved the solver and the client knows how to best allocate and/or schedule the investigated resource or whether to perform a certain kind of investment, etc. OR models typically do not themselves lead to software derived from the OR model, but sometimes results of OR analysis become constants in or parameters for otherwise independently developed software.

1.13.3 Domain Models

"SLIDE 196"

Domain models are usually established (i) to understand an area of a domain much wider than that analysable by current OR techniques, and sometimes (ii) for purposes of "deriving" appropriate requirements, and (iii) for implementing the right software. It has then turned out that in order to achieve items (i–iii) above one has to use the kind of mathematics shown in this book.

1.13.4 Domain and OR Models

"SLIDE 197"

But domain and OR modelling are not really that separated — as it may appear from the above. Oftentimes software (as well as hardware) design decisions must (or ought to) be based on OR analysis. The two kinds of modelling must still be pursued. But it is desirable that their scientists and engineers, i.e., that their practitioners, collaborate. Today they do not collaborate. Today only the domain engineers are aware of the existence of OR engineers.

1.13.5 Domain versus Mathematical Modelling

"SLIDE 198"

We could widen our examination of domain modelling versus OR modelling to domain modelling versus mathematical modelling, where the latter extends well beyond OR modelling to the modelling of physical and human made domains in its widest sense — such as also practiced by physicists, biologists, etc.

For OR modelling as well as mathematical modelling we can say that domain modelling currently lacks the formal techniques offered by the former.

But we are digressing !

1.14 Summary

"SLIDE 199"

The exercises of this chapter, see next, reveal the essence of this chapter: (i) the 'triptych paradigm' (Sect. 1.2); (ii) the 'triptych phases of software engineering' (Sect. 1.3); (iii) the 'stages' and 'steps' of software development (Sect. 1.4); (iv) the three classes of development documents (Sect. 1.5); (v) the detailed nature of 16 kinds of 'informative documents' (Sect. 1.6); (vi) the concepts of 'modelling documents' (Sect. 1.7); (vii) the concepts of 'analysis documents' (Sect. 1.8); (viii) the concepts of 'descriptions, prescriptions' and 'specifications' (Sect. 1.9); (ix) the concept of 'software' (Sect. 1.10); (x) the concepts (Sect. 1.11) of 'informal development', 'formal development', 'informal and formal development', 'formal software development technique', 'systematic development', 'rigorous development' and 'formal development'; (xi) the concepts of 'entities', 'functions', 'events' and 'behaviours' (Sect. 1.12); and (xii) the concepts of 'operational research' versus those of 'domain models' (Sect. 1.13).

"slide 200"

1.15 Exercises

Exercise 1. The Triptych Paradigm: Rehearse the text of the triptych dogma so that you can express it, "by heart", precisely.
Solution 1 Vol. II, Page 529, suggests a way of answering this exercise.

Exercise 2. The Triptych Phases of Software Development: Rehearse the text of the triptych phases of software development so that you can express it, "by heart", precisely.
Solution 2 Vol. II, Page 529, suggests a way of answering this exercise.

Exercise 3. Phases, Stages and Steps of Software Development: Explain the concepts of software development phase, stage and step.
Solution 3 Vol. II, Page 529, suggests a way of answering this exercise.

Exercise 4. Development Documents:
Enumerate the three kinds of development documents outlined in this chapter, that is, express them, "by heart", precisely.
Solution 4 Vol. II, Page 530, suggests a way of answering this exercise.

Exercise 5. Enumeration of Informative Documents:
Enumerate, as best as you can, the 16 kinds of informative documents outlined in this chapter — express them, "by heart", as precisely as you can.
Solution 5 Vol. II, Page 530, suggests a way of answering this exercise.

Exercise 6. Descriptions, Prescriptions, Specifications:
In which contexts are the terms descriptions, prescriptions and specifications used in this book, such as proclaimed in this chapter.
Solution 6 Vol. II, Page 530, suggests a way of answering this exercise.

Exercise 7. Software:
Explain what the term 'software' covers, such as characterised in this chapter. That is: list as you can best remember, the names of the documents that together "make up" software.
Solution 7 Vol. II, Page 530, suggests a way of answering this exercise.

Exercise 8. Informal and Formal Software Development:
Explain, as precisely as possible, the terms 'informal development', 'formal development', 'informal and formal development', 'formal software development technique', 'systematic development', 'rigorous development' and 'formal development' such as characterised in this chapter.

Solution 8 Vol. II, Page 532, suggests a way of answering this exercise.

Exercise 9. Entities and States, Functions and Actions, Events and Behaviours:

Please define, as close to the characterisations of this chapter the notions of entities, states, functions, actions, events and behaviours.
Solution 9 Vol. II, Page 532, suggests a way of answering this exercise.

Exercise 10. Mereology, Atomic and Composite Entities:

Please define, as close to the characterisations of this chapter the notion of entities: atomic and composite. Focus, in particular, on the issue of the attributes of composite entities, their sub-entities and their mereology.
Solution 10 Vol. II, Page 533, suggests a way of answering this exercise.

"slide 201"

Part II

A Triptych of Software Engineering

This part consists, as is implied by the 'triptych' adjective, of three chapters.

We have, for each of these three chapters, perhaps somewhat "artificially", structured these according to almost the same sectioning:

1 Informative Documents
2 Stakeholder Identification
3 Acquisition
4 Analysis & Concept Formation
5 $\mathcal{D}$ Business Processes
  $\mathcal{R}$ Business Process Re-Engineering
  $\mathcal{S}$ Software Design Options
6 Terminology
7 Modelling
8 Verification
9 Validation
10 $\mathcal{D}$ Verification versus Validation
   $\mathcal{R}$ Satisfiability and Feasibility
   $\mathcal{S}$ Release, Transfer & Maintenance
11 $\mathcal{D}, \mathcal{R}$ Theory Formation
   $\mathcal{S}$ Documentation

The $\mathcal{D}$, $\mathcal{R}$, $\mathcal{S}$ and$\mathcal{D}, \mathcal{R}$ markers shall indicate that the associated section title relates to only the domain, the requirements, the software design, or to both domain and requirements development. No marker shall indicate that the section title is common to all the triptych phases.

Of course, for these 'commonalities', there are content-wise differences.

Several of the 'common'-type sections, of respective chapters, that is, several of the phase stages, cover almost identical methodology steps.

• • •

Here are more detailed listings of the respective chapter topics:

Chapter 2: Domain Engineering

Chapter 3: Requirements Engineering

Chapter 4: Software Design

“slide 202”

2

Domain Engineering

"slide 203"

Domain engineering is a new element of software engineering. Domain engineering is to be performed prior to requirements engineering for the case where there is no relevant domain description on which to base the requirements engineering. For the case that such a description exists that description has to first be checked: its scope must cover at least that of the desired requirements. "slide 204"

This chapter shall outline the stages and steps of development actions to be taken in order to arrive, in a proper way, at a proper domain description.

2.1 Discussions of The Domain Concept

"slide 205"

2.1.1 The Novelty

The idea of domain engineering preceding requirements engineering is new. Well, in some presentations of requirements engineering there are elements of domain analysis. But basically those requirements engineering-based forms of analysis do not expect the requirements engineer to write down, that is, to seriously describe the domain, and certainly not in a form which is independent of, that is, separated from the requirements prescriptions. "slide 206"

As also outlined in Sects. 1.2 and 1.13, domain models are as necessary for requirements development and — thus also — for software design, as physics is for the classical branches of electrical and electronics, mechanics, civil, and chemical engineering.

2.1.2 Implications

"slide 207"

This new aspect of software engineering implies that software engineers, as a group, engaged in a software development project, from (and including) domain engineering via requirements engineering to (and including) software design, must possess the necessary formal and practical bases: the science skills of domain engineering, the R&D skills of requirements engineering, and the (by now) engineering skills of software design.

2.1.3 The Domain Dogma

"slide 208"

From Sect. 1.2 we repeat:

> *Before software can be designed one must understand its requirements. Before requirements can be expressed one must understand the application domain.*

2.2 Stages of Domain Engineering

"slide 209"

2.2.1 An Overview of "What to Do ?"

How do we then construct a domain description ? That is, which are the stages of domain engineering ? The answer is: there are a number of stages, which, when followed in some order, some possibly concurrently, will lead you reasonably disciplined way from scratch to goal ! Before enumerating the stages let us argue their presence and basic purpose.

[1] Domain Information[1]

We are here referring to the construction of informative documents.

We have earlier, as mentioned above, extensively (Pages 6–24) covered the general issues of informative documents. The reader is strongly encouraged to review those pages, Sect. 1.5.

Suffice it here to restate that each and every of the items listed on Page 7 must be kept up-to-date during the full development cycle. This means that this activity is of "continuing concern" all during development.

"slide 211"

The purpose of this stage of development, to repeat, is to record all relevant administrative, socio-economic, budgetary, project management (planning) and all such non-formalisable information which has a bearing on the domain description project.

[2] Domain Stakeholder Identification[2]

The domain is populated with staff (management, workers, etc.), customers (clients, users), providers of support, equipment, etc., the public at large — always "interfering, having opinions", regulatory agencies, politicians seeking "14 minutes of TV coverage", etcetera.

There are many kinds of staff, many kinds of customers, many kinds of providers, etc. All these need be identified so that as complete a coverage of sources of domain knowledge can be established and used when actively acquiring, that is, soliciting and eliciting knowledge about the domain.

Section 2.4 will elaborate on this topic.

[1] "slide 210"

[2] "slide 212"

[3] Domain Acquisition[3]

The software engineers need a domain description. Software engineers, today, are basically the only ones who have the tools[4], techniques and experience in creating large scale specifications. But the software engineers do not possess the domain knowledge. They must solicit and elicit, that is, they must acquire this knowledge from the domain stakeholders. "slide 214"

Characterisation 46 (Domain Acquisition (I)) By *domain acquisition* we understand a process in which documents, interviews, etc., informing — "in any shape or form" — about the domain entities, functions, events and behaviours are collected from the domain stakeholders. ▪

Compare the above characterisation to that of Characterisation 50 on page 57.

Section 2.5 will elaborate on this topic.

[4] Domain Analysis and Concept Formation[5]

The acquired domain knowledge is then analysed, that is, studied with a view towards discovering inconsistencies and incompleteness of what has been acquired as well as concepts that capture properties of knowledge about the phenomena and concepts being analysed.

Section 2.6 will elaborate on this topic.

[5] Domain Business Processes[6]

On the basis of acquired knowledge, sometimes as part of its acquisition one is either presented with or constructs rough sketches of the business processes of the domain. An aim of describing these business processes is to check the acquired knowledge for inconsistencies and completeness and whether proposed concepts help improve the informal understanding.

Section 2.7 will elaborate on this topic.

[6] Domain Terminology[7]

Out of the domain acquisition, analysis and business process rough-sketching processes emerges a domain terminology. That is, a set of terms that cover entities, functions, events and behaviours of the domain. It is an important aspect of software development to establish, use and maintain a variety of terminologies. And first comes the domain terminology. "slide 218"

Section 2.8 will elaborate on this topic.

[3] "SLIDE 213"

[4] The two main tools of domain description are concise English and a number of formal specification languages.

[5] "SLIDE 215"

[6] "SLIDE 216"

[7] "SLIDE 217"

[7] Domain Modelling[8]

Based on properly analysed domain acquisitions these are "domain description units" we can now model the domain. The major stage of the domain engineering phase is that of domain modelling, that is, of precisely describe in narrative and possibly also in formal terms the domain as it is. Several principles, many techniques and many tools can be given for describing domains.

Section 2.9 will elaborate on this topic.

[8] Domain Verification[9]

While describing a domain one may wish to verify properties of what is being described. The use here of the term 'verification' covers (i) formal testing, that is, testing (symbolic executions of descriptions) based on formally derived test cases and test answers, (ii) model checking, that is, executions of simplified, but crucial models of what is being described, and (iii) formal verification that is, formal, possibly mechanisable proof of theorems (propositions etc.) about what is being described.

Section 2.10 will elaborate on this topic.

[9] Domain Validation[10]

Characterisation 47 (Validation) By *validation* we shall mean a systematic process — involving representatives of all stakeholders and the domain engineers — going carefully through all the narrative descriptions and confirming or rejecting these descriptions. ▪

Section 2.11 will elaborate on this topic.

[10] Domain Verification versus Domain Validation[11]

Verification serves to ensure that the domain model is right. Validation serves to ensure that one obtains the right model.

[11] Domain Theory Formation[12]

Describing a domain, precisely, and even formally, verifying propositions and theorems, is tantamount to establishing a basis for a domain theory. Just as in physics, we need theories also of the man-made universes.

Section 2.13 will elaborate on this topic.

[8] "SLIDE 219"
[9] "SLIDE 220"
[10] "SLIDE 221"
[11] "SLIDE 222"
[12] "SLIDE 223"

2.2.2 A Summary Enumeration

"slide 224"

We can now summarise the relevant stages of domain engineering:

1 Domain Information Sect. 2.3 (Page 55)
2 Domain Stakeholder Identification, Sect. 2.4 (Page 56)
3 Domain Acquisition, Sect. 2.5 (Page 57)
4 Domain Analysis and Concept Formation, Sect. 2.6 (Page 60)
5 Domain [i.e., Business] Processes, Sect. 2.7 (Page 62)
6 Domain Terminology, Sect. 2.8 (Page 63)
7 Domain Modelling, Sect. 2.9 (Page 64)

(a) Intrinsics Sect. 2.9.4 (Page 65)
(b) Support TechnologiesSect. 2.9.5 (Page 70)
(c) Management & Organistation Sect. 2.9.6 (Page 73)
(d) Rules & Regulations Sect. 2.9.7 (Page 80)
(e) Scripts Sect. 2.9.8 (Page 84)
(f) Human Behaviour Sect. 2.9.9 (Page 100)

8 Domain Verification, Sect. 2.10 (Page 102)
9 Domain Validation and Sect. 2.11 (Page 102)
10 Domain Verification Versus Domain Validation and Sect. 2.12 (Page 102)
11 Domain Theory Formation, Sect. 2.13 (Page 102)

2.3 Domain Information

"slide 225"

We highlight, in the next many "highlighted" paragraphs, what is special, to domain description developments, with respect to the many items of project information.

Current Situation: Cf. Sect. 1.6.3, Page 9 "slide 226"

As mentioned in Sect. 1.6.3 on page 9 the context in which the domain developments starts must be emphasized. Focus on just that. Please no reference to possible requirements or software designs.

MORE TO COME

Needs and Ideas: Cf. Sect. 1.6.4, Pages 9–10 "slide 227"

TO BE WRITTEN

Concepts and Facilities: Cf. Sect. 1.6.5, Pages 10–11 "slide 228"

TO BE WRITTEN

Scope and Span: Cf. Sect. 1.6.6, Page 11 "slide 229"

TO BE WRITTEN

Assumptions and Dependencies: Cf. Sect. 1.6.7, Pages 11–12 "slide 230"

TO BE WRITTEN

"slide 231" **Implicit/Derivative Goals:** Cf. Sect. 1.6.8, Page 12

TO BE WRITTEN

"slide 232" **Synopsis:** Cf. Sect. 1.6.9, Page 13

TO BE WRITTEN

"slide 233" **Software Development Graphs:** Cf. Sect. 1.6.10, Pages 13–15

TO BE WRITTEN

"slide 234" **Resource Allocation:** Cf. Sect. 1.6.11, Pages 15–16

TO BE WRITTEN

"slide 235" **Budget (and Other) Estimates:** Cf. Sect. 1.6.12, Page 16

TO BE WRITTEN

"slide 236" **Standards Compliance:** Cf. Sect. 1.6.13, Pages 16–19

TO BE WRITTEN

"slide 237" **Contracts and Design Briefs:** Cf. Sect. 1.6.14, Pages 19–23

TO BE WRITTEN

2.4 Domain Stakeholders "slide 238"

2.4.1 Characterisations

Characterisation 48 (Stakeholder) By a domain *stakeholder* we shall understand a person, or a group of persons, "united" somehow in their common interest in, or dependency on the domain; or an institution, an enterprise, or a group of such, (again) characterised (and, again, loosely) by their common interest in, or dependency on the domain. ∎

"slide 239"

Characterisation 49 (General Application Domain Stakeholder) By *general application domain stakeholders* we understand stakeholders whose primary interest is neither the projects which develop software (from domains, via requirements to software design), nor the products evolving from such projects. Instead we mean stakeholders from typically non-IT business areas. ∎

2.4.2 Why Be Concerned About Stakeholders ?

"SLIDE 240"

The domain stakeholders are the main sources of domain knowledge. So the domain engineers must acquire as much and more than the knowledge relevant to describe the domain. And the domain stakeholders must eventually validate the domain engineers' domain description.

2.4.3 How to Establish List of Stakeholders ?

"SLIDE 241"

Awareness, by the domain engineers, of who and which are the main and the subordinate domain "players", is obtained by the same initial processes that first acquire domain knowledge, namely by reading about the domain, from books, journals, the Internet, by talking to stakeholders, and by interviewing these systematically.

The process is an iterative one. One cannot know till "deep" into domain modelling whether one has obtained a reasonably complete list.

2.4.4 Form of Contact With Stakeholders

"SLIDE 242"

Sections 2.5 and 2.11 outlinethe regular interactions between domain stakeholders and domain engineers from the early stages of domain acquisition to the late stage of domain validation. This form of domain stakeholder and engineers interaction alternates betweenone-on-one meetings, e-mails, the joint filling out of larger questionnaires, and joint multi-stakeholder group and domain engineer presentations. The domain engineers shall carefully keep record of all that is communicated.

For examples of domain stakeholder identification modelling and resulting documents we refer to Appendix E, Sect. E.2 (Page 332).

2.5 Domain Acquisition

"SLIDE 243"

2.5.1 Another Characterisation

Characterisation 50 (Domain Acquisition (II)) By *domain acquisition* we shall here understand the systematic solicitation and elicitation of knowledge about the chosen domain and the systematic vetting, recording and classification of this knowledge. ∎

Compare the above characterisation to that of Characterisation 46 on page 53.

2.5.2 Sources of Domain Knowledge

"SLIDE 244"

To return to the issue of stakeholders, from where does the domain engineer acquire the domain knowledge ? The answer is: from many (stakeholder) sources. We suggest some sources: from the Internet[13], from infrastructure books, papers, etc.[14], from owners and staff of the client[15], from customers of the client[16], possibly from domain regulators[17], from consultancy, equipment and service providers for and to the client[18] and possibly others.

2.5.3 Forms of Solicitation and Elicitation

"SLIDE 245"

Solicitation

How can the domain engineer solicit[19] the desired domain knowledge ? By searching the Internet, looking up books, papers and reports (the latter typically from university and college institutes and from libraries); and by contacting and by asking to be referred to domain knowledgeable client and customer staff.

Elicitation[20]

How does the domain engineer elicit[21] the desired domain knowledge ? By studying hopefully relevant Internet Web pages, books, papers and reports and by forming "impressions of" ("first ideas about") the domain from such studies; and by interviewing ("questionnairing") contacted domain stakeholders, with interviews being based on the prior 'impressions' from Web pages, books, papers, reports, or from other stakeholder interviews.

[13] For each infrastructure domain: air traffic, airports, banking, health care in general and hospitals in particular, for railways, roads, shipping, etc., there are many Web pages that can be searched.

[14] Similarly to footnote 13.

[15] This includes all management levels [executive (strategic), tactical and operational management], planners, schedulers, and "blue collar" workers (!).

[16] Notice the distinction between client and customer: By client we here refer to the domain institution with whom the domain engineers have a contract for developing a domain description. By customer we here refer to that client's customers.

[17] Most, if not all, domains have their own regulators. The air line industry have their global and national civil aviation organisations or authorities. The banking industry have their federal or national finance "watchdogs". Etcetera.

[18] We exclude the developers from this list.

[19] To solicit: to try to obtain by usually urgent requests or pleas.

[20] "SLIDE 246"

[21] To elicit: to call forth or draw out.

2.5.4 Solicitation and Elicitation

"SLIDE 247"

Solicitation and elicitation is an iterative process: Impressions obtained early in the process may turn out to be wrong. Hence they must be scrapped and lead to reevaluation of the acquisition process, and to it being repeated.

2.5.5 Aims and Objectives of Elicitation

"SLIDE 248"

The aims of elicitation is to cover the span of the domain as accurately and fully as possible.

The objectives of elicitation is to obtain "bits and pieces" — and hopefully much more – of relevant domain knowledge within the scope of the domain being studied. We shall refer to the 'bits and pieces' of domain knowledge as domain description units.

2.5.6 Domain Description Units

"SLIDE 249"

Characterisation

Characterisation 51 (Domain Description Unit) By a domain description unit we shall mean an as far as possible well-formed sentence, something which names and describes some entity, function, event or behaviour of the domain, that is, something expressible which "makes sense", that is, which can contribute to the modelling of an entity, a function, an event or a behaviour . ∎

Handling[22]

Thus domain acquisition amounts to the laborious, painstaking process of collecting (storing) what may appear to the domain engineer as "zillions" of domain description units. In preparation for the ongoing, say concurrent domain analysis and concept formation process domain description units are provided with attributes such as name(s) (of one or more kinds of phenomena and/or concepts), kinds (entity, function, event and behaviour), source (name, etc., of stakeholder and domain engineer), and date(s) (of first acquisition and possible updates or revisions[23]).

For examples of domain acquisition unit modelling and resulting documents we refer to Appendix E, Sect. E.3 (Page 332).

[22] "SLIDE 250"

[23] We omit treatment of the necessary handling of all versions of any domain description unit.

2.6 Domain Analysis and Concept Formation "slide 251"

Given a suitable set, not necessarily what may be believed to be a reasonably complete set, of reasonably related domain description units, where, by 'related', we mean domain description units that contain overlapping (names of) entities, functions, events and behaviours, one can start analysing these domain description units.

2.6.1 Characterisations "slide 252"

First some preliminaries.

Consistency

Characterisation 52 (Consistency) By *consistency* of a set of two or more domain description units we mean that no combination of any subset of these contradicts another combination of a subset of these. ∎

Contradiction[24]

Characterisation 53 (Contradiction) By two different sets of domain description units being in *contradiction* of one another we mean that one can claim a propertyand its negation to hold in the model of the domain description units. ∎

Completeness[25]

Characterisation 54 (Relative Completeness) By *relative completeness* of a set of domain description units we mean a consistent set of domain description units which allows a meaningful modelling of what is being described such that the model does not leave something accidentally undefined. ∎

That is, we can perfectly well imagine that we leave some domain aspects purposely undefined.

Conflict[26]

Characterisation 55 (Conflict) By a *conflict* of a set of domain description units we mean an inconsistency that cannot be resolved by the domain engineer only discussing the conflicting domain description units with the stakeholders from whom the units are elicited. ∎

"slide 256"

There are three cases of conflict resolution. (i) A single stakeholder is assumed not to generate conflicts. (ii) Two or more stakeholders from the same stakeholder group should be able, together with the domain engineers, to resolve the conflict. (iii) Two or more stakeholders from different stakeholder groups may, together with the domain engineers, have to refer to their management for resolution.

2.6.2 Aims and Objectives of Domain Analysis

"SLIDE 257"

Aims of Domain Analysis

Characterisation 56 (Domain Analysis, Aims) By *domain analysis* we mean a systematic study of all domain description units, that is a "close reading and review" of these whose aim is to cover them all. ∎

Objectives of Domain Analysis[27]

Characterisation 57 (Domain Analysis, Objectives) By *domain analysis objectives* we mean a domain analysis whose objective it is to find [all] inconsistencies and [all] incompletenesses, to remove these, and to ensure a relatively scope-complete set of consistent domain description units. ∎

2.6.3 Concept Formation

"SLIDE 259"

In addition to detecting inconsistencies, conflicts and incompleteness of a set of domain description units, domain analysis also has as objective to possibly form concepts.

Characterisation 58 (Domain Concept) By a domain concept we mean a concept, an abstraction, a mental construction, which captures all essential properties and "suppresses" expression of properties deemed not essential. ∎

Aims and Objectives of Domain Concept Formation[28]

The aim of domain concept formation is to focus on similarities of domain phenomena or already defined domain concepts and, from these possibly form new, usually more generic concepts.

[24] "SLIDE 253"
[25] "SLIDE 254"
[26] "SLIDE 255"
[27] "SLIDE 258"
[28] "SLIDE 260"

The objective of domain concept formation is to arrive at simpler domain models, at generic domain models, that is, models which cover several more concrete, i.e., instantiated domains.

For examples of domain analysis and concept formation modelling and resulting documents we refer to Appendix E, Sect. E.4 (Page 336).

2.7 Domain, i.e., Business Processes "SLIDE 261"

2.7.1 Characterisation

Characterisation 59 (Business Process) By a *business process* we understand the procedurally describable aspects, of one of the (possibly many) 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 process may involve strategic, tactical or operational management and work-flow planning and decision activities; or 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. ∎

2.7.2 Business Process Description "SLIDE 262"

A business process description is usually in the form of a behaviour description which covers core entities, functions and events. Usually one describes several (more or less related) business processes

2.7.3 Aims & Objectives of Business Process Description "SLIDE 263"

Aims

The aims of describing a set of domain business processes is to cover all the "standard", i.e., all the most common as well as a reasonable number of the more special business processes of the chosen span and scope while covering most of the entities, functions and events that were identified is the full set of domain description units.

Objectives[29]

The objectives of describing a set of domain business processes is to discover domain entities, functions and events that were omitted from, i.e., are not found in the the full set of domain description units; that is, to somehow "test" and validate the domain acquisition stage.

[29] "SLIDE 264"

2.7.4 Disposition

"slide 265"

So what do we do if and when we find that the full set of domain description units and the rough-sketched domain business processes are at odds ? We obviously have to inquire with the relevant domain stakeholders. Based on their "feedback" we have to modify the full set of domain description units as well as the rough-sketched domain business processes. This is an iterative process and may involve modifying the domain analysis and concept formation findings.

• • •

For examples of business processes modelling and resulting documents we refer to Appendix E, Sect. E.5 (Page 337).

2.8 Domain Terminology

"slide 266"

2.8.1 The 'Terminology' Dogma

It is an important aspect of domain engineering to establish, use and maintain a domain terminology.

2.8.2 Characterisations

"slide 267"

Characterisation 60 (Term) By a *term* is here meant [140]: a word or phrase used in a definite or precise sense in some particular subject, as a science or art; a technical expression; by word or group of words expressing a notion or conception, or denoting an object of thought. ▪

"slide 268"

Characterisation 61 (Terminology) By *terminology* is meant [140]: the doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature. ▪

2.8.3 Term Definitions

"slide 269"

Thus a terminology is a set of definitions consisting of a "left-hand side" definiendum, usually a name, "the term", of that which is to be defined, and a "right-hand side" definiens, the expression which defines.

The definiens expressionmay either contain ground terms, that is, terms that are taken for understood, and the definiens expression is then called an atomic expression; or it contains other terms being defined in the terminology and the definiens expression is then called a composite expression.

"slide 270"

A set of term definitions form a well-formed terminology if all professional, i.e., domain-specific terms are defined, and, although some terms may be (mutually) recursively defined, these recursions do terminate by means of alternative definition choices.

2.8.4 Aims and Objectives of a Terminology "slide 271"

The aims of a domain terminology (i.e., of domain terminologisation) is to cover all the terms that are specific to the domain.

The objectives of a domain terminology (i.e., of domain terminologisation) is to ensure that all stakeholders[30], the developers and the domain description readers obtain as near, if not, the same understanding of the recorded terms.

2.8.5 How to Establish a Terminology "slide 272"

First a set of terms to be defined is selected. Then each term is defined, either atomically, or in composite manner, possibly recursively. The definition ends when all selected terms have been defined and all uses of domain-specific terms not already in the list of selected terms have been defined. "slide 273"

As can be seen from the above procedure it requires careful administration and usually ends up in a prolonged, iterated process.

When defined informally, the domain engineer may wish to use pictures, diagrams. When defined formally one may have to prove that the definitions are sound.

For examples of domain terminology modelling and resulting documents we refer to Appendix M, Sect. E.6 (Page 338). "slide 274" "slide 275"

2.9 Domain Modelling "slide 276"

2.9.1 Aims & Objectives

The **aims** of the domain modelling stage of domain engineering are to **research** the chosen domain, to find suitable **delineations** within and **structures** of that domain. The **objectives** of the domain modelling stage of domain engineering are to **develop** narrative and formal descriptions of the domain, to **analyse** those descriptions, and hence to establish a and contribute to a **theory** of that domain.

For a large scale example of domain modelling we refer to Appendices E–L.

2.9.2 Domain Facets "slide 277"

In this, a major methodology chapter of the current book, we shall start unravelling a number of principles, techniques of and a tool (RSL) for domain modelling.

Domain modelling, as we shall see, entails modelling a number of domain facets. "slide 278"

[30] Different stakeholder groups often have quite different interpretations of some terms — and these co-existing interpretations have to be reconciled.

Characterisation 62 (Domain Facet) By a **domain facet** we mean one amongst a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain. ∎

2.9.3 Describing Facets

"SLIDE 279"

These are the facets that we find "span" a domain in a pragmatically sound way: (i) intrinsics, (ii) support technology, (iii) management & organisation, (iv) rules & regulations, (v) scripts and (vi) human behaviour:
There may be other ways in which to view, that is, to understand the domain. That is, there may be other compositions of other "facets", which together also "span" the domain. The ones listed above, (i–vi), are the ones we shall pursue.

We shall cover these facets — in Chaps. F–K.

2.9.4 Domain Intrinsics

"SLIDE 280"

Characterisation 63 (Domain Intrinsics) By **domain intrinsics** we mean those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stakeholder view. ∎

By studying just the domain intrinsics we get to understand a, or rather, the essence of the domain.

If we remove any one aspect of the domain intrinsics then we jeopardise our understanding of the domain.

Construction of Model of Domain Intrinsics[31]

So the domain engineer, on the basis of analysed and possibly abstracted domain description units must construct a domain intrinsics model. The model consists, we advocate, of two complimentary parts: a narrative description and a formal description. The usual description principles and techniques apply:these are shown applied in the support example that complements this volume; we advice the reader to study that example carefully: learn by reading.

Overview of Support Example[32]

Appendix F (Pages 343–364) exemplify a domain intrinsics. It does so while also exemplifying: (i) entities, (i) functions, (i) events and (i) behaviours.

[31] "SLIDE 281"
[32] "SLIDE 282"

We advice the reader to recall the material on entities, functions, events and behaviors of Sect. 1.12 (Pages 34–42). Section F.6 exemplifies some important concepts which provide a "bridge" between events and behaviours.

Review of Support Example[33]

Entities

Section F.3 (Pages 343–348) exemplifies some intrinsic entities of an abstract transport domain.

The intrinsic net entities include those of hubs, links, nets, hub identifiers and link identifiers.[34]

We start, cf. Page 344, by narrating and formalising what we have chosen to be the most basic aspects of transportation: hubs (intersections, junctions, nodes) (Item 5) and links (segments, edges) (also Item 5) of transportation nets (Item 6). We model these as sorts, i.e., abstract types. Then we model nets as simple Cartesians of sets of hubs and sets of links. Finally an axiom (also Item 6) secures that a net has at least one link and at least two hubs.

"slide 284"

Recall, please, that what we model are phenomena in a perceived, real world. So hubs are "real" hubs; and links are "real" links; they are not representations (for example in some computer, but abstractions) of these. One can, however, introduce a notion of (unique) hub and link identifiers — and we do so in Item 7 on page 344. That is, we model (Item 8a) "observing" hubs by observing their distinct hub identifiers; and "observing" links by observing their distinct link identifiers. Distinctness is modelled by an axiom (Item 8b).

"slide 285"

And any given link is connected to a pair of specific, distinct hubs (Item 9, Page 345). Any given hub is connected to a definite number of specific, distinct links (Item 10). In our perceived reality we can be located at a hub and from there we can observe these links (Item 9(a)); and we can be located at, or on, a link and from there we can observe the connected hubs (Item 10(a)). That is, we model (Item 9(a)) "observing" hubs by observing their distinct hub identifiers (Item 9); and "observing" links by observing their distinct link identifiers (Item 10). To avoid "dangling"[35] identifiers we express two axioms: Items 11–12. To express these axioms we introduce two auxiliary functions (iohs, iols).

"slide 286"

Magic Functions on Entities: Link and hub identifiers are such attributes. Links and hubs may have any number of attributes[36]. When types are expressed as sorts, that is, as abstract types, then there is no limit to which

[33] "SLIDE 283"

[34] Later we shall introduce further intrinsic entities: vehicles, traffic, etc.

[35] A hub or a link identifiers is 'dangling' if there is no hub, respectively link of that identifier. The identifier "reference" hangs "dangling in the air".

[36] Links may, for example, have the following attributes: cadestral coordinates (say according to some Bezier curve), length, mode (whether rail, or road, or other), name (not to be confused with identifier), maintenance status, etc.

kinds of observer functions one may introduce on values of the sort type, in order to observe attributes of those values. If we had a definite, that is, a concrete type model of a type, viz.: "slide 287"

type T = A × B × C × D × E × UId

then values of type T would be expressible as (a,b,c,d,e,uid) where a:A, b:B, c:C, d:D and e:E and where uid:UId is "a" unique identifier. Now we can compare two T values t′ and t″ without involving the UId component, for example for equality: The purpose of the Appendix section on **Entity Projections** Sect. F.3.2, Pages 347–348 is to be able to compare sort values free of some attribute. "slide 288"

Some Preliminary Observations: We have illustrated a few entity types and values. Please observe that with seven such: hubs, links, nets, hub identifiers, link identifiers, pseudo-hubs and pseudo links as well as with their associated observer functions quite a lot can be said about transportation nets. "slide 289"

Exercises 2.17 on page 104 – 2.17 on page 105 — which can be solved on the basis of material presented so far — show that these simple intrinsic entities "cover a lot of ground". This leads us to formulate the following[37]:

Principle 3 ("Narrow Bridge") Search for and then select a possibly smallest set of intrinsic phenomena and concepts, typically entities and functions, based on which further phenomena can be better understood and on which further concepts can be defined. ▪

Functions [Operations][38]

General: Section F.4 exemplifies some intrinsic functions[39] of an abstract transport domain. The functions are, perhaps, somewhat "strange". You may have expected functions such asvehicles entering and leaving the net. vehicles accelerating or decelerating and stopping, or other, but we have chosen the 'Insert' and 'Remove' link operations on nets so as to stay with a minimum of phenomena and concepts and not have to introduce the concepts of vehicles, entering and leaving positions and traffic (these will soon be introduced, anyway, but now in the context of behaviours). "slide 291"

Syntax and Semantics: In introducing the Insert and Remove operations we illustrate additional concepts of abstraction and modelling, namely those of syntax and semantics, and of well-formedness of syntactical entities. Well-formedness of the semantic types of nets has already been illustrated — by the axioms governing relations between hub and links and their identifiers. Now these hubs, links and identifiers also 'occur' in the syntactical commands, out of context from the intended nets on which to perform the commanded operations. "slide 292"

[37] — as inspired by Michael A. Jackson's Principle of the 'Narrow Bridge' [123].

[38] "SLIDE 290"

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

Preliminary Observations: Some functions directly correspond to operations of the domain, for example, Insert and Remove. Other functions are introduced, by the domain engineer, in order to factor the definition of the domain operations into a manageable "size". The former are called main or sometimes semantic functions. The latter are called auxiliary functions.

Principle 4 (Syntax and Semantics) When considering a function of a domain examine whether (some of) the arguments of the function form an entity that can be considered a syntactic entity for which the function being considered is to define the semantics of the syntactic entity. ∎

"slide 293"

Technique 2 (Function Factoring) When considering the definition of functions one should seriously experiment with the introduction of auxiliary functions. Often this can be done more optimally once a number of main, or semantic, functions have been, or are being defined. Then one can better survey the need for auxiliary functions — and then which such to introduce and define. ∎

Events[40]

Section F.5 exemplifies some intrinsic events of an abstract transport domain.

The event concept, as defined (Sect. 1.12.3 on page 39), is strongly intertwined with the concept of time. One example of an event further involves a concept of position (of vehicles on the net). The two concepts, time and position, can be thought of as entities and can be claimed to be intrinsic. They were not exemplified, so far, in Sect. F.3, but will be covered in Sect. F.6.

"slide 295"

On A Concept of 'Interesting Events': The definition of events is, perhaps, too broad. It covers the examples that we have labelled as events. But it also covers examples which we, for pragmatic reasons, have no interest in singling out as events. Examples of un-interesting events are: (i) a vehicle leaving a link while entering a hub; and (ii) the increase of a bank account balance as the direct, willed result of a deposit. We would say that such state changes which occur at times that are determined by willed, oftentimes human actions (to wit: deposits and driving), as events are very explicitly willed and are hence un-interesting. The event of a bank balance exceeding, that is, "going below" the credit limit, we would, however call an interesting one.

"slide 296"

As the reader may now discern: we cannot give a precise, formal characterisation of the borderline between un-interesting and interesting events; it is, in our thinking, purely a pragmatic choice made by the domain stakeholders and domain engineers as to what constitutes an interesting event. The choices, across a domain description, as to that borderline is crucial, it is often determined by an individual stakeholder (group) view: events that are, or may be, interesting to one stakeholder group may very well be un-interesting to another stakeholder group. An abstract example could be: whether one event (usu-

"slide 297"

[40] "SLIDE 294"

ally un-interesting event) precedes or succeeds another (usually un-interesting event) could be an interesting event. The two (usually) un-interesting events could then be said to be "atomic" with respect to the "composite" interesting event.

Auxiliary Concepts[41]

Section F.6 exemplifies some auxiliary concepts of an abstract transport domain. They are: time and vehicle positions on the transport net. The time concept is not a simple one: We decompose the time concept into two: one is the concept of absolute time, the other is the concept of time intervals. The definition of (absolute) time given in Appendix Sect. F.6.1 The vehicle position concept is a simple one. "slide 299"

In Appendix D we present two axioms systems: one is a van Benthem's continuum theory of time, Sect. D.1, the other is Blizard's theory of time-space, Sect. D.2. The description of time given in Appendix Sect. F.6.1 is not a proper axiom system but, in a short way hints at the problem of a proper axiomatisation of time — such as demonstrated in Benthem's continuum theory of time, Sect. D.1

Behaviours[42]

Section F.7 exemplifies some intrinsic events and behaviours of an abstract transport domain.

We remind the reader of the definition of what constitutes a behaviour, Sect. 1.12.4: a finite set of possibly infinite length sequences of actions and events where events express synchronisation and communication between behaviours.

Two Forms of Behaviour Abstraction: In Appendix Sect. F.7 we model behaviours in two different ways: as entities that abstract functions from time into "whatever the behaviour is really about", for example, the vehicle behaviour of Page 362, and as a set of communicating sequential processes, that is, in CSP. "slide 301"

A Functional Behaviour Abstraction: The former form of abstraction, 'as entities that abstract functions from time into ...', abstracts the synchronisation and communication aspects of events, instead these are "hidden" in "whatever the behaviour is really about". First, however, we narrate: traffic is a discrete, monotonic function from a proper subset of time to positions of vehicles (modelled as maps from vehicles to positions). Now: when we say that we hide the synchronisation and communication aspects of events, we mean to say that some properties of these events only transpire indirectly from the time-wise succession of 'positions of vehicles'. That is: one has to examine the traffic

[41] "SLIDE 298"
[42] "SLIDE 300"

abstraction to identify such events as: vehicle crashes (two or more vehicles occupying same positions), vehicles leaving the traffic at some velocity (that is, driving, at some speed, off the links), etc.

"slide 302"

Well-formedness of Functional Abstractions: The author of this book is of the unabashed opinion that the abstract model of traffic: $\mathsf{TF}' = \mathsf{PSoTime} \xrightarrow[m]{} (\mathsf{N} \times (\mathsf{V} \xrightarrow[m]{} \mathsf{P}))$ is an elegant model ! But that perceived elegance comes at at least one cost: The TF type admits "traffics" that are not intended; hence the TF type must be subtyped: **type** $\mathsf{TF} = \{|\ \mathsf{tf}{:}\mathsf{TF}' \bullet \mathsf{wf\_TF}\{\mathsf{tf}\}\ |\}$. and The Time type must be constrained with respect to a dense type of enumerable times T: $\mathsf{Time} = \{|\mathsf{tset}{:}\mathsf{T\text{-}set} \bullet \mathsf{wf\_PSoTime}(\mathsf{tset})|\}$ where wf_TF and wf_PSoTime are defined on Appendix Sect. F.7.1 on page 361 etcetera and Sect. F.6.1 on page 358

"slide 303"

A [CSP] Process-oriented Behaviour Abstraction: The latter form of abstraction: 'as a set of communicating sequential processes', is shown in Sect. F.7.2.

Here we abstract behaviour (a finite set of possibly infinite length sequences of actions and events) by the set of traces (of actions and input/output events) of a finite set of — in this case CSP — processes[43]. The abstraction from domain behaviours to description language processes is thus rather "direct", that is, hardly an abstraction !

Discussion of Domain Intrinsics[44]

TO BE WRITTEN

2.9.5 Support Technologies

"SLIDE 305"

Characterisation 64 (Support Technologies) By **domain support technologies** we mean ways and means of concretesing certain observed (abstract or concrete) phenomena or certain conceived concepts in terms of (possibly combinations of) human work, mechanical, hydro mechanical, thermo-mechanical, pneumatic, aero-mechanical, electro-mechanical, electrical, electronic, telecommunication, photo/opto-electric, chemical, etc. (possibly computerised) sensor, actuator tools. ∎

Technology as an Embodiment of Laws of Physics[45]

For examples of domain support technology modelling and resulting documents we refer to Appendix G.

By technology, we here mean "gadgets" (instruments, machines, artifacts) which somehow or other embody, exploit, rely on, etc., laws of physics (including chemistry).

[43] By the trace of a CSP process we understand, as an example, a sequence of labels of CSP actions and CSP input/output events. The labels name (and describe) the actions (and, if need be, also the state in which the action is interpreted) and the input/output channel and the communicated message value.

[44] "SLIDE 304"

[45] "SLIDE 306"

From Abstract Domain States to Concrete Technology States

Usually an intrinsic domain phenomenon or concept embody an abstract notion of state. The essence of a support technology is then to render such an abstract notion of state more concrete.

Intrinsics versus Other Facets[46]

Take as "other facets" those of supporting technologies. The nature of intrinsics in the light of a supporting technology is to force the domain engineer to think abstractly in order to capture an essence of a phenomenon or concept of the domain, not by its "implementing" support technologies, i.e., the how, but by what that domain phenomenon or concept really means, semantically.

This point is illustrated in the examples of an intrinsic concepts of states versus the examples of a corresponding support technology concepts of states of the three examples of Appendix G:

- (intrinsics) Sect. G.1.1 on page 367 versus (support technologies) Sect. G.1.2 on page 370;
- (intrinsics) Sect. G.2 on page 372 versus (support technologies) Sect. G.2.2 on page 373;
- (intrinsics) Sect. G.3 on page 381 versus (support technologies) Sect. G.3.3 on page 382;

The Three Support Examples[47]

In review, we model, in Appendix G, three sets of supporting technologies: road intersection semaphore (green, yellow, red) signalling, road-rail level crossing (a more complicated variant of the road intersection semaphore example) and rail switching (a very simplified treatment).

Transport Net Signalling

The road intersection semaphore is rather conventional: instead of letting car drivers just drive as they please, a road intersection semaphore shows which crossings of the road intersection are advised by means of green/yellow/red lamps. "slide 309"

The example does not model clearly inconsistent signalling such as green signals in all directions (rather than unlit lamps), or such as too quick lamp changes from red/yellow to green, for example along a pair of opposite directions in a four street intersection while the lamps in the cross direction are still green/yellow, etcetera. We leave it to the reader to describe such cases.

The example also does not model failure of semaphore equipment. The road-rail level crossing hints at such modelling.

[46] "SLIDE 307"

[47] "SLIDE 308"

Road-Rail Level Crossing[48]

This example examines a "tiny", recurrent part of rail nets. After some preliminaries, Sects. G.2.1–G.2.2, on the intrinsic and the concrete concepts of road-rail level intersection (i.e., hub) states, and an overview, Sect. G.2.3, follows five technical parts: (i) 'Function and Safety' Sect. G.2.4, (ii) 'The Road Traffic Domain' Sect. G.2.5, (iii) 'The Train Traffic Domain' Sect. G.2.6, (iv) 'The Device Domain' Sect. G.2.7 and (v) 'The Software Design' Sect. G.2.8.

"slide 311"

(i) The 'Function and Safety' Sect. G.2.4 part is general for basically any safety critical, real-time embedded system development, and, as such, introduces a number of concepts by means of which we express, not only the basic function and safety properties, but also those of the following four parts (ii–v). Part (i) also illustrates also the first use of the Duration Calculus. We advice the reader, at this point, to study Sect. G.2.4 carefully (Pages 374–378).

"slide 312"

(ii) The 'Road Traffic Domain' Sect. G.2.5 part is very brief. Part (ii) separates out the "minimum" assumptions about the road traffic with respect to the road-rail intersection. The developer must express these in order for the entire development to make sense, that is, proven correct. We advice the reader to study Sect. G.2.5 carefully (Pages 378–378).

"slide 313"

(iii) The 'Train Traffic Domain' Sect. G.2.6 part is also brief. Again the part separates out the "minimum" assumptions about the train traffic, with respect to the road-rail intersection. The developer must express these in order for the entire development to make sense, that is, proven correct. We advice the reader to study Sect. G.2.6 carefully (Pages 379–379).

"slide 314"

(iv) The 'Device Domain' Sect. G.2.7 part, is also very brief Part (iv) again expresses "minimum" assumptions about the technological devices that govern the monitoring and control of the road-rail intersection. The developer must formulate these in order for the entire development to make sense, that is, proven correct. We advice the reader to study Sect. G.2.7 carefully (Pages 379–380).

"slide 315"

(v) The 'Software Design' Sect. G.2.8 part, finally, is also brief. We advice the reader to study Sect. G.2.8 carefully (Pages 380–381).

Rail Switching[49]

The last example of a support technology is that of an ordinary rail switch, cf. Sect. G.3 Pages 381–384. First the example clarifies the issue of intrinsic rail switches in contrast to a variety of human and mechanical and electro-mechanical support technologies. Then the example focuses, briefly, on the "statistics" of such electro-mechanically supported switches.

"slide 317"

The example does not complete "the line of thought" introduced by the probability state diagram of Fig. G.4 on page 383. A completion would then go on to show how the probabilities (provided by the supplier of such rail

[48] "SLIDE 310"

[49] "SLIDE 316"

switches) would be part of a Markov modelling of the state switching behaviour. That Markov model models an implementation in which repeated signalling (with assumed sensing of whether a state change has indeed taken place) serves to "lower" the probability of an eventual erroneous state — usually to some such number as 10^{-8}.

Discussion of Support Technologies[50]

The modelling of support technologies must face several aspects and deploy several modelling tools. In our modelling of support technologies we exemplify just three aspects: "refinement" of simple support technology state switching to compositions of such atomic state switching signals; consideration of timing aspects; and consideration of probabilities of state switching signals not achieving their desired effect. Two related comments are in order: (i) actual support technologies are orders of magnitude more complex than the ones illustrated here; and (ii) their proper domain description becomes relatively large development projects "in-and-by-themselves"! "slide 319" "slide 320" "slide 321"

2.9.6 Management and Organisation "SLIDE 322"

Management

Management is an elusive term. Business schools and private consultancy firms excel in offering degrees and 2–3 day courses in 'management'. In the mind of your author most of what is being taught — and even researched — is a lot of "hot air". Well, the problem here, is, of course, that your author was educated at a science & technology university[51]. In the following we shall repeat some of this 'hot air '. And after that we shall speculate on how to properly describe the outlined ("cold air") management concepts. "slide 323"

Characterisation 65 (Domain Management) By **domain management** we mean people (i) who determine, formulate and thus set standards (cf. rules and regulations, a later lecture topic) concerning strategic, tactical and operational decisions; (ii) who ensure that these decisions are passed on to (lower) levels of management, and to "floor" staff; (iii) who make sure that such orders, as they were, are indeed carried out; (iv) who handle undesirable deviations in the carrying out of these orders cum decisions; and (v) who "backstop" complaints from lower management levels and from floor staff. ∎

[50] "SLIDE 318"

[51] — which, alas, now also offers such 'management' degree courses !

Management Issues[52]

Management in simple terms means the act of getting people together to accomplish desired goals. Management comprises (vi) planning, (vii) organizing, (viii) resourcing, (ix) leading or directing, and (x) controlling an organization (a group of one or more people or entities) or effort for the purpose of accomplishing a goal. Resourcing encompasses the (xi) deployment and manipulation of human resources, (xii) financial resources, (xiii) technological resources, and (xiv) natural resources

Basic Functions of Management[53]

These are normally seen as management issues:

Planning: (xv) deciding what needs to happen in the future (today, next week, next month, next year, over the next 5 years, etc.) (xvi) and generating plans for action. Organizing: (xvii) making optimum use of the resources (xix) required to enable the successful carrying out of plans. Leading/Motivating: (xx) exhibiting skills in these areas (xxi) for getting others to play an effective part in achieving plans. Controlling: (xxii) monitoring – (xxiii) checking progress against plans, (xxiv) which may need modification based on feedback.

Formation of Business Policy[54]

(xxvi) The **mission** of a business seems to be its most obvious purpose – which may be, for example, to make soap. (xxvii) The **vision** of a business is seen as reflecting its aspirations and specifies its intended direction or future destination. (xxviii) The **objectives** of a business refers to the ends or activity at which a certain task is aimed[55]. The business **policy** is a guide that stipulates (xix) rules, regulations and objectives, (xxx) and may be used in the managers' decision-making. (xxxi) It must be flexible and easily interpreted and understood by all employees. The business **strategy** refers to (xxxii) the coordinated plan of action that it is going to take, (xxxiii) as well as the resources that it will use, to realize its vision and long-term objectives. (xxxiv) It is a guideline to managers, stipulating how they ought to allocate and utilize the factors of production to the business's advantage. (xxxv) Initially, it could help the managers decide on what type of business they want to form.

"slide 327"

[52] "SLIDE 324"
[53] "SLIDE 325"
[54] "SLIDE 326"
[55] Pls. note that, in this book, we otherwise make a distinction between aims and objectives: Aims is what we plan to do; objectives are what we expect to happen if we fulfill the aims.

Implementation of Policies and Strategies[56]

(xxxvi) All policies and strategies are normally discussed with managerial personnel and staff. (xxxvii) Managers usually understand where and how they can implement their policies and strategies. (xxxviii) A plan of action is normally devised for the entire company as well as for each department. (xxxix) Policies and strategies are normally reviewed regularly. (xxxvii) Contingency plans are normally devised in case the environment changes. (xl) Assessments of progress are normally and regularly carried out by top-level managers. (xli) A good environment is seen as required within the business.

Development of Policies and Strategies[57]

(xlii) The missions, objectives, strengths and weaknesses of each department or normally analysed to determine their rôles in achieving the business mission. (xliii) Forecasting develops a picture of the business's future environment. (xliv) Planning unit are often created to ensure that all plans are consistent and that policies and strategies are aimed at achieving the same mission and objectives. (xlv) Contingency plans are developed — just in case ! (xlvi) Policies are normally discussed with all managerial personnel and staff that is required in the execution of any departmental policy.

Management Levels[58]

A careful analysis has to be made by the domain engineer of how management is structured in the domain being described. One view, but not necessarily the most adequate view for a given domain is that management can be seen as composed from the board of directors (representing owners, private or public, or both), the senior level or strategic (or top, upper or executive) management, the mid level or tactical management, the low level or operational management, and supervisors and team leaders. Other views, other "management theories" may apply. We shall briefly pursue the above view.

Resources[59]

Management is about resources. A resource is any physical or virtual entity of limited availability such as, for example, time and (office, factory, etc.) space, people (staff, consultants, etc.), equipment (tools, machines, computers, etc.), capital (cash, goodwill, stocks, etc.), etcetera.

Resources have to be managed allocated (to [factory, sales, etc.] processes, projects, etc.), and scheduled (to time slots).

[56] "SLIDE 328"
[57] "SLIDE 329"
[58] "SLIDE 330"
[59] "SLIDE 331"

Resource Conversion[60]

Resources can be traded for other resources: capital funds can be spent on acquiring space, staff and equipment, services and products can be traded for other such or for monies, etc.

The decisions as to who schedules, allocates and converts resources are made by strategic and tactical management. Operational management transforms abstract, general schedules and allocations into concrete, specific such.

Strategic Management[61]

A strategy is a long term plan of action designed to achieve a particular goal. Strategy is differentiated from tactics or immediate actions with resources at hand by its nature of being extensively premeditated, and often practically rehearsed. Strategies are used to make business problems easier to understand and solve. Strategic management deals with conversion of long term resources involving financial issues and with long term scheduling issues.

"slide 334"

Among examples of strategic management issues (in supply chain management) we find: (xlvii) strategic network optimization, including the number, location, and size of warehouses, distribution centers and facilities; (xlviii) strategic partnership with suppliers, distributors, and customers, creating communication channels for critical information and operational improvements such as cross docking, direct shipping, and third-party logistics; (xlix) product design coordination, so that new and existing products can be optimally integrated into the supply chain, load management; (l) information technology infrastructure, to support supply chain operations; (li) where-to-make and what-to-make-or-buy decisions; and (lii) aligning overall organizational strategy with supply strategy. The problem, in domain modelling, is to find suitable abstractions of these mundane activities.

"slide 335"

Strategic management (liii) requires knowledge of management rôles and skills; (liv) have to be aware of external factors such as markets; (lv) decisions are generally of a long-term nature; (lvi) decision are made using analytic, directive, conceptual and/or behavioral/participative processes; (lvii) are responsible for strategic decisions; (lviii) have to chalk out the plan and see that plan may be effective in the future; and (lix) is executive in nature.

Tactical Management[62]

Tactical management deals with shorter term issues than strategic management, but longer term issues than operational management. Tactical management deals with allocation and short term scheduling.

"slide 337"

Among examples of tactical management issues (in supply chain management) we find: (lx) sourcing contracts and other purchasing decisions; (lxi)

[60] "SLIDE 332"
[61] "SLIDE 333"
[62] "SLIDE 336"

production decisions, including contracting, locations, scheduling, and planning process definition; (lxii) inventory decisions, including quantity, location, and quality of inventory; (lxiii) transportation strategy, including frequency, routes, and contracting; (lxiv) benchmarking of all operations against competitors and implementation of best practices throughout the enterprise; (lxv) milestone payments; and (lxvi) focus on customer demand. The problem, in domain modelling, is to find suitable abstractions of these mundane activities.

Operational Management[63]

Operational management deals with day-to-day and week-to-week issues where tactical management deals with month-to-month and quarter-to-quarter issues and strategic management deals with year-to-year and longer term issues. (Operational management is not to be confused with the concept of operational research and operational analysis which deals with optimising resource usage (allocation and scheduling). "slide 339"

Among examples of operational management issues (in supply chain management) we find: (lxviii) daily production and distribution planning, including all nodes in the supply chain; (lxix) production scheduling for each manufacturing facility in the supply chain (minute by minute); (lxx) demand planning and forecasting, coordinating the demand forecast of all customers and sharing the forecast with all suppliers; (lxxi) sourcing planning, including current inventory and forecast demand, in collaboration with all suppliers; (lxxii) inbound operations, including transportation from suppliers and receiving inventory; (lxxiii) production operations, including the consumption "slide 340" of materials and flow of finished goods; (lxxiv) outbound operations, including all fulfillment activities and transportation to customers; (lxxv) order promising, accounting for all constraints in the supply chain, including all suppliers, manufacturing facilities, distribution centers, and other customers. The problem, in domain modelling, is to find suitable abstractions of these mundane activities.

Supervisors and Team Leaders[64]

We make here a distinction between managers, "on one side", and supervisors and team leaders, "on the other side". The distinction is based on managers being able to make own decisions without necessarily having to confer or discuss these beforehand or to report these afterwards, while supervisors and team leaders normally are not expected to make own decisions: if they have to make decisions then such are considered to be of "urgency", must normally be approved of beforehand, or, at the very least, reported on afterwards. "slide 342"

Supervisors basically monitor that work processes are carried out as planned and report other than minor discrepancies. Team leaders coordinate

[63] "SLIDE 338"

[64] "SLIDE 341"

work in a group ("the team") while participating in that work themselves; additionally they are also supervisors.

Description of 'Management'[65]

On the last several pages (73–78) we have outlined conventional issues of management.

The problems confronting us now are: Which aspects of domain management are we to describe ? How are we describe, especially formally, the chosen issues ?

The reason why these two "leading questions" questions are posed is that the management issues mentioned on pages 73–78 are generally "too lofty", "too woolly", that is, are more about "feelings" than about "hard, observable facts".

"slide 344"

We, for example, consider the following issues for "too lofty", "too woolly": Item (xix) Page 74: "to enable the successful ..." is problematic; Item (xx) Page 74: how to check that managers "exhibit these skills" ?; Item (xxi) Page 74: "play an effective part" is problematic; Item (xxvii) Page 74: how to check that vision is being or is achieved ?; Item (xxviii) Page 74: the objectives must, in order to be objectively checked, be spelled out in measurable details; Item (xxxi) Page 74: how to check "flexible" and "easily"; Item (xxxiii) Page 74: how to check that the deployed resources are those that contribute to "achieving vision and long term objectives; Item (xxxiv) Page 74: "guideline", "factors of production" and "advantage" cannot really be measured; Item (xxxv) Page 74: "what type of business they want to form" is too indeterminate; Item (xxxvi) Page 75: how to describe (and eventually check) "are normally or must be discussed" other than "just check" without making sure that managerial personnel and staff have really understood the issues and will indeed follow policies and strategies; Item (xxxvii) Page 75: how does one describe "managers must, or usually understand where and how" ?; Item (xxxix) Page 75: in what does a review actually consists ?; Item (xli) Page 75: how does one objectively describe "a good environment" ?; Item (xlii) Page 75: how does one objectively describe that which is being "analysed", the "analysis" and the "determination" processes ?; Item (xliii) Page 75: how is the "development" and a "picture" objectively described ?; etcetera.

"slide 345"

As we see from the above "quick" analysis the problems hinge on our [in]ability to formally, let alone informally describe many management issues. In a sense that is acceptable in as much as 'management' is clearly accepted as a non-mechanisable process, one that requires subjective evaluations: "feelings", "hunches", and one that requires informal contacts with other managerial personnel and staff.

"slide 346"

But still we are left with the problems: Which aspects of domain management are we to describe ? How are we describe, especially formally, the chosen issues ?

[65] "SLIDE 343"

Our simplifying and hence simple answer is: the domain engineer shall describe what is objectively observable or concepts that are precisely defined in terms of objectively observable phenomena and concepts defined from these and such defined concepts.

This makes the domain description task a reasonable one, one that can be objectively validated and one where domain description evaluators can objectively judge whether (projected) requirements involving these descriptions may be feasible and satisfactory.

Review of Support Examples[66]

There are three examples (i) a grossly simplifying abstraction: *the enterprise function*, which focuses on the abstract interplay between management groups, workers, etc.; and the formal model is expressed in a recursive function style; (ii) a grossly simplifying abstraction *the enterprise processes*, which focuses on the sequential, non-deterministic processes with input/output messages that communicate between management groups, workers, etc.; the formal model is expressed in the CSP style. "slide 348"

The Enterprise Function: The enterprise function is narrated in Sect. H.1.1, and formalised in Sect. H.1.2 on page 388; the formalisation is explained and commented upon on Pages 388–390 (Sect. H.1.3). Here we shall just briefly discuss meta-issues of domain description, modelling and abstraction. "slide 349"

The description is grossly 'abstracted': it leaves out any modelling of what distinguishes top-level, executive, strategic management from mid-level, tactic management, and these from "low-level" operations management, and all of these from supervisors, team leaders and workers. Emphasis has been put sôlely on abstractions of their intercommunication in order to achieve a "next step" state. "slide 350"

The formalisation of enterprise is, formally speaking, doubtful. The semantics of the formal specification language, RSL, does not allow such recursions, or rather, put far too severe restrictions on the state space Σ, for the definition to be of even pragmatic interest. Thus the definition is really a fake: at most it hints at what goes on, such as outlined on Pages 388–390 (Sect. H.1.3). Why is the definition a fake? Or rather: Why do we show this "definition"? "slide 351"

In order for a recursive function definition, enterprise, (as here over states Σ) to make sense the type Σ must satisfy some ordering properties and so must the component types whose values are involved in any of the auxiliary functions invoked by enterprise. Since we have not specified any of these types we take the position that function definition, enterprise, is just a pseudo function. It is indicative of "what is going on", and that is why we bring it! "slide 352"

The Enterprise Processes: The enterprise processes are narrated and formalised, alternatively, in Sect. H.2 on Pages 390–398, Here we shall just briefly discuss meta-issues of domain description, modelling and abstraction. "slide 353"

[66] "SLIDE 347"

"slide 354"

TO BE WRITTEN

TO BE WRITTEN

Organisation[67]

Characterisation 66 (Domain Organisation) By **domain organisation** we mean the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels; and hence the "lines of command": who does what and who reports to whom — administratively and functionally.

■

"slide 356"

"slide 357"

TO BE WRITTEN

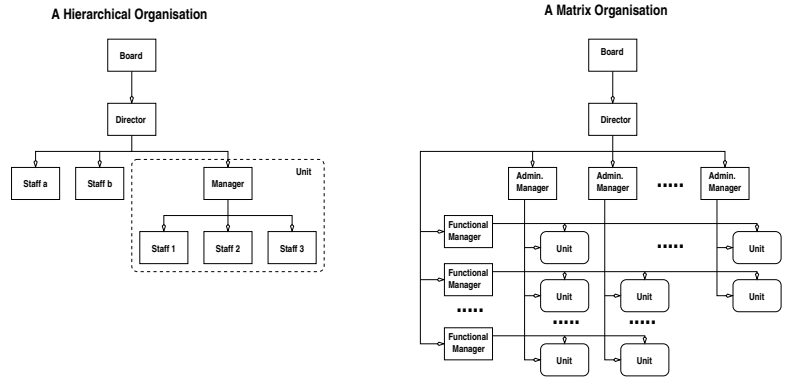


Fig. 2.1. Two organisational structures

For examples of domain management and organisation modelling and resulting documents we refer to Appendix H.

2.9.7 Rules and Regulations

"slide 358"

Human stakeholders act in the domain, whether clients, workers, managers, suppliers, regulatory authorities, or other. Their actions are guided and constrained by rules and regulations. These are sometimes implicit, that is, not "written down". But we can talk about rules and regulations as if they were explicitly formulated.

For examples of narratives of domain rules and regulations we refer to appendix Examples 28–29 (Page 407).

"slide 359"

[67] "slide 355"

The main difference between rules and regulations is that rules express properties that must hold and regulations express state changes that must be effected if rules are observed broken.

Rules and regulations are directed not only at human behaviour but also at expected behaviours of support technologies.

Rules and regulations are formulated by enterprise staff, management or workers, and/or by business and industry associations, for example in the form of binding or guiding national, regional or international standards[68], and/or by public regulatory agencies.

Domain Rules[69]

For examples of narratives of domain rules we refer to appendix Examples 28–29 (Page 407).

Characterisation 67 (Domain Rule) By a **domain rule** we mean some text which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their functions. ▪

"slide 361"

Usually the rule text, when written down, appears in some, not necessarily public documents. It is not our intention to formalise these rule texts, but to formally define the crucial predicates and, if not already formalised, then also the domain entities over which the predicate ranges.

Domain Regulations[70]

For examples of narratives of domain regulations we refer to appendix Examples 28–29 (Page 407).

Characterisation 68 (Domain Regulation) By a **domain regulation** we mean some text which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention. ▪

"slide 363"

Usually the regulation text, when written down, appears in some, not necessarily public documents. It is not our intention to formalise these rule texts, but to formally define the crucial functions and, if not already formalised, then also the domain entities over which these functions range.

[68] Viz.: ISO (International Organisation for Standardisation, www.iso.org/iso/-home.htm), CENELEC (European Committee for Electrotechnical Standardization, www.cenelec.eu/Cenelec/Homepage.htm), etc.

[69] "SLIDE 360"

[70] "SLIDE 362"

Formalisation of the Rules and Regulations Concepts

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following:

Rules, as already mentioned, express predicates, and regulations express state changes. In the following we shall review a semantics of rules and regulations.

"slide 364"

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

"slide 365"

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

"slide 366"

type
  Stimulus, Rule, Θ
  STI = $\Theta \to \Theta$
  RUL = $(\Theta \times \Theta) \to$ **Bool**
value
  meaning: Stimulus → STI
  meaning: Rule → RUL

  valid: Stimulus × Rule → Θ → **Bool**
  valid(sy_sti,sy_rul)(θ) ≡ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

  valid: Stimulus × RUL → Θ → **Bool**
  valid(sy_sti,se_rul)(θ) ≡ se_rul(θ,(meaning(sy_sti))(θ))

"slide 367"

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

"slide 368"

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

"slide 369"

type

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

"slide 370"

The idea is now the following: Any action of the system, i.e., the application of any stimulus, may be an action in accordance with the rules, or it may not. Rules therefore express whether stimuli are valid or not in the current configuration. And regulations therefore express whether they should be applied, and, if so, with what effort. "slide 371"

More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where θ' = (meaning(sy_sti))(θ). Let θ' violate the rule, ∼valid(sy_sti,sy_rul)(θ), then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,(meaning(sy_sti))(θ)), then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ), to remedy the situation. "slide 372"

```
axiom
  ∀ (sy_rul,sy_reg):Rul_and_Regs •
    let se_rul = meaning(sy_rul),
        (pre_reg,act_reg) = meaning(sy_reg) in
    ∀ sy_sti:Stimulus, θ:Θ •
      ∼valid(sy_sti,se_rul)(θ)
        ⇒ pre_reg(θ,(meaning(sy_sti))(θ))
          ⇒ ∃ nθ:Θ • act_reg(θ)=nθ ∧ se_rul(θ,nθ)
    end
```

"slide 373"

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

On Modelling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into "the state", functions, events, and behaviours. Thus the full spectrum of modelling techniques and notations may be needed. "slide 374" Since rules usually express properties one often uses some combination of axioms and well-formedness predicates. Properties sometimes include temporality and hence temporal notations (like Duration Calculus or Temporal Logic

of Actions) are used. And since regulations usually express state (restoration) changes one often uses state changing notations (such as found in B, RSL, VDM-SL, and Z). In some cases it may be relevant to model using some constraint satisfaction notation [12] or some Fuzzy Logic notations [219].

"slide 375"
"slide 376"

2.9.8 Scripts

"SLIDE 377"

For examples of narratives of domain scripts we refer to appendix Examples 30–32 (Pages 415–418).

Characterisation 69 (Domain Script) By a **domain script** we mean the structured wording of a rule or a regulation that has legally binding power, that is, which may be contested in a court of law. ∎

Analysis of Examples[71]

1 **Bus [Train] Timetables (Schedules):** Page 415
The bus/train timetable is informally sketched. Sect. J.2 will elaborate, and formalise, this timetable example. In addition that section will relate timetables to the underlying net of to the resulting and possible traffics. A timetable script thus can be given several pragmatics: (i-ii) a, perhaps not exactly legally binding, contract between the bus/train operator and the passengers, as well as a contract between the bus/train operator and the public authorities which may be financially supporting community commuting; (iii) a particular timetable (considered as syntax) semantically denotes a possibly infinite set of bus/train traffics, each of which satisfies the timetable, i.e., runs to schedule; and (iv) a script, to be followed by the drivers/train engine men, guiding these in the bus/train journey (to speed up or slow down, etc.).

"slide 379"

2 **Aircraft Flight Simulator Script:** Pages 415–416
The example script is from a specific aircraft simulator demo. It has been abstracted a bit from the real case script. You may think of the example script being partly "programmed" into the flight simulator which is a reactive system awaiting pilot trainee actions and reactions. As you note, it is quite detailed. It mentions many phenomena and concepts of aircraft flights: entities (simple as well as behavioural), operations, events, and itself prescribes a behaviour. You may additionally think of the example script as also (in addition to the flight simulator hardware and software) "scripting" the pilot trainee. Thus a specific script, for example, denotes an infinity of actual behaviours of pilot trainees working in conjunction with flight simulators.

"slide 380"

3 **Bill of Lading:** Pages 417–418
The bill of lading is also a script, but it is quite different from the previous two examples. It only very, very loosely hints at transport behaviours.

[71] "SLIDE 378"

Whereas it certainly puts some constraints on freight transport. The bill of lading script is a legal instrument which entitles the consignee to receive the freight at the destination harbour; stipulates, in the closing "conditions" item, legal protection of the two parties to the contract; etcetera.

Licenses[72]

License:

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 [209]

"slide 382"

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. "slide 383"

We shall consider three 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, and (iii) documents related to public government. "slide 384"

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; and (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. "slide 385"

In this section we shall rough-sketch-describe pragmatic aspects of the three domains of (1) production, distribution and consumption of artistic works, (2) the hospitalisation of patient, i.e., hospital health care and (3) the handling of law-based document in public government. The emphasis is on the pragmatics of the terms, i.e., the language used in these three domains.

[72] "SLIDE 381"

The Performing Arts: Producers and Consumers[73]

The intrinsic entities of the performing arts are the artistic works: drama or opera performances, music performances, readings of poems, short stories, novels, or jokes, movies, documentaries, newsreels, etc. We shall limit our span to the scope of electronic renditions of these artistic works: videos, CDs or other. In this paper we shall not touch upon the technical issues of "downloading"(whether "streaming" or copying, or other).

Operations on Digital Works[74]

For a consumer to be able to enjoy these works that consumer must (normally first) usually "buy a ticket" to their performances. The consumer, i.e., the theatre, opera, concert, etc., "goer" (usually) cannot copy the performance (e.g., "tape it"), let alone edit such copies of performances. In the context of electronic, i.e., digital renditions of these performances the above "cannots" take on a new meaning. The consumer may copy digital recordings, may edit these, and may further pass on such copies or editions to others. To do so, while protecting the rights of the producers (owners, performers), the consumer requests permission to have the digital works transferred ("downloaded") from the owner/producer to the consumer, so that the consumer can render ("play") these works on own rendering devices (CD, DVD, etc., players), possibly can copy all or parts of them, then possibly can edit all or parts of the copies, and, finally, possibly can further license these "edited" versions to other consumers subject to payments to "original" licensor.

"slide 388"

License Agreement and Obligation[75]

To be able to obtain these permissions the user agrees with the wording of some license and pays for the rights to operate on the digital works.

Two Assumptions[76]

Two, related assumptions underlie the pragmatics of the electronics of the artistic works. The first assumption is that the format, the electronic representation of the artistic works is proprietary, that is, that the producer still owns that format. Either the format is publicly known or it is not, that is, it is somehow "secret". In either case we "derive" the second assumption (from the fulfilment of the first). The second assumption is that the consumer is not allowed to, or cannot operate[77] on the works by own means (software, machines). The second assumption implies that acceptance of a license results

"slide 391"

[73] "SLIDE 386"
[74] "SLIDE 387"
[75] "SLIDE 389"
[76] "SLIDE 390"
[77] render, copy and edit

in the consumer receiving software that supports the consumer in performing all operations on licensed works, their copies and edited versions: rendering, copying, editing and sub-licensing.

Protection of the Artistic Electronic Works[78]

The issue now is: how to protect the intellectual property (i.e., artistic) and financial (exploitation) rights of the owners of the possibly rendered, copied and edited works, both when, and when not further distributed.

A License Language[79]

type
0. Ln, Nm, W, S, V
1. L = Ln × Lic
2. Lic == mkLic(licensor:Nm,licensee:Nm,work:W,cmds:Cmd-**set**)
3. Cmd == Rndr | Copy | Edit | RdMe | SuLi
4. Rndr = mkRndr(vw:(V|″work″),sl:S*)
5. Copy = mkCopy(fvw:(V|″work″),sl:S*,tv:V)
6. Edit = mkEdit(fvw:(V|″work″),sl:S*,tv:V)
7. RdMe = ″readme″
8. SuLi = mkSuLi(cs:Cmd-**set**,work:V)

"slide 394"

(0.) Licenses are given names, ln:Ln, so are actors (owners, licensors, and users, licensees), nn:Nm. By w:W we mean a (net) reference to (a name of) the downloaded possibly segmented artistic work being licensed, where segments are named (s:S), that is, s:S is a selector to either a segment of a downloaded work or to a segment of a copied and or and edited work. "slide 395"

(1.) Every license (lic:Lic) has a unique name (ln:Ln).

(2.) A license (lic:Lic) contains four parts: the name of the licensor, the name of the licensee, a reference to (the name of) the work, a set of commands (that may be permitted to be performed on the work). "slide 396"

(3.) A command is either a render, a copy or an edit or a readme command, or a sub-licensing (sub-license) command. "slide 397"

(4.–6.) The render, copy and edit commands are each "decorated" with an ordered list of selectors (i.e., selector names) and a (work) variable name. The license command

copy ⟨s1,s2,s7⟩ v

means that the licensed work, ω, may have its sections s_1, s_2 and s_7 copied, in that sequence, into a new variable named v, Other copy commands may specify other sequences. Similarly for render and edit commands. "slide 398"

[78] "SLIDE 392"

[79] "SLIDE 393"

(7.) The "readme" license command, in a license, ln, referring, by means of w, to work ω, somehow displays a graphical/textual "image" of, that is, information about ω. We do not here bother to detail what kind of information may be so displayed. But you may think of the following display information names of artistic work,artists, authors, etc., names and details about licensed commands, a table of fees for performing respective licensed commands, etcetera.

"slide 399"

(8.) The license command

schema: **license** cmd1,cmd2,...,cmdn **on work** v
formal: mkSuLi({cmd1,cmd2,...,cmdn},v)

means that the licensee is allowed to act as a licensor, to name sub-licensees (that is, licensees) freely, to select only a (possibly full) subset of the sub-licensed commands (that are listed) for the sub-licensee to enjoy. The license need thus not mention the name(s) of the possible sub-licensees. But one could design a license language, i.e., modify the present one to reflect such constraints. The license also do not mention the payment fee component. As we shall see under licensor actions such a function will eventually be inserted.

"slide 400"

A license licenses the licensee to render, copy, edit and license (possibly the results of editing) any selection of downloaded works. In any order — but see below — and any number of times. For every time any of these operations take place payment according to the payment function occurs (that can be inspected by means of the **read license** command). The user can render the downloaded work and can render copies of the work as well as edited versions of these. Edited versions are given own names. Editing is always of copied versions. Copying is either of downloaded or of copied or edited versions. This does not transpire from the license syntax but is expressed by the licensee, see below, and can be checked and duly paid for according to the payment function.

"slide 401"

The payment function is considered a partial function of the selections of the work being licensed.

Please recall that licensed works are proprietary. Either the work format is known, or it is not supposed to be known, In any case, the rendering, editing, copying and the license-"assembling" (see next section) functions are part of the license and the licensed work and are also assumed to be proprietary. Thus the licensee is not allowed to and may not be able to use own software for rendering, editing, copying and license assemblage.

"slide 402"

Licenses specify sets of permitted actions. Licenses do not normally mandate specific sequences of actions. Of course, the licensee, assumed to be an un-cloned human, can only perform one action at a time. So licensed actions are carried out sequentially. The order is not prescribed, but is decided upon by the licensee. Of course, some actions must precede other actions. Licensees must copy before they can edit, and they usually must edit some copied work before they can sub-license it. But the latter is strictly speaking not necessary.

"slide 403"

type

5. V
6. Act = Ln × (Rndr|Copy|Edit|License)
7. Rndr == mkR(sel:S*,wrk:(W|V))
8. Copy == mkC(sel:S*,wrk:(W|V),into:V)
9. Edit == mkE(wrks:V*,into:V)
10. License == mkL(ln:Ln,licensee:Nm,wrk:V,cmds:Cmd-**set**,fees:PF)

"slide 404"

(5.) By V we mean the name of a variable in the users own storage into which downloaded works can be copied (now becoming a local work. The variables are not declared. They become defined when the licensee names them in a copy command. They can be overwritten. No type system is suggested. "slide 405"

(6.) Every action of a licensee is tagged by the name of a relevant license. If the action is not authorised by the named license then it is rejected. Render and copy actions mention a specific sequence of selectors. If this sequence is not an allowed (a licensed) one, then the action is rejected. (Notice that the license may authorise a specific action, *a* with different sets of sequences of selectors — thus allowing for a variety of possibilities as well as constraints.) "slide 406"

(7.) The licensee, having now received a license, can **render** selections of the licensed work, or of copied and/or edited versions of the licensed work. No reference is made to the payment function. When rendering the semantics is that this function is invoked and duly applied. That is, render payments are automatically made: subtracted from the licensees account and forwarded to the licensor. "slide 407"

(8.) The licensee can **copy** selections of the licensed work, or of previously copied and/or edited versions of the licensed work. The licensee identifies a name for the local storage file where the copy will be kept. No reference is made to the payment function. When copying the semantics is that this function is invoked and duly applied. That is, copy payments are automatically made: subtracted from the licensees account and forwarded to the licensor. "slide 408"

(9.) The licensee can **edit** selections of the licensed work, or of copied and/or previously edited versions of the licensed work. The licensee identifies a name for the local storage file where the new edited version will be kept. The result of editing is a new work. No reference is made to the **payment** function. "slide 409" When copying the semantics is that this function is invoked and duly applied. That is, copy payments are automatically made: subtracted from the licensees account and forwarded to the licensor. Although no reference is made to any edit functions these are made available to the licensee when invoking the edit command. You may think of these edit functions being downloaded at the time of downloading the license. Other than this we need not further specify the editing functions. Same remarks apply to the above copying functions. "slide 410"

(10.) The licensee can further **sub-license** copied and/or edited work. The licensee must give the license being assembled a unique name. And the licensee must choose to whom to license this work. A sub-license, like does a license, authorises which actions can be performed, and then with which one of a set of alternative selection sequences. No payment function is explicitly mentioned.

"slide 411"

It is to be semi-automatically derived (from the originally licensed payment fee function and the licensee's payment demands) by means of functionalities provided as part of the licensed payment fee function.

The sub-license command information is thus **compiled** (**assembled**) into a license of the form given in (1.–3.). The licensee becomes the licensor and the recipient of the new, the sub-license, become the new licensee. The assemblage refers to the context of the action. That context knows who, the licensor, is issuing the sub-license. From the license label of the command it is known whether the sub-license actions are a subset of those for which sub-licensing has been permitted.

A Hospital Health Care License Language[80]

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.

Patients and Patient Medical Records[81]

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.

Medical Staff[82]

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.

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.

[80] "SLIDE 412"
[81] "SLIDE 413"
[82] "SLIDE 414"

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 [14].[83]

A Notion of License Execution State[84]

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. 2.2 for an idealised hospitalisation plan. "slide 416"

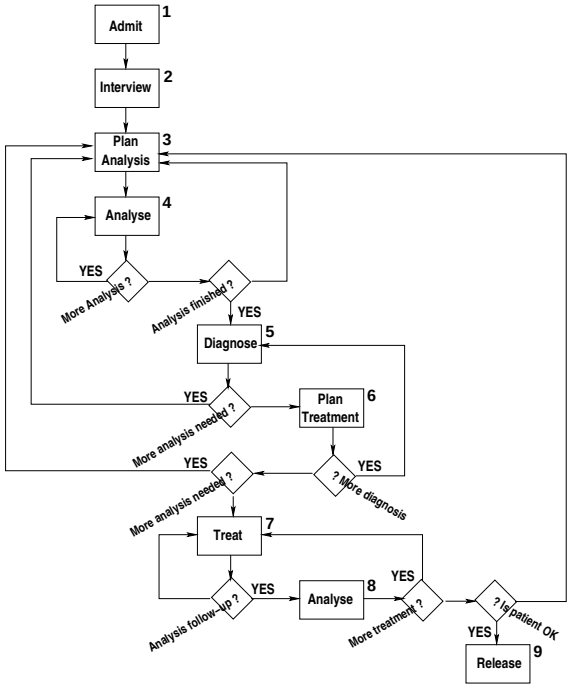


Fig. 2.2. An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

"slide 417"

[83] As this work, [14], has yet to be completed the syntax and annotations given here may change.

[84] "SLIDE 415"

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 and prevent it being misused!

The License Language[85]

"slide 419" 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**

"slide 420" 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.

"slide 421" (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 sub-set 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

"slide 422" to confer actions to other staff.

[85] "SLIDE 418"

(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. "slide 423"

8. Action = Ln × Act
9. Act = Stmt | SubLic
10. SubLic = mkSubLic(sublicensee:Ln,license:ML)

"slide 424"

(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. Each 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. "slide 425"

(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.

Public Government and the Citizens[86]

The Three Branches of Government

By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)[87], 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.). "slide 427"

Documents[88]

A crucial means of expressing public administration is through *documents*.[89] 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.) "slide 429"

[86] "SLIDE 426"

[87] *De l'esprit des lois* (*The Spirit of the Laws*), published 1748

[88] "SLIDE 428"

[89] Documents are, for the case of public government to be the "equivalent" of artistic works.

Documents are *created*, *edited* and *read*; and documents can be *copied*, *distributed*, the subject of *calculations* (interpretations) and be *shared* and *shredded*.

Document Attributes[90]

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.

"slide 431"

With these operations on documents, and hence as attributes of documents one can, again conceptually, associate the *location* and *time* of these operations.

Actor Attributes and Licenses[91]

With actors (whether agents of public government or citizens) one can associate the *authority* (i.e., the *rights*) these actors have with respect to performing actions on documents. We now intend to express these *authorisations as licenses*.

Document Tracing[92]

An issue of public government is whether citizens and agents of public government act in accordance with the laws — with actions and laws reflected in documents such that the action documents enables a trace from the actions to the laws "governing" these actions.

We shall therefore assume that every document can be traced back to its law-origin as well as to all the documents any one document-creation or -editing was based on.

A Document License Language[93]

The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform.

The Form of Licenses

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

[90] "SLIDE 430"
[91] "SLIDE 432"
[92] "SLIDE 433"
[93] "SLIDE 434"

"slide 435"

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)

"slide 436"

(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.[94] A set of operations on (named) documents are granted.

"slide 437"

(3.–5.) Actors who have issued named licenses may extend, restrict or withdraw the license rights (wrt. operations, or fully).

(6.) There are nine kinds of operation authorisations. Some of the next explications also explain parts of some of the corresponding actions (see (16.–24.).

(7.) There are names of documents (Dn), names of classes of documents (DCn), and there are unique document identifiers (UDI).

"slide 438"

(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[95]) typed by a document class name (dcn:DCn) and possibly based on one or more identified documents (over which the licensee (at least) has reading rights). We can presently omit consideration of the document class concept. "based on" means that the initially void document contains references to those (zero, one or more) documents.[96] The "based on" documents are moved from licensor to licensee.

"slide 439"

[94] This constraint can be enforced by letting the actor name be part of the license name.

[95] — hence there is an assumption here that the create operation is invoked by the licensee exactly (or at most) once.

[96] They can therefore be traced (etc.) — as per [32].

(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.

"slide 440"

(10.) **Reading** a document only changes its "having been read" status (etc.) — as per [32]. 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.

"slide 441"

(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 [32]. 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.

"slide 442"

(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.

"slide 443"

(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.

"slide 444"

(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.

"slide 445"

(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.

"slide 446"

(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. "slide 447"

17. Action = Ln × Clause
18. Clause = Cre | Edt | Rea | Cop | Lic | Sha | Rvk | Rel | Ret | Cal | Shr
19. Cre == mkCre(dcn:DCn,based_on_docs:UID-**set**)
20. Edt == mkEdt(uid:UID,based_on_docs:UID-**set**)
21. Rea == mkRea(uid:UID)
22. Cop == mkCop(uid:UID)
23. Lic == mkLic(license:L)
24. Sha == mkSha(uid:UID,with:An-**set**)
25. Rvk == mkRvk(uid:UID,from:An-**set**)
25. Rev == mkRev(uid:UID,from:An-**set**)
26. Rel == mkRel(dn:Dn,uid:UID)
27. Ret == mkRet(dn:Dn,uid:UID)
28. Cal == mkCal(fct:Cfn,over_docs:UID-**set**)
29. Shr == mkShr(uid:UID)

"slide 448"

A clause elaborates to a state change and usually some value. The value yielded by elaboration of the above create, copy, and calculation clauses are **unique document identifiers**. These are chosen by the "system". "slide 449"

(17.) Actions are **tagged** by the name of the license with respect to which their authorisation and document names has to be checked. No action can be performed by a licensee unless it is so authorised by the named license, both as concerns the operation (create, edit, read, copy, license, share, revoke, calculate and shred) and the documents actually named in the action. They must have been mentioned in the license, or, created or copies of downloaded (and possibly edited) documents or copies of these — in which cases operations are inherited. "slide 450"

(19.) A licensee may **create** documents if so licensed — and obtains all operation authorisations to this document.

(20.) A licensee may **edit** "downloaded" (edited and/or copied) or created documents.

(21.) A licensee may **read** "downloaded" (edited and/or copied) or created and edited documents.

(22.) A licensee may (conditionally) **copy** "downloaded" (edited and/or copied) or created and edited documents. The licensee decides which name to give the new document, i.e., the copy. All rights of the master are inherited to the copy. "slide 451"

(23.) A licensee may **issue licenses** of the kind permitted. The licensee decides whether to do so or not. The licensee decides to whom, over which, if

any, documents, and for which operations. The licensee looks after a proper ordering of licensing commands: first grant, then sequences of zero, one or more either extensions or restrictions, and finally, perhaps, a withdrawal.

"slide 452"

(24.) A "downloaded" (possibly edited or copied) document may (conditionally) be **shared** with one or more other actors. Sharing, in a digital world, for example, means that any edits done after the opening of the sharing session, can be read by all so-granted other actors.

(25.) Sharing may (conditionally) be **revoked**, partially or fully, that is, wrt. original "sharers".

"slide 453"

(26.) A document may be **released**. It means that the licensor who originally requested a document (named dn:Dn) to be created now is being able to see the results — and is expected to comment on this document and eventually to re-license the licensee to further work.

(27.) A document may be **returned**. It means that the licensor who originally requested a document (named dn:Dn) to be created is now given back the full control over this document. The licensee will no longer operate on it.

"slide 454"

(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.

Discussion: Comparisons[97]

"slide 456"

We have "designed", or rather proposed three different kinds of license languages. In which ways are they "similar", and in which ways are they "different"? Proper answers to these questions can only be given after we have formalised these languages. The comparisons can be properly founded on comparing the semantic values of these languages.

But before we embark on such formalisations we need some informal comparisons so as to guide our formalisations. So we shall attempt such analysis now with the understanding that it is only a temporary one.

Work Items[98]

The **work items** of the **artistic** license language(s) are essentially "kept" by the licensor. The **work items** of the **hospital health care** license language(s) are fixed and, for a large set of licenses there is one work item, the patient which is shared between many licensors and licenses. The **work items** of the **public administration** license language(s) — namely document — are distributed to or created and copied by licenses and may possibly be shared.

[97] "SLIDE 455"

[98] "SLIDE 457"

Operations[99]

The **operations** of the **artistic** license language(s) are are essentially "kept" by the licensor. The **operations** of the **hospital health care** license language(s) are are essentially "kept" by the licensees (as reflected in their professional training and conduct). The **operations** of the **public administration** license language(s) are essentially "kept" by the licensees (as reflected in their professional training and conduct).

Permissions and Obligations[100]

Generally we can say that the **modalities** of the **artistic** license language(s) are essentially **permissions** with **payment** (as well as use of licensor functions) being an **obligation**; that the **modalities** of the **hospital health care** license language(s) are are essentially **obligations**; and, as well, that the **modalities** of the **public administration** license language(s) are essentially **obligations** We shall have more to say about permissions and obligations later (much later).

Script and Contract Languages[101]

By a **domain** **script** **language** we mean the definition of a set of licenses and actions where these licenses when issued and actions when performed have morally obliging power.

By a **domain** **contract** **language** we mean a domain script language whose licenses and actions have legally binding power, that is, the issue of licenses and the invocation of actions may be contested in a court of law. We now refer to licenses as contracts.

Review of Support Examples[102]

TO BE WRITTEN

The Aircraft Simulator Script[103]

TO BE WRITTEN

The Bill-of-Lading Script[104]

TO BE WRITTEN

99 "SLIDE 458"
100 "SLIDE 459"
101 "SLIDE 460"
102 "SLIDE 461"
103 "SLIDE 462"
104 "SLIDE 463"

The Timetable Script Language[105]

TO BE WRITTEN

The Bus Transport Contract Language[106]

TO BE WRITTEN

"slide 466"

TO BE WRITTEN

"slide 467"

TO BE WRITTEN

Modelling Scripts[107]

2.9.9 Human Behaviours

"SLIDE 469"

Characterisation 70 (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. ∎

A Meta-characterisation of Human Behaviour[108]

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

Our final specification pattern is therefore:

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

"slide 471"

[105] "SLIDE 464"
[106] "SLIDE 465"
[107] "SLIDE 468"
[108] "SLIDE 470"

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed — whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not. "slide 472"

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.

Review of Support Examples[109]

TO BE WRITTEN "slide 474"

TO BE WRITTEN

On Modelling Human Behaviour[110]

To model human behaviour is, "initially", much like modelling management and organisation. But only 'initially'. The most significant human behaviour modelling aspect is then that of modelling non-determinism and looseness, even ambiguity. So a specification language which allows specifying non-determinism and looseness (like CafeOBJ and RSL) is to be preferred.

For examples of domain human behaviour modelling and resulting documents we refer to Appendix K.

2.9.10 Consolidation of Domain Facets Description "slide 476"

The many previous domain facet stages may have yielded descriptions which, typically at the formal level, does not reveal how it all "hangs together". In such cases, and in general, consolidation of these domain facet documentaton stages could take the following forms.

With each potential management unit we associate a process or an indexed set of two or more processes, usually an indeterminate number. Such management units will usually involve entities and behaviours — whether staff of entity behaviours. Usually type definitions and axioms (about sorts) and value definitions of auxiliary and well-formedness functions about values can be kept separate from the process definitions. The entity processes usually take, as arguments, the entity whose time-wise behaviour and interaction with oother entity processes is being domain modelled. "slide 477"

With each structural component of the organisation we associate one or more channels, or vector or array or tensor (or . . .) indexed sets of channels.

[109] "slide 473"

[110] "slide 475"

More to come

2.9.11 Discussion of Facets "slide 478"

"slide 479"
"slide 480"

2.10 Domain Verification "slide 481"

For examples of domain verification modelling and resulting documents we refer to Appendix L, Sect. L.1 (Page 457).

2.11 Domain Validation "slide 482"

For examples of domain validation modelling and resulting documents we refer to Appendix L, Sect. L.2 (Page 457).

2.12 Verification Versus Validation "slide 483"

2.13 Domain Theory Formation "slide 484"

For examples of domain theory formation modelling and resulting documents we refer to Appendix L, Sect. L.3 (Page 457).

2.14 Domain Engineering Process Graph "slide 485"

2.15 Domain Engineering Documents "slide 486"

There are basically three kinds of domain development documents:

- information documents,
- description documents and
- analytic documents.

We have already covered, in Sect. 1.6, the concept of informative documents.

For examples of informative domain documents modelling and resulting documents we refer to Appendix E, Sect. E.1 (Page 317).

In the next two sections we shall cover the motivation for and principles and techniques of description and analysis documents.

2.15.1 Description Documents "slide 487"

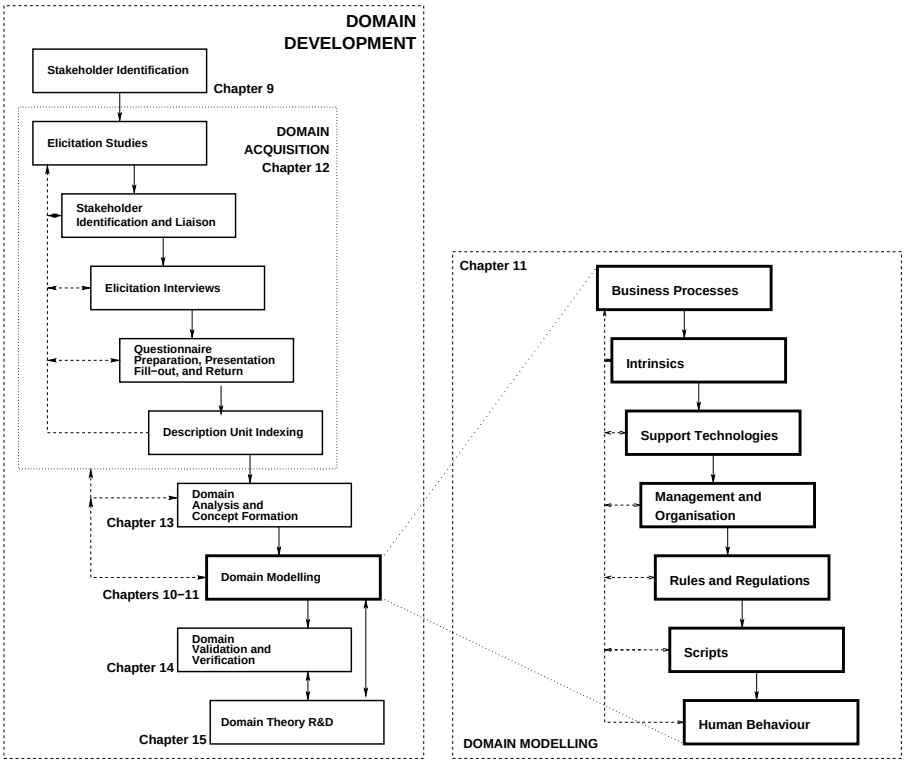


Fig. 2.3. Domain engineering process graph

1 Stakeholders
2 The Acquisition Process
   (a) Studies
   (b) Interviews
   (c) Questionnaires
   (d) Indexed Description Units
3 Terminology
4 Business Processes
5 Facets:
   (a) Intrinsics
   (b) Support Technologies
   (c) Management and Organisation
   (d) Rules and Regulations
   (e) Scripts
   (f) Human Behaviour
6 Consolidated Description

2.15.2 Analytic Documents

"SLIDE 488"

1 Domain Analysis and Concept Formation
   (a) Inconsistencies
   (b) Conflicts
   (c) Incompletenesses
   (d) Resolutions
2 Domain Validation
   (a) Stakeholder Walkthroughs

(b) Resolutions
3 Domain Verification
(a) Model Checkings
(b) Theorems and Proofs
(c) Test Cases and Tests
4 (Towards a) Domain Theory

2.16 Summary

"SLIDE 489"

2.17 Exercises

Exercise 11. Route Type: Given the concepts of hubs, links, nets, hub identifiers, link identifiers, and their associated observer functions you are to define the following concepts:

- transport routes (or just routes, for short) as suitable sequences of hub and link identifiers.

If hubs identified by h_i, h_j and h_k are linked by links identified by ℓ_{ij} and ℓ_{jk}, then the sequence of triples $< (h_i, \ell_{ij}, h_j), (h_j, \ell_{jk}, h_k) >$ is an example of a way to represent a route.

- In other words, please define the type of such routes both informally, by a precise narration, and formally, by a concrete type definition involving Cartesians and lists.

Hint: You may wish to consult Appendix S's outline of the Cartesian and list concepts:

- Cartesian types (Item 9 on page 504 and Page S.1.1, Item [9]),
- list types (Items 10–11 and Page S.1.1, Items [10-11]) and
- their definition (framed box 'Type Definition' Page 505).
- Cartesian and list enumerations (Sects. S.3.3 on page 509 and S.3.4 on page 509) and
- Cartesian and list operations (Sects. S.3.7 on page 513 and S.3.8 on page 513).

Hint: We suggest that routes, r:R, are sequences of adjacent link visits where a link visits are triples of hub, link, and hub identifiers — suitably constrained.

- Thus you are also to define a suitable well-formedness predicate:
  1 wf_R over routes and nets
  2 which checks that adjacent link visits share hub identifiers
  3 and that all hub and link identifiers are indeed indentifiers of hubs and links of the net.

Solution 11 Vol. II, Page 533, suggests a way of answering this exercise.

Exercise 12. Route Generation: Given a net you are to formally define a function which generates the set of all finite routes of the net.

1 **Basis Clause:** A singleton sequence of the triplet of a (from) hub identifier, h_f, a(n along) link identifier, ℓ_{ft}, and a (to) hub identifier, h_t, where the link, ℓ, identified by ℓ_{ft} is connected to the two distinct hubs identfied by the distinct identifiers h_f and h_t, such a singleton sequence is a route.
2 In the above defined singleton route, $< (h_f, \ell_{ft}, h_t) >$, h_f is called the first hub identifier and h_t is called the last hub identifier.
3 **Inductive Clause:** If r and r' are routes such that some hub identifier h_i is the last in r and the first in r', then the concatenation, rr', of the two routes is a route.
4 **Extremal Clause:** Only such routes which can be formed by a finite number of uses of the basis and the induction clauses are routes.

Solution 12 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 13. Route Lengths:

- With every link we associate a length.
- With every hub we can associate a set of lengths, one for each pair of incoming and outgoing link. (The set can be thought of as being 'indexed' by the pairs of link identfiers.)

Now define the following functions:

1 Given a route and a net (of the route) calculate the length of the route in that net.
2 Given a pair of distinct hub identifiers and a net of the identified hubs find the set of routes between those hubs.
3 As for Item 2, find the shortest route between a given pair of distinct hub identifiers.
4 As for Items 2–3, find the longest route.
5 Given a net and a set of distinct hub identifiers find the set of routes that visits the so-identified hubs (at least once).
6 As for Item 5, find the shortest of the routes "found" by the function of that item.

Solution 13 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 14. DE4:
Solution 14 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 15. DE4:
Solution 15 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 16. DE4:

Solution 16 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 17. DE4:
Solution 17 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 18. DE4:
Solution 18 Vol. II, Page 534, suggests a way of answering this exercise.

"slide 490"

3

Requirements Engineering

"SLIDE 492"

3.1 Discussion of The Requirements Concept

3.1.1 Some Principles

The objective of requirements engineering is to create a requirements prescription.

IEEE Definition of 'Requirements'

By a requirements we understand (cf. IEEE Standard 610.12 [113]): *"A condition or capability needed by a user to solve a problem or achieve an objective".*

"slide 493"

The above definition[1] is adequate for our purposes. It stresses what requirements are. It is not operational, and that is good. It does not define the thing, the requirements, by how they look, or how you construct them. The 'what' (not the 'how') of requirements is the purpose of this and the next seven chapters.

"slide 494"

The "Golden Rule" of Requirements Engineering

Principle 5 (Requirements Engineering [1])]Prescribe only those requirements that can be objectively shown to hold for the designed software. ∎

"Objectively shown" means that the designed software can either be proved (verified), or be model checked, or be tested, to satisfy the requirements.

"slide 495"

An "Ideal Rule" of Requirements Engineering

Principle 6 (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. ∎

[1] We shall mostly be using the term 'requirements' in its plural form, but think of it as "one body" of such!

The rule is labelled "ideal". We shall not show such precautions in this volume. They ought to be shown. But either we would show one, or a few instances, and they would "drown" in the mass of material otherwise presented. Or they would, we claim, trivially take up too much space. The rule is clear. It is a question for proper management to see that it is adhered to.

"slide 496"

Principle 7 (Requirements Adequacy) Make sure that requirements cover what users expect. ∎

That is, do not express a requirement for which you have no users, but make sure that all users' requirements are represented or somehow accommodated. In other words: the requirements gathering process needs to be like an extremely "fine-meshed net": One must make sure that all possible stakeholders have been involved in the requirements acquisition process, and that possible conflicts and other inconsistencies have been obviated.

"slide 497"

The approach which we put forward in this chapter, namely "deriving" domain requirements "directly" from domain descriptions, and systematically conceiving interface requirements also from domain descriptions shall help secure adherence to this 'Requirements Adequacy' principle. We shall soon explain the terms 'domain requirements' and 'interface requirements'.

"slide 498"

Principle 8 (Requirements Implementability) Make sure that requirements are implementable. ∎

That is, do not express a requirement for which you have no assurance that it can be implemented. In other words, although the requirements phase is not a design phase, one must tacitly assume, perhaps even indicate, somehow, that an implementation is possible. But the requirements in and by themselves, stay short of expressing such designs.

"slide 499"

Principle 9 (Requirements Verifiability and Validability) Make sure that requirements are verifiable and can be validated. ∎

That is, do not express a requirement for which you have no assurance that it can be verified and validated. In other words, once a first-level software design has been proposed, one must show that it satisfies the requirements. Thus specific parts of even abstract software designs are usually provided with references to specific parts of the requirements that they are (thus) claimed to implement.

"slide 500"

We conclude this discussion.

Characterisation 71 (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". ∎

3.1.2 One Domain, Many Requirements

"SLIDE 501"

Domain descriptions nearly always cover a much larger span than do any one individual (set of) requirements, that is, a requirements for a specific software product. Thus one can expect one domain description to be the basis for many (more-or-less) distinct requirements prescriptions, which, since they are (to be) based on the same domain description, and when they share some domain phenomena and concepts, must (somehow) be made to fit one another. We shall cover the notion of requirements fitting in Sect. 3.9.3 (starting Page 131) when covering the topic of domain requirements.

3.1.3 The Machine as Target

"SLIDE 502"

A requirements prescription specifies externally observable properties of simple entities, functions, events and behaviours of **the machine** such as the requirements stakeholders wish them to be.

3.1.4 Machine = Hardware + Software

"SLIDE 503"

The **machine** is what is required, that is, the **hardware** and **software** that is to be designed and which are to satisfy the requirements.

3.1.5 On "Derivation" of Requirements

"SLIDE 504"

It is a highlight of this book 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. Conventional requirements engineering, as covered in a great number of software engineering textbooks [175, 181, 201, 222, 88], does not have (such) a scientific foundation. This foundation allows us to pursue requirements engineering in quite a new manner. "slide 505"

The way in which we shall pursue the central core of requirements engineering will now be sketched. When modelling the requirements, after all the initial stages of requirements stakeholder identification, etc., we "divide" the work into three kinds of requirements: (i) the domain requirements, (ii) the interface requirements, and (iii) the machine requirements. "slide 506"

These sub-stages are related to their underlying domain as follows. (i) The domain requirements are those requirements which can be expressed sôlely using terms from the domain (in addition to ordinary, say, English. (iii) The machine requirements are those requirements which can be expressed sôlely using terms from the machine, that is, the hardware and software regime. (ii) The interface requirements are then those requirements which can be expressed using terms from both the domain and the machine. "slide 507"

This approach materially alters the way in which we pursue the preparatory requirements engineering stages of stakeholder identification, requirements acquisition, requirements analysis and concept formation, business process re-engineering and requirements terminology.

We will cover these preparatory requirements engineering stages in two rounds: in an overview fashion, Sect. 3.2.1, and in some detail, Sects. 3.3–3.8,. In our coverage we rely on the reader having first carefully studied the "similarly named" domain engineering stage.

3.1.6 Summary

"SLIDE 508"

A **requirements prescription** thus (**putatively**) expresses what there should be. A requirements prescription expresses nothing about the design of the possibly desired (required) software.

• • •

We shall show how a major part of a requirements prescription can be "derived" from "its" prerequisite domain description.

3.2 Stages of Requirements Engineering

"SLIDE 509"

3.2.1 An Overview of "What To Do?"

The present section's main material, Pages 112–116, on "what to do" in requirements development is very much like section on "what to do" in domain development (Pages 52–54). So we kindly ask the reader to recall that former section.

We first summarise what is to be done, ending that summary with an overview listing (Page 116) (as we did for the stages of domain engineering (Page 55) . Then, in subsequent section we cover, in some detail, how to do that which is to be done (Sect. 3.3–3.13).

[1] Requirements Information[2]

This (numbered [1]) section is very much like the similarly named subsection ([1]) on Page 52.

The purpose of this stage of development, to repeat, is to record all relevant administrative, socio-economic, budgetary, project management (planning) and such non-formalisable information which has a bearing on the requirements prescription project.

For more specifics on this topic we refer to Sect. 3.3 (Page 117).

For the example 'requirements information' document we refer to (Appendix) Sect. M.1, Pages 461–466.

[2] "SLIDE 510"

[2] Requirements Stakeholder Identification[3]

This (numbered [2]) section is very much like the similarly named subsection ([2]) on Page 52.

The purpose of this stage of development is to identify the requirements stakeholders. They are usually a proper subset of the domain stakeholders. One does not need to interact with all domain stakeholders as that was supposedly done in the domain description development phase, and the domain description is, of course, assumed available and the basis for the requirements prescription development phase. For more specifics on this topic we refer to Sect. 3.4

For the example 'requirements stakeholder identification' document we refer to (Appendix) Sect. M.2, Pages 466–466.

[3] Requirements Acquisition[4]

This (numbered [3]) subsection is very much like the similarly named subsection ([3]) on Page 53.

The purpose of this stage of development is to acquire and elicit, to list, to enumerate, to collect the requirements. Now since, as we shall see, the requirements shall consist of three parts: the domain requirements, the interface requirements and the machine requirements, and since the first two relies very strongly on the domain description, a major part of the requirements acquisition takes on a rather different form than the way in which domain acquisition takes place.

For more specifics on this topic we refer to Sect. 3.5

For the example 'requirements acquisition' document we refer to (Appendix) Sect. M.3, Pages 466–466.

[4] Requirements Analysis & Concept Formation[5]

This (numbered [4]) subsection is very much like the similarly named subsection ([4]) on Page 53.

The purpose of this stage of development is to analyse and conceive of concepts around which to structure the requirements, for those relevant such concepts that are not already part of the domain description.

While performing the steps of (a) the domain requirements, (b) the interface requirements and (c) the machine requirements stages of development these steps may give rise to inconsistencies incompletenesses for which analysis has to check their absence. "slide 514"

While deriving the above three stages (a–b–c) analyses must be made to secure that the required simple entities, functions, events and behaviours are

[3] "SLIDE 511"

[4] "SLIDE 512"

[5] "SLIDE 513"

computable. (For a domain description incomputable concepts are allowed.) (In fact, the whole purpose of requirements construction is to secure some form of computability[6].

For more specifics on this topic we refer to Sect. 3.6

For the example 'requirements analysis' document we refer to (Appendix) Sect. M.4, Pages 467–467.

[5] Requirements Business Process Re-Engineering[7]

This (numbered [5]) subsection is very much like the similarly named subsection ([5]) on Page 53.

The background for this stage of development is the set of current business processes as carried out in the domain at present and as described during construction of the domain description. The purpose of this stage of development is to prescribe how thee business processes are to be in future, once the required software has been installed; that is: the business processes often need be re-engineered since that is (to be) assumed by the required software.

For more specifics on this topic we refer to Sect. 3.7

For the example 'business process re-engineering' document we refer to (Appendix) Sect. M.5, Pages 467–469.

[6] Requirements Terminology[8]

This (numbered [6]) subsection is very much like the similarly named subsection ([6]) on Page 53.

The purpose of this stage of development, one which "links" up to the domain terminology, is to extend that domain terminology with the new terms that arise as a consequence of interface and machine requirements. (The stages of development [interface and machine requirements] will be covered later.)

For more specifics on this topic we refer to Sect. 3.8

For the example 'requirements terminology' document we refer to (Appendix) Sect. M.6, Pages 469–469.

[7] Requirements Modelling[9]

This (numbered [7]) subsection is very much like the similarly named subsection ([7]) on Page 54.

[6] By this "some form" is meant: Either the concepts are computable or, through interaction with an environment (a human or otherwise) the desired effect can be achieved.

[7] "SLIDE 515"

[8] "SLIDE 516"

[9] "SLIDE 517"

The purpose of this stage of development is to develop a requirements prescription, that is a narrative and a formal specification of three requirements facets: domain requirements, interface requirements and machine requirements. "slide 518"

These relate as follows: domain requirements are those requirements which can be expressed sôlely by using terms from the domain (and otherwise ordinary terms of English); machine requirements are those requirements which can be expressed sôlely by using terms "from" the machine, i.e., hardware and software terms (and otherwise ordinary terms of English); interface requirements are those requirements which can only be expressed using terms both of the domain and of the machine. "slide 519"

For more specifics on this topic we refer to Sect. 3.9

For the example 'requirements prescription' document we refer to Appendices N–P, Pages 471–491.

[8] Requirements Verification[10]

This (numbered [8]) subsection is very much like the similarly named subsection ([8]) on Page 54.

The purpose of this stage of development, which normally goes hand–in–hand with requirements modelling, is to verify (prove, model check and/or test) properties of what is being prescribed. Examples of requirements prescription verification are that all arguments to defined functions are within their range of applicability, that postulated functions can be implemented, that defined functions are of a desired complexity, etcetera.

For more specifics on this topic we refer to Sect. 3.10

For the example 'requirements verification' document we refer to (Appendix) Sect. Q.1, Pages 493–493.

[9] Requirements Validation[11]

This (numbered [9]) subsection is very much like the similarly named subsection ([9]) on Page 54.

The purpose of this stage of development which normally comes after proper completion of a requirements document, is to make sure that what has been prescribed is actually what the relevant requirements stakeholder have requested. The validation process is necessarily informal in that it relies sôlely on the narrative prescriptions as these are the only ones that all requirements stakeholders understand.

For more specifics on this topic we refer to Sect. 3.11

For the example 'requirements validation' document we refer to (Appendix) Sect. Q.2 Pages 493–493.

[10] "SLIDE 520"

[11] "SLIDE 521"

[10] Requirements Satisfiability and Feasibility[12]

This (numbered [10]) section is new. It has no direct counterpart in domain engineering.

The purpose of this stage of development is to secure that the full requirements are implementable and in a way that is economic and relevant. By feasible (i.e., relevance) we mean that an implementation does not consume unreasonable resources: time, space, etcetera.

For more specifics on this topic we refer to Sect. 3.12

For the example 'requirements satisfiability and feasibility' document we refer to (Appendix) Sect. Q.3 Pages, 493–493.

[11] Requirements Theory Formation[13]

This (numbered [11]) subsection is very much like the similarly named subsection ([11]) on Page 54.

The purpose of this stage of development further develop the domain theory, Sect. 2.13, by examining the theorems of that theory as to their also holding for the specific requirements hold and to develop such theorems that hold for the specific requirements.

For more specifics on this topic we refer to Sect. 3.13

For the example 'requirements theory formation' document we refer to (Appendix) Sect. Q.4 Pages 493–493.

3.2.2 A Summary Enumeration

"SLIDE 524"

[12] "SLIDE 522"

[13] "SLIDE 523"

3.3 Requirements Information "slide 525"

This (numbered [1]) section is very much like the similarly named subsection ([1]) on Page 52.

For the example 'requirements information' document we refer to (Appendix) Sect. M.1, Pages 461–466.

We have earlier, as mentioned above, extensively (Pages 6–24) covered the general issues of informative documents. The reader is strongly encouraged to review those pages, Sect. 1.5. Suffice it here to emphasize the following.

We highlight, in the next many "highlighted" paragraphs, what is special, to requirements prescription developments, with respect to the many items of project information.

Current Situation: Cf. Sect. 1.6.3, Page 9 "slide 526"

As mentioned in Sect. 1.6.3 on page 9 the context in which the requirements developments starts must be emphasized. That context invariably includes the existence of a domain description. Please no reference to possible software designs.

We refer to Appendix Sect. M.1.3 starting on Page 462 for an example related to the *current situation* topic.

Needs and Ideas: Cf. Sect. 1.6.4, Pages 9–10 "slide 527"

TO BE WRITTEN

We refer to Appendix Sect. M.1.4 starting on Page 462 for an example related to the *needs and ideas* topic.

Concepts and Facilities: Cf. Sect. 1.6.5, Pages 10–11 "slide 528"

TO BE WRITTEN

We refer to Appendix Sect. M.1.5 starting on Page 463 for an example related to the *concepts and facilities* topic.

Scope and Span: Cf. Sect. 1.6.6, Page 11 "slide 529"

TO BE WRITTEN

We refer to Appendix Sect. M.1.6 starting on Page 464 for an example related to the *scope and span* topic.

Assumptions and Dependencies: Cf. Sect. 1.6.7, Pages 11–12 "slide 530"

TO BE WRITTEN

We refer to Appendix Sect. M.1.7 starting on Page 465 for an example related to the *assumptions and dependencies* topic.

Implicit/Derivative Goals: Cf. Sect. 1.6.8, Page 12 "slide 531"

TO BE WRITTEN

We refer to Appendix Sect. M.1.8 starting on Page 465 for an example related to the *implicit/derivative goals* topic.

Synopsis: Cf. Sect. 1.6.9, Page 13 "slide 532"

TO BE WRITTEN

We refer to Appendix Sect. M.1.10 starting on Page 465 for an example related to the *synopsis* topic.

"slide 533" **Software Development Graphs:** Cf. Sect. 1.6.10, Pages 13–15

TO BE WRITTEN

We refer to Appendix Sect. M.1.11 starting on Page 465 for an example related to the *software development graph* topic.

"slide 534" **Resource Allocation:** Cf. Sect. 1.6.11, Pages 15–16

TO BE WRITTEN

We refer to Appendix Sect. M.1.12 starting on Page 465 for an example related to the *resource allocation* topic.

"slide 535" **Budget (and Other) Estimates:** Cf. Sect. 1.6.12, Page 16

TO BE WRITTEN

We refer to Appendix Sect. M.1.13 starting on Page 466 for an example related to the *budget (and other) estimates* topic.

"slide 536" **Standards Compliance:** Cf. Sect. 1.6.13, Pages 16–19

TO BE WRITTEN

We refer to Appendix Sect. M.1.14 starting on Page 466 for an example related to the *standards compliance* topic.

"slide 537" **Contracts and Design Briefs:** Cf. Sect. 1.6.14, Pages 19–23

TO BE WRITTEN

We refer to Appendix Sect. M.1.15 starting on Page 466 for an example related to the *contract and design brief* topic.

"slide 538"

MORE TO COME

3.4 Requirements Stakeholders

"SLIDE 539"

This section is an expansion on Item [2] on Page 52 and Sect. 2.4

On Page 113 we wrote that the set of requirements stakeholder *is usual* a proper subset of of the domain stakeholders. We may need to modify this statement. We assume that a list of domain stakeholders omitted administrative staff from the areas of general services staff: accounting, archiving (journal), general procurement, etcetera. We assume they were omitted since the domain, "technically speaking" was not about these areas. The domain, for example, may be about railways for which we may have included passenger ticketing staff but for which we omitted for example the "back room" accounting staff. That latter staff would, in principle, not know whether they were doing accounting for a railway-related company or for a financial service system or for a hospital. If the requirements are for such general services of a domain which do indeed impinge on such, previously omitted staff, then such staff gropus must be included among the stakeholders. (But we do remind the "slide 540" reader that our domain is in such a case extended with the phenomena and concepts of the areas of general services thus identified — and appropriate domain extensions must be provided for.)

For the example 'requirements stakeholder identification' document we refer to (Appendix) Sect. M.2, Pages 466–466. We refer to Appendix Sect. M.2 starting on Page 466 for an example related to the *requirements stakeholder* topic.

3.5 Requirements Acquisition

"SLIDE 541"

We have, in Sect. 3.2.1, Item [3], on Page 113, outlined the three stages of requirements modelling: domain requirements, interface requirements and machine requirements. These requirements modelling stages were first overviewed, in more detail in Sect. 3.1.5, Pages 111–111. The three requirements modelling stages will be dealt with in "final" detail in Sects. 3.9.3–3.9.5, Pages 128–163.

We shall assume you have followed those brief outlines. Requirements acquisition now basically follows these three kinds of requirements modelling stages.

3.5.1 Domain Requirements Acquisition

"SLIDE 542"

Domain requirements acquisition is based sôlely on the domain description. From that domain description is developed, in sub-stages (or steps) of development a domain requirements acquisition document. The steps are *projection, instantiation, determination, extension* and *fitting*. Each step takes a document and results in a requirements acquisition document. The first step takes a domain description document. All steps results in a domain requirements acquisition document. "slide 543"

Sect. 3.9.3 shall outline the details of the *projection, instantiation, determination, extension* and *fitting* operations. In the domain requirements acquisition stage one basically marks up a "requirements acquisition copy" of the domain description. In the domain requirements (Sect. 3.9.3) acquisition stage one collects the 'domain requirements acquisition' mark-ups into a consistent acquisition document editing it as per the mark-ups. All domain-to-domain requirements acquisition operations (*projection, instantiation, determination, extension* and *fitting*) are conducted interactively, between the domain engineer and all the relevant requirement stakeholder, in turn, one after the other, and, in principle, in no particular order. The domain engineer, in a sense, guides the requirement stakeholders through the emerging domain requirements acquisition prescription document determining what to *project, instantiate,* make more *deterministic, extend* and *fit*.

"slide 544"

3.5.2 Interface Requirements Acquisition

Interface requirements acquisition is based on the domain requirements acquisition document and on some awareness of the facilities of machine being required. Awareness and determination of which these facilities are, or could be, evolve as a result of conducting the below steps of interface requrements. First the domain engineer and the relevant requirement stakeholders determine which phenomena simple entities, functions, events and behaviours of the domain need be represented by the machine. Then they go through each of these kinds of phenomena to determine what sharing means: what properties of simple entities, functions, events and behaviours are to be satisfied by the machine. This entails decisions on abstractions of initial data input and refreshment, interactive computation of functions between the machine and the domain, "translation" of events in the domain into the machine, and interactive behaviours between the machine and the domain. Again, each of these steps result in the input requirements acquisition document being futher annotated — as were the domain requirements acquisition documents. Proper requirements modelling documents now take these annotated acquisition documents and rework them, "clean them up", into proper requirements prescripton documents.

"slide 545"

"slide 546"

3.5.3 Machine Requirements Acquisition

"SLIDE 547"

Machine requirements acquisition is basically concerned with the acquisition of such required properties of the desired machine as machine: performance, dependability, maintenance, platform and documentation. We shall defer the acquisition aspects of these machine requirements till we later, Sect. 3.9.5, treat machine requirements modelling.

• • •

We refer to Appendix Sect. M.3 starting on Page 466 for an example related to the *requirements acquisition* topic.

3.6 Analysis and Concept Formation

"slide 548"

TO BE WRITTEN

We refer to Appendix Sect. M.4 starting on Page 467 for an example related to the *requirements analysis and concept formation* topic.

3.7 Business Process Re-Engineering

"slide 549"

Characterisation 72 (Business Process Reengineering) By *business process reengineering* we understand the reformulation of previously adopted business process descriptions, together with additional business process engineering work. ∎

3.7.1 What Are *BPR Requirements?*

"slide 550"

Two "paths" lead to business process reengineering:

- 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 reengineer the human operations within and without the enterprise. "slide 551"
- 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 reengineering directives a need arises to also deploy new software, for which requirements therefore have to be enunciated.

One way or the other, business process reengineering is an integral component in deploying new computing systems.

3.7.2 Overview of BPR Operations

"slide 552"

We suggest six domain-to-business process reengineering operations:

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 introduction of some new and removal of some old work practices (relating to *human behaviours*); and
6 related *scripting.*

3.7.3 BPR and the Requirements Document "slide 553"

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[14]

We shall thus, in Sects. 3.7.4–3.7.8, 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.

"slide 555"

As the BPR requirements "rebuilds" the business process description part of the domain description[15], 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.

"slide 556"

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}. Or 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[16]

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.

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. We may then decide to either develop entirely new descriptions of the new business processes, i.e., actually prescriptions for the business reengineered 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 $\mathcal{D}_{BP}$.

"slide 558"

[14] "slide 554"

[15] — Even if that business process description part of the domain description is "empty" or nearly so!

[16] "slide 557"

3.7.4 Intrinsics Review and Replacement

"SLIDE 559"

Characterisation 73 (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. ▪

"slide 560"

Example. 1 – **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.

3.7.5 Support Technology Review and Replacement

"SLIDE 561"

Characterisation 74 (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. ▪

"slide 562"

Example. 2 – **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. The following business process reengineering 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*

"slide 563"

copies (not the required computing system). The above, somewhat construed example, shows the "division of labour" between the contemplated (required, desired) computing system (the "machine") and the "business reengineered" persons authorised to print and possess confidential documents. "slide 564"

It is implied in the above that the reengineered handling of documents would not be feasible without proper computing support. Thus there is a "spill-off" from the business reengineered world to the world of computing systems requirements. •

3.7.6 Management and Organisation Reengineering "SLIDE 565"

Characterisation 75 (Management and Organisation Reengineering) By *management and organisation reengineering* 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. ▪

"slide 566"

Example. 3 – **Management and Organisation Reengineering:** *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.*

"slide 567"

The 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.

"slide 568"

The business process reengineering 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 reengineered procurement process. •

3.7.7 Rules and Regulations Reengineering "SLIDE 569"

Characterisation 76 (Rules and Regulation Reengineering) By *rules and regulations reengineering* 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. "slide 570"

Example. 4 – **Rules and Regulations Reengineering:** *Assume now, due to reengineered support technologies, that interlock signalling can be made magnitudes safer than before, without interlocking. Thence it makes sense to reengineer a domain rule of 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 reengineered rule is subsequently made into a domain requirements, namely that the software system for interlocking is bound by that rule. •

3.7.8 Script Reengineering "slide 571"

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. "slide 572"

Characterisation 77 (Script Reengineering) By *script reengineering* 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. ▪ "slide 573"

More to come "slide 574"

More to come "slide 575"

More to come

3.7.9 Human Behaviour Reengineering

"SLIDE 576"

Characterisation 78 (Human Behaviour Reengineering) By *human behaviour reengineering* 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. ▪

"slide 577"

Example. 5 – **Human Behaviour Reengineering:** *A company has experienced certain lax attitudes among members of a certain category of staff. The progress of certain work procedures therefore is reengineered, 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 reengineering leads to a number of requirements for computerised monitoring of the two groups of staff. •

3.7.10 Discussion: Business Process Reengineering

"SLIDE 578"

Who Should Do the Business Process Reengineering?

It is not in our power, as software engineers, to make the kind of business process reengineering 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 reengineering has been made, it then behooves the client stakeholders to further decide whether the BP reengineering shall imply some requirements, or not.

Who Should Do the Business Process Reengineering?[17]

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. **??**.

General[18]

Business process reengineering 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

[17] "SLIDE 579"

[18] "SLIDE 580"

functions, products or geographical regions. This often means that business processes "cut across" vertical units. Others are "horizontally" structured 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 refer to Appendix Sect. M.5 starting on Page 467 for an example related to the *business process re-engineering* topic.

3.8 Requirements Terminology

"SLIDE 581"

TO BE WRITTEN

We refer to Appendix Sect. M.6 starting on Page 469 for an example related to the *requirements terminology* topic.

"slide 582"
"slide 583"

3.9 Requirements Modelling

"SLIDE 584"

3.9.1 Aims & Objectives

The **aims** of the requirements modelling stage of requirements engineering are to finalise a **delineation** of the span for the requirements, to ensure that the requirements **relate strongly** to the domain, and to help **secure** that that the client gets the **right software.** The **objectives** of the requirements modelling stage of requirements engineering are to **develop** a proper requirements prescription, one that is well **analysed** ("theoretised"), and one that leads to **efficient** and **correct** software.

3.9.2 Requirements Facets

"SLIDE 585"

It has been highlighted, significantly, above that there are three facets to requirements modelling: modelling those, the **domain** requirements which can be expressed sôlely using terms from the domain, modelling those, the **interface** requirements which can be expressed using terms from both the domain and the machine, and modelling those, the **machine** requirements which can be expressed sôlely using terms from the machine. For interface requirements, by *"using terms both from the domain and the machine"* we mean: that the "smallest" requirements prescription units contain both kinds, that is, they are used in an inseparable way.

We refer to Appendices N–P (Pages 471–491) for an example related to the *requirements modelling* topic.

3.9.3 Domain Requirements

"slide 586"

A domain requirements prescription is that part of the overall requirements prescription which can be expressed solely using terms from the domain description. Thus to construct the domain requirements prescription all we need is collaboration with the requirements stakeholders (who, with the requirements engineers, developed the BPR) and the possibly rewritten (BPR influenced) domain description — which we shall refer to as a domain requirements document.

"slide 587"

The domain span is usually much, or just, larger than the domain implied by the requirements one is about to construct. Therefore a first domain-to-requirements operation is to **project** onto the first step domain requirements only those phenomena and concepts of the domain which are required. The resulting domain requirements projection often defines a much–too-general set of phenomena and concepts which, although they are needed, may not be needed in the generality given, hence they are **instantiated** to more specific ones. The resulting domain requirements projection & instantiation often defines its phenomena and concepts too non–deterministically, hence "excess" non–determinism is more more **determinate**. Often computing and communication allows for phenomena and concepts which are not feasible in the domain, hence we **extend** the emerging domain requirements projection & instantiation & determination with such domain–lile phenomena and concepts which are feasible. Finally, requirements engineering may take place in a context where more than one group is developing requirements, but for "more-or-less" distinct "areas", hence we suggest to **fit** the various emerging domain requirements into a consistent and cohenrent set of such.

"slide 588"

"slide 589"

Thus we end up with the following domain-to-requirements steps:

We shall next treat each of these steps in some detail. But first we must now understand that whereas where the narratives and the formalisations of the domain description characterised to — "spoke of" — phenomena in the domain these, when projected, instantiated, etc., become concepts of the machine !

Domain Requirements Projection[19]

By a **domain projection** we mean *a subset of the domain description, one which leaves out all those domain phenomena and concepts entities, functions,*

[19] "slide 590"

events, and (thus) behaviours that the stakeholders do not wish represented by the machine. The resulting document is a **partial domain requirements prescription**.

We refer to Appendix Sect. N.1 Pages 471–472 for an example related to the *projection* topic.

Guidelines[20]

Domain phenomena and concepts that are "removed" from the emerging partial domain requirements prescription may leave uses of these phenomena and concepts elsewhere in the remaining, emerging prescription undefined. Thus they rally cannot be fully removed. The requirements modeller must make a judicious decision as how to express these phenomena and concepts, usually in some furher instantiated and more deterministic form as outlined next.

Discussion[21]

Thus projection is not an automatable operation. It is one which is carried out jointly between the appropriate stakeholders and the domain engineers where the latter edits the emerging outcome of projection.

'Removing' phenomena and concepts that are unwanted in the emerging partial domain requirements prescription was first indicated in a domain acquisition step where it was left as markings in a domain description document; it is now "completed" by proper deletions and editing of both narratives and formalisations.

Discussion of Support Example[22]

Domain Requirements Instantiation[23]

By **domain instantiation** we mean *a refinement of the partial domain requirements prescription, resulting from the projection step, in which the refinements aim at rendering the entities, functions, events, and (thus) behaviours of the partial domain requirements prescription more concrete, more specific.* Instantiations usually render these concepts less general. The resulting document is a **partial domain requirements prescription**.

We refer to Appendix Sect. N.2 Pages 473–476 for an example related to the *instantiation* topic.

[20] "SLIDE 591"
[21] "SLIDE 592"
[22] "SLIDE 593"
[23] "SLIDE 594"

Guidelines[24]

Discussion[25]

Discussion of Support Example[26]

Domain Requirements Determination[27]

By **domain determination** we mean a *refinement of the partial domain requirements prescription, resulting from the instantiation step, in which the refinements aim at rendering the entities, functions, events, and (thus) behaviours of the partial domain requirements prescription less non-determinate, more determinate.* Instantiations usually render these concepts less general. The resulting document is a **partial domain requirements prescription**.

We refer to Appendix Sect. N.3 Pages 476–479 for an example related to the *determination* topic.

Guidelines[28]

Discussion[29]

Discussion of Support Example[30]

Domain Requirements Extension[31]

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

We refer to Appendix Sect. N.4 Pages 479–480 for an example related to the *extension* topic.

[24] "SLIDE 595"
[25] "SLIDE 596"
[26] "SLIDE 597"
[27] "SLIDE 598"
[28] "SLIDE 599"
[29] "SLIDE 600"
[30] "SLIDE 601"
[31] "SLIDE 602"

Guidelines[32]

Discussion[33]

Discussion of Support Example[34]

Domain Requirements Fitting[35]

The issue of requirements fitting arises when two or more software development projects are based on what appears to be the same domain. The problem then is to harmonise the two or more software development projects by harmonising, if not too late, their requirements developments. The result is usually a three or more documents two or more specific requirements documents and a document, perhaps more than one, consisting of requirements that are common to two or more of the specific requirements.

We refer to Appendix Sect. N.5 Pages 480–481 for an example related to the *fitting* topic. "slide 607"

We thus assume that there are n domain requirements developments, d_{r_1}, d_{r_2}, ..., d_{r_n}, being considered, and that these pertain to the same domain — and can hence be assumed covered by a same domain description. "slide 608"

By requirements fitting we mean *a harmonisation of $n > 1$ domain requirements that have overlapping (common) not always consistent parts and which results in n 'modified and partial domain requirements', and m 'common domain requirements' that "fit into" two or more of the 'modified and partial domain requirements'.* "slide 609"

By a *modified and partial domain requirements* we mean a domain requirements which is short of (that is, is missing) some description parts: text and formula. By a *common domain requirements* we mean a domain requirements. By the m common domain requirements parts, $cdrs$, *fitting into* the n modified and partial domain requirements we mean that there is for each modified and partial domain requirements, $mapdr_i$, an identified subset of $cdrs$ (could be all of $cdrs$), $scdrs$, such that textually conjoining $scdrs$ to $mapdr$ can be claimed to yield the "original" d_{r_i}.

A Requirements Fitting Procedure[36]

Requirements fitting consists primarily of a pragmatically determined sequence of analytic and synthetic ('fitting') steps. It is first decided which n domain requirements documents to fit. Then a 'manual' analysis is made of the selected, n domain requirements. During this analysis tentative common domain requirements are identified. It is then decided which m common

[32] "SLIDE 603"
[33] "SLIDE 604"
[34] "SLIDE 605"
[35] "SLIDE 606"
[36] "SLIDE 610"

domain requirements to single out. This decision results in a tentative construction of n modified and partial domain requirements. An analysis is made of the tentative modified and partial and also common domain requirements. A decision is then made whether to accept the resulting documents or to iterate the steps above.

Requirements Fitting Verification[37]

Domain Requirements Consolidation[38]

After projection, instantiation, determination, extension and fitting, it is time to review, consolidate and possibly restructure (including re-specify) the domain requirements prescription before the next stage of requirements development.

"slide 613"

We refer to Sect. 2.9.10 Pages 101–102 (Consolidation of Domain Facets Description) for remarks similar to the below.

The many previous domain requirements stages may have yielded descriptions which, typically at the formal level, does not reveal how it all "hangs together". In such cases, and in general, consolidation of these domain requirements documentaton stages could take the following forms.

With each potential management unit we associate a process or an indexed set of two or more processes, usually an indeterminate number. Such management units will usually involve entities and behaviours — whether staff of entity behaviours. Usually type definitions and axioms (about sorts) and value definitions of auxiliary and well-formedness functions about values can be kept separate from the process definitions. The entity processes usually take, as arguments, the entity whose time-wise behaviour and interaction with oother entity processes is being requirements modelled.

"slide 614"

With each structural component of the organisation we associate one or more channels, or vector or array or tensor (or ...) indexed sets of channels.

MORE TO COME

• • •

We refer to Appendix N (Pages 471–481) for an example related to the *domain requirements* topic.

"slide 616"

"SLIDE 615"

[37] "SLIDE 611"

[38] "SLIDE 612"

3.9.4 Interface Requirements

"SLIDE 617"

By an **interface requirements** we mean *a requirements prescription which refines and extends the domain requirements by considering those requirements of the domain requirements whose entities, operations, events and behaviours are* **"shared"** *between the domain and the machine (being requirements prescribed).*

Domain/Machine Sharing[39]

'Sharing' means *(a) that an* **entity** *is represented both in the domain and "inside" the machine, and that its machine representation must at suitable times reflect its state in the domain; (b) that an* **operation** *requires a sequence of several "on-line" interactions between the machine (being requirements prescribed) and the domain, usually a person or another machine; (c) that an* "slide 619"
event *arises either in the domain, that is, in the environment of the machine, or in the machine, and need be communicated to the machine, respectively to the environment; and (d) that a* **behaviour** *is manifested both by actions and events of the domain and by actions and events of the machine.* "slide 620"

So a systematic reading of the domain requirements shall result in an identification of all shared entities, operations, events and behaviours. "slide 621"

Each such shared phenomenon shall then be individually dealt with: **entity sharing** shall lead to interface requirements for data initialisation and refreshment; **operation sharing** shall lead to interface requirements for interactive dialogues between the machine and its environment; **event sharing** shall lead to interface requirements for how such event are communicated between the environment of the machine and the machine. **behaviour sharing** shall lead to interface requirements for action and event dialogues between the machine and its environment. "slide 622"

• • •

We shall now illustrate these domain interface requirements development steps with respect to our ongoing example. "slide 623"

Interface Modalities

"slide 624"

Data Communication

"slide 625"

Digital Sampling

"slide 626"

Tactile: Keyboards &c.

"slide 627"

Visual: Displays, Lamps, &c.

"slide 628"

Audio: Voice, Alarms, &c.

"slide 629"

Other Sensory Interface Modalities

"slide 630"

Entities: Domain/Machine Sharing

"slide 631"

Data Intialisation

"slide 632"

Data Refreshment

"slide 633"

Functions: Domain/Machine Sharing

"slide 634"

Interactive Human/Machine Dialogues

"slide 635"

Interactive Machine/Machine Protocols

"slide 636"

Events: Domain/Machine Sharing

"slide 637"

Human/Machine/Human Events

"slide 638"

Machine/Machine Events

"slide 639"

Other Context/Machine Events

"slide 640"

Behaviour: Domain/Machine Sharing

"slide 641"

Human/Machine/Human Behaviours

"slide 642"

Machine/Machine Behaviours

"slide 643"

Other Context/Machine Behaviours

• • •

We refer to Appendix O (Pages 483–485) for an example related to the *interface requirements* topic.

"slide 644"

"slide 645"

[39] "SLIDE 618"

3.9.5 Machine Requirements

"SLIDE 646"

Characterisation 79 (Machine Requirements) By *machine requirements* we understand those requirements that can be expressed sôlely in terms of (or with prime reference to) machine concepts. ∎

An Enumeration of Issues[40]

"slide 648"

Performance Requirements[41]

Characterisation 80 (Performance Requirements) By *performance requirements* we mean 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. ∎

"slide 651"

Pragmatically speaking, performance requirements translate into financial resources spent, or to be spent.

Example. 6 – **Performance Requirements: Timetable System Users and Staff — Narrative Prescription Unit:** *We continue Example* **??** *on page ??. 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.* •

"slide 652"

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.

"slide 653"

Example. 7 – **Performance Requirements: Timetable System Users and Staff:** *We continue Example 20 and Example 6. In Example 20 the sharing of the timetable between users and staff was expressed parametrically.*

$system(tt) \equiv client(tt) \,[\!]\, staff(tt)$

$client$: $TT \to$ **Unit**
$client(tt) \equiv$ **let** q:*Query* **in let** $v = \mathcal{M}_q(q)(tt)$ **in** $system(tt)$ **end end**

$staff$: $TT \to$ **Unit**
$staff(tt) \equiv$
let u:*Update* **in let** $(r,tt') = \mathcal{M}_u(u)(tt)$ **in** $system(tt')$ **end end**

"slide 654"

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.

"slide 655"

```
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()
```

"slide 656"

[40] "slide 647"
[41] "slide 650"

client: i:CIdx → **out** *ct*[*i*] **in** *tc*[*i*] **Unit**
client(i) ≡ **let** *qc:Query* **in** *ct*[*i*]!$\mathcal{M}_q$(*qc*) **end** *tc*[*i*]?;*client(i)*

staff: **Unit** → **out** *st* **in** *ts* **Unit**
staff() ≡ **let** *uc:Update* **in** *st*!$\mathcal{M}_u$(*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**

"slide 657"

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.* •

"slide 658"

The next example, Example 8, 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 + data communication equipment facilities versus time and space.

"slide 659"

Example. 8 – **Performance Requirements of Storage and Speed for *n*-Transfer Travel Inquiries:** *We continue Example* **??** *on page ??. 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.* •

Other Resource Consumption[42]

Dependability Requirements[43]

"slide 662"

To properly define the concept of *dependability* we need first introduce and define the concepts of *failure*, *error*, and *fault*.

"slide 663"

Characterisation 81 (Failure) A machine *failure* occurs when the delivered service deviates from fulfilling the machine function, the latter being what the machine is aimed at [182]. ▪

"slide 664"

Characterisation 82 (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 [182]. ∎

"slide 665"

Characterisation 83 (Fault) The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a *fault* [182]. ∎

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,[44] we stop at *the cause which is intended to be prevented or tolerated.*

"slide 666"

Characterisation 84 (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 [182]. ∎

"slide 667"

Characterisation 85 (Dependability) *Dependability* is defined as the property of a machine such that reliance can justifiably be placed on the service it delivers [182]. ∎

We continue, less formally, by characterising the above defined concepts [182]. "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.

"slide 668"

[42] "SLIDE 660"

[43] "SLIDE 661"

[44] 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."

Having already discussed the "threats" aspect, we shall therefore discuss the "means" aspect of the *dependability tree*.

- Attributes:
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Integrity
  - ⋆ Reliability
  - ⋆ Safety
  - ⋆ Security

"SLIDE 669"

- Means:
  - ⋆ Procurement
    - · Fault prevention
    - · Fault tolerance
  - ⋆ Validation
    - · Fault removal
    - · Fault forecasting
- Threats:
  - ⋆ Faults
  - ⋆ Errors
  - ⋆ Failures

"SLIDE 670"

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*. Our increasing dependence on computing systems in the end brings in the need for *fault tolerance*. We refer to special texts [134, 230, 233] on the above four topics.

"SLIDE 671"

Characterisation 86 (Dependability Attribute) By a *dependability attribute* we shall mean either one 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. ▪ "slide 672"

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 [134].

Accesability Requirements[45]

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, into an implementation, some *"coin tossers"* might never get access to the machine.

"slide 674"

Characterisation 87 (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). ∎

"slide 675"

Example. 9 – **Accessibility Requirements: Timetable Access:** *Based on Examples 20 on page 172 and* **??** *on page ??, we can express: The timetable (system) shall be inquirable by any number of users, and shall be updateable 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[46]

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"!

"slide 677"

Characterisation 88 (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. ∎

"slide 678"

Example. 10 – **Availability Requirements: Timetable Availability:** *We continue Examples 20,* **??**, *and 9: 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 —*

[45] "SLIDE 673"
[46] "SLIDE 676"

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[47]

Characterisation 89 (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[48]

Characterisation 90 (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. 11 – **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[49]

Characterisation 91 (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. 12 – **Timetable Safety:** *Mean time between failures whose resulting downtime is more than 4 hours shall be at least 120 days.* •

Security Requirements[50]

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 *unauthorised user* (for a resource) is anyone who is not authorised access to that resource.

[47] "SLIDE 679"
[48] "SLIDE 680"
[49] "SLIDE 681"
[50] "SLIDE 682"

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.)!

"slide 683"

Characterisation 92 (Security) A system being *secure* — in the context of a machine being dependable — means that an *unauthorised 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 unauthorised access to computing and/or handling of information (i.e., data). ▪

"slide 684"

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!

"slide 685"

Example. 13 – **Security Requirements: Timetable Security:** *We continue Examples 20, ??, 9, and 10. 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.* •

"slide 686"

Example. 14 – **Security Requirements: A Hospital Information System:** *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 postprocessing (i.e., evaluation) of such "panic button" accesses can take take place.* •

Robustness Requirements[51]

Characterisation 93 (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.

• • •

[51] "SLIDE 687"

Maintenance Requirements[52]

Characterisation 94 (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*. ▪

"slide 689"

Maintenance of building, mechanical, electrotechnical 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. We refer to the very beginning of Sect. **??** for a proper definition of what we mean by software.

Adaptive Maintenance Requirements[53]

Characterisation 95 (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). ▪

"slide 691"

Example. 15 – **Adaptive Maintenance Requirements: Timetable System:** *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.* •

Corrective Maintenance Requirements[54]

Characterisation 96 (Corrective Maintenance) By *corrective maintenance* we understand such maintenance which corrects a software error. ▪

Example. 16 – **Corrective Maintenance Requirements: Timetable System:** *Corrective maintenance shall be done remotely: from a developer site, via secure Internet connections.* •

[52] "SLIDE 688"
[53] "SLIDE 690"
[54] "SLIDE 692"

Perfective Maintenance Requirements[55]

Characterisation 97 (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. ▪

Example. 17 – **Perfective Maintenance Requirements: Timetable System:** *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[56]

Characterisation 98 (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[57]

Characterisation 99 (Extensional Maintenance) By *extensional maintenance* we understand such maintenance which adds new functionalities to the software, i.e., which implements additional requirements. ▪

Example. 18 – **Extensional Maintenance Requirements: Timetable System:** *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.* "SLIDE 696" *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.* •

"slide 697"

• • •

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.

[55] "SLIDE 693"
[56] "SLIDE 694"
[57] "SLIDE 695"

Platform Requirements[58]

Characterisation 100 (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.

Characterisation 101 (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*. ▪

"slide 699"

Example. 19 – **Platform Requirements: Space Satellite Software:** *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[59]

Characterisation 102 (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[60]

Characterisation 103 (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[61]

Characterisation 104 (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. ▪

[58] "SLIDE 698"
[59] "SLIDE 700"
[60] "SLIDE 701"
[61] "SLIDE 702"

Demonstration Platform Requirements[62]

Characterisation 105 (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[63]

Example 19 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.

Documentation Requirements[64]

We refer to Chap. **??** for a thorough treatment of the kind of documents that normally should result from a proper software development project. And we refer to overviews of these documents as they pertain to domain engineering (Sects. **??** and **??**), requirements engineering (Sects. **??** and **??**), and software design (Sect. **??**).

Characterisation 106 (Documentation Requirements) By *documentation requirements* we mean requirements of any of the software documents that together make up software (cf. the very first part of Section **??**): (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*. 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*. ∎

"slide 706"

"slide 707"

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.

[62] "SLIDE 703"
[63] "SLIDE 704"
[64] "SLIDE 705"

Fault Analysis[65]

In pursuing the formulation of requirements for dependable systems it is often required that the requirements engineer perform what is called *fault analysis*. A particular approach is called *fault tree analysis*. Dependable systems development is worth a whole study in itself. So we cut short our mentioning of this very important subject by emphasising its importance and otherwise referring the reader to the relevant literature. A good introduction to the issues of safety analysis in the context of formal techniques is [183]. We strongly recommend this source — also for references to "the relevant literature".

Source: Kirsten Mark Hansen

This example was kindly provided by Kirsten Mark Hansen. It is edited from Chap. 4 of her splendid PhD Thesis [95].

Fault tree analysis is one of the most widely used safety analysis techniques. It presumes a hazard analysis, which has revealed the catastrophic system failures [231]. For each system failure, it deduces the possible combinations of component failures which may cause this failure.

Fault tree analysis is a graphical technique, in which fault trees are drawn using a predefined set of symbols. The graphic representation may be appealing, but it also causes the fault trees to be big and unmanageable.

A fault tree analysis is closely related to a system model, as the different levels of system abstraction are reflected in the tree. The root corresponds to a system failure, and the immediate causes of this failure are deduced as logical combinations (conjunction and disjunction) of failures of the system components.

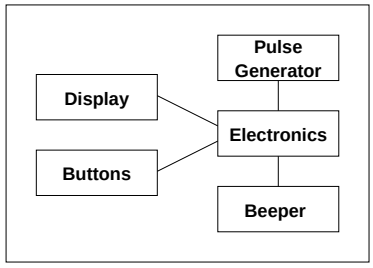


Fig. 3.1. Alarm clock

Figure 3.1 shows an alarm clock which is built from the components: A display, some buttons, a pulse generator, some electronics, and a beeper. A fault tree analysis of the failure of the alarm clock failing to activate the alarm is presented in Fig. 3.2. The causes of this failure may either be the beeper

[65] "SLIDE 708"

failing; the pulse generator not generating the right pulses; the electronics failing, either by not activating the beeper or by not registering the buttons pushed; or the buttons failing. We assume that the display has no impact on this failure. Each of the components may again be considered as a system consisting of components. The analysis stops when a component is considered to be atomic.

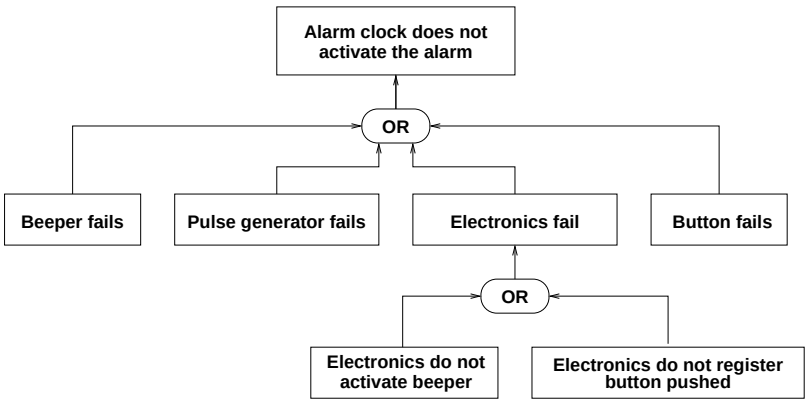


Fig. 3.2. Fault tree for an alarm clock

A minimal cut set of a fault tree is the smallest combination of component failures which, if they all occur, will cause the top event to occur. Smallest means that if just one component failure is missing from the cut set, then the top event does not occur. The fault tree in Fig. 3.2 has five minimal cut sets, each containing a leaf as its only element. Two fault trees are defined to be equivalent if they have the same minimal cut sets.

A concept related to the minimal cut set is the minimal path set. A minimal path set is the smallest combination of primary events whose non-occurrence assures the non-occurrence of the top event. The fault tree in Fig. 3.2 has one minimal path set containing all the leaves of the tree.

As fault trees are used to analyse safety-critical systems for safety, it is important that they have an unambiguous semantics. We will later illustrate that often this is not the case. The aim of this chapter is therefore to assign a formal semantics to fault trees, and to illustrate how such a semantics may be used in the formulation of system safety requirements. The main reference in this chapter is the fault tree handbook [237], which has been used intensively in defining the syntax and the semantics of fault trees.

Some of the nodes of a fault tree are called events by safety analysts. In order to avoid confusion, we stress that we use the safety analysis meaning of the term event, namely the occurrence of a system state, rather than the computer science meaning of an event, namely a transition between two states.

Fault Tree Syntax

A fault tree analysis consists of building fault trees by connecting nodes from a predefined set of node symbols by directed edges. Edges are directed in the sense that for a given node the child nodes are called input nodes, and the father node is called the output node. The node symbols are divided into three groups: event symbols, gate symbols, and transfer symbols. We describe each of the groups separately.

Event Symbols

The event symbols are divided into primary event symbols and intermediate event symbols, where the primary event symbols are the leaves of the tree.

Primary events:: The primary event symbols are shown in Fig. 3.3.

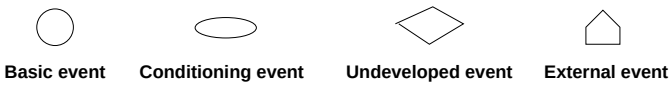


Fig. 3.3. Primary event symbols

- **Basic event:** A basic event contains an atomic component failure.
- **Conditioning event:** Conditioning events are most often used as input to PRIORITY AND and to INHIBIT gates. When used as input to a PRIORITY AND gate, the condition event is used to specify the order in which the input events must occur.
- **Undeveloped event:** An undeveloped event contains a non-atomic component failure. The fault tree is not developed further from this event due to lack of time, money, interest, etc. The component is not atomic, so it is possible later to develop the event further.
- **External event:** The content of an external event is not a failure, but something that is expected to occur in the system environment.

Intermediate events:: The intermediate events consist only of one symbol, namely the intermediate event symbol, a rectangular box. Intermediate events cannot be found in the leaves of a fault tree.

Gate Symbols

Gate symbols designate Boolean combinators. They are shown in Fig. 3.4.

OR gate:: The informal description of an OR gate is that the output event occurs when at least one of the input events occur. An OR gate may have any number of input events. Fig. 3.2 is an example of a fault tree with two OR gates.

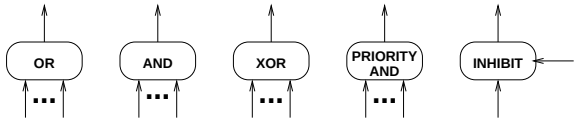


Fig. 3.4. Gate symbols

AND gate:: The informal description of an AND gate is that the output event occurs only when all the input events occur. An AND gate may have any number of input events. Fig. 3.5 is an example of a fault tree with an AND gate. This fault tree states that all brakes on a bike have failed, when both the foot brake "and" the hand brake have failed.

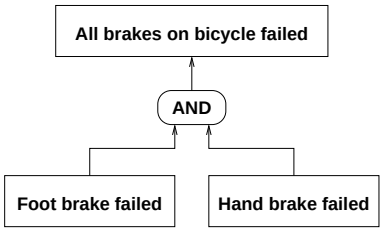


Fig. 3.5. Fault tree with AND gate

INHIBIT gate:: An INHIBIT gate is a special case of an AND gate. An INHIBIT gate has one input event and one condition. The output event occurs when both the input event occurs and the condition is satisfied. In the fault tree in Fig. 3.6, the chemical reaction goes to completion when all reagents and the catalyst are present.

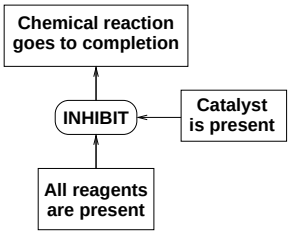


Fig. 3.6. Fault tree with INHIBIT gate

XOR (exclusive or) gate:: The output event occurs only if exactly one of the input events occurs. If more than one of the input events occur, the output

event does not occur. An XOR gate may have any number of input events. Fig. 3.7 shows a fault tree with an XOR gate. This fault tree states that a train is not at the platform, either if the train is ahead of the platform, or if it is behind the platform. Since the (specific) train cannot be at both places it is exactly at one or the other.

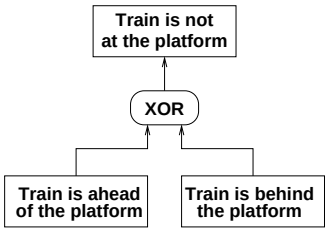


Fig. 3.7. Fault tree with XOR gate

PRIORITY AND gate:: The output event occurs only if all the input events occur, and if they occur in a left to right order. A PRIORITY AND gate may have any number of input events. The fault tree in Fig. 3.8 states that the door is locked if the door is (first) closed and the key is (then) turned.

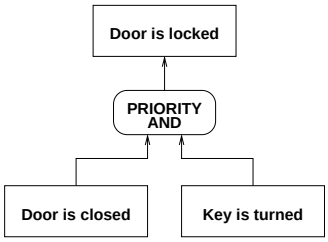


Fig. 3.8. Fault tree with PRIORITY AND gate

Fault Tree Semantics

In our attempt to give fault trees a formal semantics, we discovered that the accepted informal descriptions of fault tree gates are ambiguous, allowing several very different interpretations. For instance, the semantics of an AND gate is defined as [237]: "The output fault occurs only if all the input faults occur"; but what does this mean? Does it mean that all input faults have to occur at the same time, or does it mean that all input faults have to occur, but that they need not overlap in time? Does the output fault necessarily occur when the input faults occur? Clearly such uncertainty is not desirable

when dealing with safety-critical systems. In this section we therefore give fault trees a formal semantics.

Primary Events: The first step in assigning a formal semantics to fault trees is to define a model of the system on which the fault tree analysis is performed. Assume that we have defined such a model and that it takes the form of system states evolving over time. (This "system states evolving over time" model is the basis for the duration calculus [228, 229]. We refer to Chap. 15, Vol. 2, for an introduction to the duration calculus.) Using this model, we interpret the leaves of a fault tree, i.e., the basic events, the undeveloped events, the conditioning events, and the external events as duration calculus formulas. Such a formula may *for instance* be:

- the constants *true*, *false*
- occurrence of a state P, i.e., $\lceil P \rceil$
- occurrence of a transition to state P, i.e., $\lceil \neg P \rceil ; \lceil P \rceil$
- lapse of a certain time, i.e., $\ell \geq (30 + \epsilon)$, or
- a limit of some duration, i.e., $\int P \leq 4 \times \epsilon$.

We consider the distinction between the different types of leaves to be pragmatic, describing why the fault tree has not been developed further from the that leaf, and therefore we make no distinction between the types of the leaves in the semantics.

Intermediate Events: The semantics of intermediate events is defined by the semantics of the leaves, edges, and gates in the subtrees in which the intermediate events are the roots. Intermediate events are merely names for the corresponding subtrees.

Edges

We now consider the meaning of the intermediate event, A, connected to an event, B, by an edge, see Fig. 3.9.

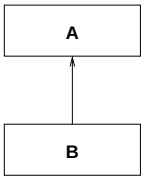


Fig. 3.9. Fault tree with no gates

Assume that the semantics of B is B. We then define the semantics of A to be

$$A = B,$$

i.e., as logical identity, meaning that the system failure A occurs when the failure B occurs. This semantics is pessimistic in the sense that it assumes that if something has a possibility of going wrong, then it does go wrong. Informal readings of fault trees often state that it is not mandatory that A holds when B holds [237, 234], which is formalised as $A \Rightarrow B$. This semantics allows an optimistic interpretation of fault trees in the sense that a system failure may be avoided if the operator intervenes fast enough, has enough luck, etc. In our opinion, speed, luck, and the like should not be parameters in safety-critical systems, and we have therefore rejected this semantics. Another issue is whether A and B occur at the same time or if there is some delay from the occurrence of B to the occurrence of A. Often there will be such a delay, but we have refrained from modelling it, as this again would give the impression that once B has occurred there is a chance that A can be prevented.

Gates

We now consider the semantics of intermediate events connected to other events through gates.

OR:: For the fault tree in Fig. 3.10 assume that the semantics of $\mathrm{B}_1, \ldots, \mathrm{B}_n$ is $B_1, \ldots, B_n$. We define the semantics of A to be

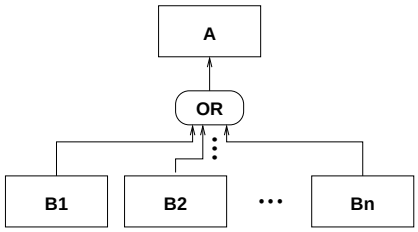


Fig. 3.10. Fault tree with OR gate

$$A = B_1 \vee \ldots \vee B_n,$$

i.e., A holds iff either B_1 or $\ldots$ or B_n holds. This interpretation shows that an OR gate introduces single point failure. The failure occurs if just one of the formulas holds.

AND:: In the fault tree in Fig. 3.11 assume that the semantics of $\mathrm{B}_1, \ldots, \mathrm{B}_n$ is $B_1, \ldots, B_n$.

We then define the semantics of A to be

$$A = B_1 \wedge \ldots \wedge B_n,$$

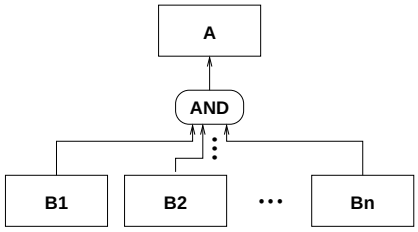


Fig. 3.11. Fault tree with AND gate

i.e., A holds iff $B_1, \ldots, B_n$ hold simultaneously. We have considered a more liberal interpretation of AND gates in which B_1 to B_n need not hold simultaneously, namely $A = \Diamond B_1 \wedge \ldots \wedge \Diamond B_n$. This has been rejected since this formula "remembers any occurrence of a B_i", such that if B_2 becomes true 1 year after B_1, and B_3 becomes true 3 years after B_2, and ..., then A holds. This is clearly not the intended meaning of an AND gate.

INHIBIT:: We only consider INHIBIT gates in which the condition is *not* a probability statement. According to the fault tree handbook, [237], the fault

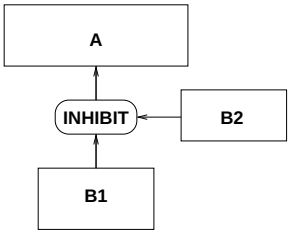


Fig. 3.12. Fault tree with INHIBIT gate

tree in Fig. 3.12 reads: "If the output A occurs then the input B_1 has occurred in the past while condition B_2 was true". We interpret this to be if A holds, then both B_1 and B_2 hold, i.e., as an AND gate with B_1 and B_2 as inputs. Thus the semantics of an INHIBIT gate is

$$A = B_1 \wedge B_2.$$

XOR:: A fault tree with an XOR gate is given in Fig. 3.13 (left). According to the fault tree handbook, [237], this tree may be drawn as in the same figure to the right, in which "Not both B_1 AND B_2" is a necessary condition for the root formula to hold. As for the INHIBIT gate we interpret the condition "Not both B_1 AND B_2" as a leaf which should also hold. By interpreting "Not both B_1 AND B_2" as $\neg(B_1 \wedge B_2)$, we obtain the semantics

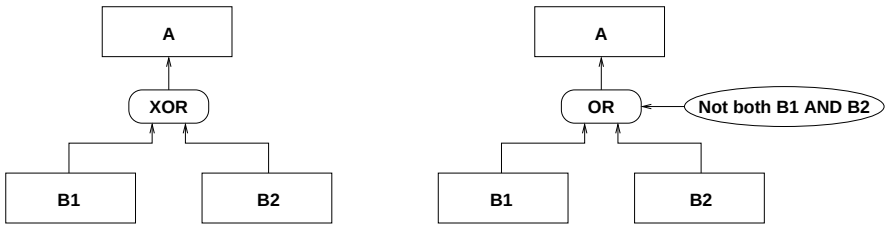


Fig. 3.13. Fault trees. Left with XOR gate. Right with OR gate and Condition

$$A = (B_1 \vee B_2) \wedge \neg(B_1 \wedge B_2)$$

which may be rewritten to

$$A = (B_1 \wedge \neg B_2) \vee (\neg B_1 \wedge B_2).$$

This generalises to

$$\begin{aligned} A = & (B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1})). \end{aligned}$$

PRIORITY AND:: A fault tree with a PRIORITY AND gate is given in Fig. 3.14. The informal semantics states that the output event occurs if all

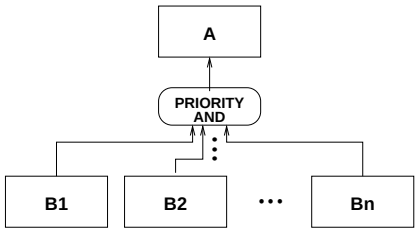


Fig. 3.14. Fault tree with PRIORITY AND gate

the input events occur in a left to right order. Assuming that $B_1, \ldots, B_n$ have the semantics $B_1, \ldots, B_n$, we define the semantics of A to be

$$A = B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n) \ldots).$$

Refinement

As we saw in the beginning of this section, fault trees are often used to model system failures at different abstraction levels, Figs. 3.1 and 3.2.

If there is a shift in abstraction levels in a fault tree, we require that it is indicated by a dashed line connecting a root in one tree (concrete model) to a leaf in another tree (abstract model) as in Fig. 3.15. (In that figure we have "abstracted" the Boolean combinators: BCx, BCy, BCz are either of OR, AND, PRIORITY AND, INHIBIT or XOR.)

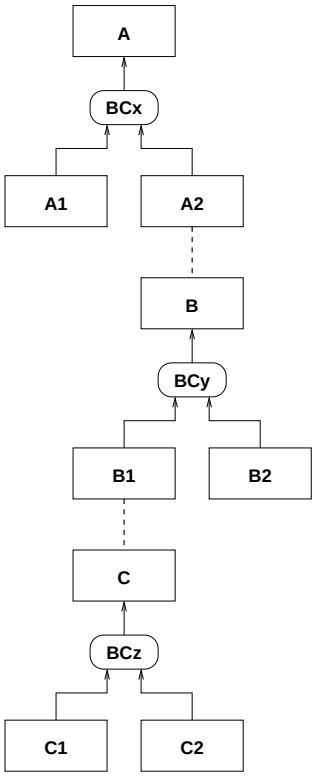


Fig. 3.15. Fault tree with three abstraction levels

We consider such a dashed line to connect two fault trees, where each of the fault trees is defined in one system model. For each of the fault trees, the semantics of the tree is defined as described previously. The dashed line indicates a refinement relation between the systems for which the fault tree analysis is performed. Consider the simple fault tree in Fig. 3.16 in which A has the semantics A and is defined by the state functions Var_a, and B has the semantics B and is defined by the state functions Var_b.

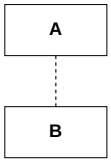


Fig. 3.16. Simple fault tree with refinement

Assume that Var_a is a subset of Var_b. As a fault tree describes the undesired system behaviours, i.e., $\neg A$ for the abstract system, and $\neg B$ for the concrete system, the refinement relation between the two systems is given by

$$\neg B \Rightarrow \neg A$$

where $\neg A$ is interpreted over the domain Var_b. It is equivalent to

$$A \Rightarrow B.$$

If the state functions of the concrete system, B, relate to the state functions of the abstract system, A, through a transformation ϕ, then the refinement relation under transformation is interpreted over $Var_a \cup Var_b$ and is given by

$$\phi \wedge \neg B \Rightarrow \neg A$$

which is equivalent to

$$\phi \wedge A \Rightarrow B.$$

In Fig. 3.15, assume that A_1 has the semantics A_1, A_2 has the semantics A_2, B_1 has the semantics B_1, etc., then it may be deduced from the semantics of fault trees that A has the semantics $A_1 \vee A_2$, B has the semantics $B_1 \wedge B_2$ and C has the semantics $C_1 \vee C_2$. Further assume that the fault tree containing the A's is defined in model 1, which has the state functions Var_a; the fault tree containing the B's is defined in model 2, which has the state functions Var_b; and the fault tree containing the C's is defined in model 3, which has the state functions Var_c. Further assume that Var_b relates to Var_a through the transformation ϕ, and that Var_b is a subset of Var_c. The proof obligations that arise from the fault tree are therefore

$$\phi \wedge A_2 \Rightarrow B_1 \wedge B_2$$

which is interpreted over $Var_a \cup Var_b$, and

$$B_1 \Rightarrow C_1 \vee C_2$$

in which B_1 is interpreted over Var_c.

In program development the chain of refinements is from *true* towards $false$. For fault trees the refinements from the top towards the bottom are from $false$ towards true. The reason for this is that fault trees specify the undesired system states, whereas program development specifies the desired system states.

Deriving Safety Requirements

Traditionally, fault trees are used to analyse existing system designs with regard to safety. Instead of first developing a design, and then performing a safety analysis, we propose that the design and the safety analysis should be developed concurrently, thereby making it possible to let the fault tree analysis influence the design. In order to do this, the fault tree analysis and the system design must at each abstraction level use the same system model. Given a common model, the system safety requirements may be deduced from the fault tree analysis. Safety requirements derived in this way can be used during system development in order to validate the design, but they can also be used in a constructive way by influencing the design. We illustrate this below.

For each fault tree in which the root is interpreted as S, the system should be designed such that S never occurs, i.e., the safety commitment which the system should implement is

$$\Box\neg S.$$

If we have n fault trees in which the roots are interpreted as $S_1, \ldots, S_n$, the safety commitment which may be deduced from these fault trees is

$$\Box\neg S_1 \wedge \ldots \wedge \Box\neg S_n,$$

i.e., the system should ensure that no top event in any fault tree ever holds. This corresponds to combining the trees by an OR gate.

Deriving Component Requirements

Assume that we have a fault tree like the one in Fig. 3.9, and that the safety commitment is $\Box\neg A$. As the fault tree has the semantics $A = B$, $\Box\neg A$ must be implemented by implementing $\Box\neg B$. If the fault tree contains gates, the derived specifications depend on the types of the gates.

OR gates: The fault tree in Fig. 3.10 has the semantics $A = B_1 \vee \ldots \vee B_n$. In order to make the system satisfy the safety commitment $\Box\neg A$, we must implement

$$\Box\neg(B_1 \vee \ldots \vee B_n)$$

or equivalently

$$\Box\neg B_1 \wedge \ldots \wedge \Box\neg B_n.$$

This formula expresses that the system only satisfies its safety commitments if all its components satisfy their local safety commitments. Now suppose that the designer cannot control the first component, i.e., it is outside the scope of the design of that component whether it satisfies B_1 or not. Making the safe choice of B_1 being *true* causes $\Box\neg B_1$ to be *false*, which trivially implies that the safety commitment is violated. Making the tacit assumption that B_1 is *false* is a very poor judgment, which essentially ignores the results of the safety analysis. The only reasonable option is to weaken the specification. We *assume* that the behaviour of the first component never satisfies B_1, i.e., that $\Box\neg B_1$ is *true*. To make the design team aware of this assumption, we add it to the environment assumptions. So, if the design involved the assumptions Asm before this design step, we have assumptions $Asm \wedge \Box\neg B_1$ afterwards. The specification of the requirements $Asm \Rightarrow Com$ has thus been weakened, to $Asm \wedge \Box\neg B_1 \Rightarrow Com$, and the designer should alert the appropriate persons as to this change in assumptions. Many design errors are located on interfaces. The interface is made clearer and the likelihood of errors is reduced if one has an explicit list of assumptions and adds to this list as the system development progresses.

AND gates: Bear in mind that the fault tree in Fig. 3.11 has the semantics $A = B_1 \wedge B_2 \wedge \ldots \wedge B_n$ and assume that the safety commitment is $\Box\neg A$. This safety commitment corresponds to specifying that the components never satisfy their duration formulas at the same time, i.e.,

$$\Box\neg(B_1 \wedge B_2 \wedge \ldots \wedge B_n).$$

One way to implement this is to implement the stronger formula

$$\Box\neg B_1 \vee \Box\neg B_2 \vee \ldots \vee \Box\neg B_n,$$

i.e., to design at least one of the components such that it always satisfies its local safety commitment. Often, the designer does not control all the input components of an AND gate. For such components a safe approach is to assume the worst case, namely that the component is in a critical state and thereby contributes to violation of the safety commitment. Let us for instance assume in the case of the fault tree in Fig. 3.11 that the first component is uncontrollable. The worst case is that the component satisfies B_1, i.e., that

$$\Box\neg(true \wedge B_2 \wedge \ldots \wedge B_n)$$

meaning that the designer has to implement

$$\Box\neg(B_2 \wedge \ldots \wedge B_n).$$

If it is not possible to make such an implementation, a final solution is to assume that B_1 always is false, and then see to it that this is implemented in

another component by adding it to the list of assumptions, i.e., if we had the assumptions Asm before this design step, we have the assumptions $Asm \wedge \Box\neg B_1$ afterwards. One should, at some point, arrive at a conjunction of B_i's which can be used in the design. Otherwise we must conclude that the system is inherently unsafe. If the design relies on the absence of only one B_i, it is a design which is vulnerable to single point failures.

INHIBIT gates: As the semantics of INHIBIT gates are the same as for AND gates, the derivations of safety requirements for INHIBIT gates are the same as for AND gates.

XOR gates: An event A which is output from an XOR gate which has $B_1, \ldots, B_n$ as input events has the semantics

$$\begin{aligned} A = & (B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1})). \end{aligned}$$

A safety commitment $\Box\neg A$ must be implemented by

$$\begin{aligned} \Box\neg \; & ((B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1}))) \end{aligned}$$

which is equivalent to

$$\begin{aligned} \Box \; & ((\neg B_1 \vee B_2 \vee \ldots \vee B_n) \\ & \wedge \\ & \vdots \\ & \wedge \\ & (\neg B_n \vee B_1 \vee \ldots \vee B_{n-1})). \end{aligned}$$

This means that the designer has to make the design such that for every observation interval either all the input events are false, or at least two of the input events are true at the same time, i.e.,

$$\Box(\text{All-false} \vee \text{Two-true})$$

where

$$\text{All-false} \equiv \neg(B_1 \vee \ldots \vee B_n),$$

$$\begin{aligned}\text{Two-true} \equiv\ & ((B_1 \wedge B_2) \vee \ldots \vee (B_1 \wedge B_n) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge B_1) \vee \ldots \vee (B_n \wedge B_{n-1})).\end{aligned}$$

Now assume that one of the components is uncontrollable, i.e., the designer cannot control whether, e.g., B_1 is true or not. If the Exclusive Or (XOR) gate has more than two input events, then the design may be made such that two of the other input events are always true. If this is not possible (perhaps because the XOR gate only has two input events), the designer either has to assume that B_1 is false and then make the design such that the rest of the B's are always false, or assume that B_1 is true and then make the design such that one of the other input events is always true. In either case, he has to make the rest of the design team aware of the assumption by adding it to the list of assumptions about the environment. So, if the design involved the assumptions, Asm, before this design step, and if the designer assumes that B_1 is always true, then the assumptions are $Asm \wedge \Box B_1$ after this design step, and if he assumes that B_1 is always false, then the assumptions are $Asm \wedge \Box\neg B_1$. In principle the designer may also assume that whenever one of the B's which he can control is true then B_1 is also true, and whenever all the B's he can control are false, then B_1 is also false. As B_1 is implemented in another component than the rest of the B's, and as A occurs if the components are out of synchronization just once, we do not recommend this solution.

PRIORITY AND gates: The fault tree in Fig. 3.8 has the semantics $A = B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n)\ldots)$. If the safety commitment is $\Box\neg A$, the designer must implement

$$\Box\neg(B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n)\ldots)).$$

This may either be done by making the design such that the B_i's do not occur in the specified order or such that one of the B_i's does not occur at all, i.e.,

$$\Box\neg B_1 \vee \Box\neg B_2 \vee \ldots \vee \Box\neg B_n.$$

If one of the B_i's, e.g., B_1 is uncontrollable, the worst case is that it does not satisfy its local safety commitment, i.e., that B_1 is true. The designer therefore assume that B_1 is true and attempts to make the design such that

$$\Box\neg(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n)\ldots)$$

holds. If it is not possible to make such a design, the last opportunity is to assume that B_1 always is false, and then to assure that this is implemented in another component by adding it to the list of assumptions about the environment, i.e., the assumptions become $Asm \wedge \Box\neg B_1$.

Refinement

Assume that we have a fault tree in which an event A, with the semantics A, is refined by an event B, with the semantics B, see Fig. 3.16. Further, assume that the refinement relation has been verified, and that the safety commitment is $\Box\neg A$. As part of the refinement relation is $A \Rightarrow B$, then $\Box\neg A$ must be implemented by implementing $\Box\neg B$.

Conclusion

In this section we have given fault trees a duration calculus semantics, and we have defined how a fault tree analysis may be used to derive safety requirements, both for systems and for system components. The semantics is compositional such that the semantics of the root is expressed in terms of the leaves. The derivation of safety requirements follows the structure of the fault tree and results in safety requirements for the system's components. This derivation of safety requirements for components should stop when the deduced requirements may be implemented using well-established methods, e.g., formal program development techniques for software components.

As for all other techniques, this technique for deriving safety requirements is no better than the people who use it. An error in the fault tree analysis is reflected in the safety requirements, and the system failures for which a safety analysis has not been performed are not extracted as requirements. If, however, we compare this method to the existing ways of deriving safety requirements, namely by more or less structured brainstorming, we think that this method is an improvement.

In terms of safety requirements, a minimal cut set corresponds to the smallest set of components which, if they do not fulfill their safety requirements, will cause the system not to fulfill its safety requirements. If the minimal cut set only contains one component, then the system is vulnerable to single point failure.

A minimal path set corresponds to the smallest set of components which must fulfill their safety requirements in order that the system fulfill its safety requirements. If all components have to fulfill their safety requirements, i.e., the cardinality of the minimal path set equals the number of components, then the system is unsafe, as it may fail if just one of the components fails.

We have defined the semantics in duration calculus, but other temporal logics, like e.g., TLA$^+$ [131, 132, 155] and linear temporal logic [146, 147, 148], could also have been applied. The important thing is that the logic is capable of expressing both the semantics of the intermediate events, based on the structure of the fault tree, and the semantics of the leaves.

Fault trees are sometimes used in a probabilistic analysis of safety. We have not given semantics to fault trees with probabilistic figures, as this requires a deeper knowledge of stochastic processes than we have. The foundation for assigning a formal semantics to such trees has been established in [141], in

which a probabilistic duration calculus based on discrete Markov chains [235] is defined and in [232] which defines a conversion algorithm from fault trees to Markov chains. The idea, in probabilistic duration calculus, is that, given an initial probability distribution, i.e., the probability that the system is initially in a state v, and a transition probability matrix, i.e., the probability that the system enters state u, given that the system is in state v, then it is possible to calculate the probability that the system is in a certain state at a discrete time t.

• • •

We refer to Appendix P (Pages 487–491) for an example related to the *machine requirements* topic.

3.9.6 Discussion: Machine Requirements "slide 709"

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. "slide 710" "slide 711" "slide 712"

3.10 Requirements Verification "slide 713"

TO BE WRITTEN

3.11 Requirements Validation "slide 714"

TO BE WRITTEN

3.12 Requirements Satisfiability and Feasibility "slide 715"

TO BE WRITTEN

3.13 Requirements Theory Formation "slide 716"

TO BE WRITTEN

3.14 Requirements Engineering Process Graph "SLIDE 717"

TO BE WRITTEN

"slide 718"

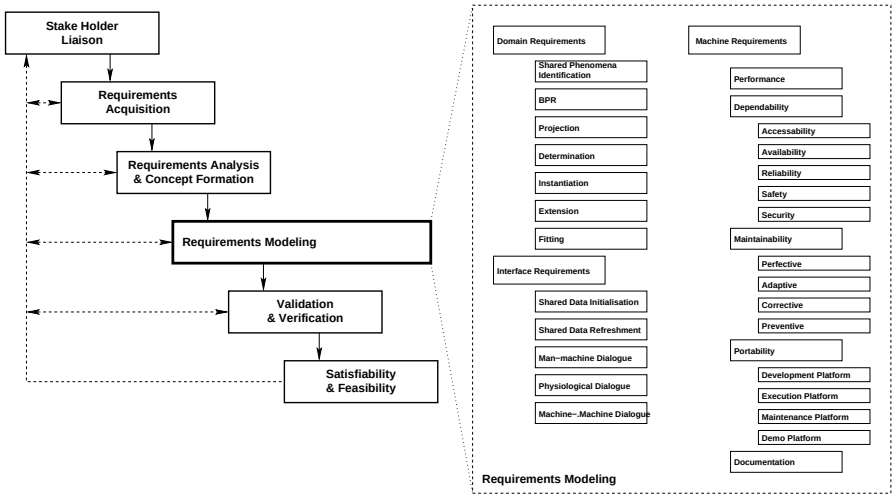


Fig. 3.17. Requirements engineering process graph

3.15 Requirements Engineering Documents "SLIDE 719"

TO BE WRITTEN

"slide 720"

3.15.1 Requirements Prescription Documents

TO BE WRITTEN

"slide 721"

1 Stakeholders
2 The Acquisition Process
   (a) Studies
   (b) Interviews
   (c) Questionnaires
   (d) Indexed Description Units
3 Rough Sketches (Eurekas, IV)
4 Business Process Re-engineering
   - Sanctity of Intrinsics
   - Support Technology
   - Management and Organisation
   - Rules and Regulations
   - Human Behaviour
   - Scripting

5 Terminology
6 Facets: PG.:722
(a) Domain Requirements
- Projection
- Determination
- Instantiation
- Extension
- Fitting

(b) Interface Requirements
- Shared Phenomena and Concept Identification
- Shared Data Initialisation
- Shared Data Refreshment
- Man-Machine Dialogue
- Physiological Interface
- Machine-Machine Dialogue

(c) Mach. Reqs. PG.:723
- Performance
  - ⋆ Storage
  - ⋆ Time
  - ⋆ Software Size
- Dependability
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Reliability
  - ⋆ Robustness
  - ⋆ Safety
  - ⋆ Security
- Maintenance
  - ⋆ Adaptive
  - ⋆ Corrective
  - ⋆ Perfective
  - ⋆ Preventive
- Platform (P)
  - ⋆ Development P
  - ⋆ Demonstration P
  - ⋆ Execution P
  - ⋆ Maintenance P
- Doc. Reqs.
- Other Requirements

(d) Full Requirements Facets Documentation

3.15.2 Requirements Analysis Documents

TO BE WRITTEN

2 Requirements Analysis and Concept Formation
(a) Inconsistencies
(b) Conflicts
(c) Incompletenesses
(d) Resolutions
3 Requirements Validation
(a) Stakeholder Walkthroughs
(b) Resolutions
4 Requirements Verification
(a) Theorem Proofs
(b) Model Checks
(c) Test Cases and Tests
5 Requirements Theory
6 Satisfiability and Feasibility
(a) Satisfaction: correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
(b) Feasibility: technical, economic, BPR "slide 724"

3.16 Summary "SLIDE 725"

TO BE WRITTEN

MORE TO COME

3.17 Exercises

Exercise 19. kap3.xs.1:
Solution 19 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 20. kap3.xs.2:
Solution 20 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 21. kap3.xs.3:
Solution 21 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 22. kap3.xs.4:
Solution 22 Vol. II, Page 534, suggests a way of answering this exercise.

Exercise 23. kap3.xs.5:
Solution 23 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 24. kap3.xs.6:
Solution 24 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 25. kap3.xs.7:
Solution 25 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 26. kap3.xs.8:
Solution 26 Vol. II, Page 535, suggests a way of answering this exercise.

"slide 726"

4

Software Design

"slide 728"

4.1 Discussion of the Software Design Concept

"slide 729"

4.2 Stages of Software Design

"slide 730"

"slide 731"

4.2.1 An Overview of "What to Do ?"

"slide 732"

[1] Software Design Information

"slide 733"

[2] Software Design Stakeholders

"slide 734"

[3] Software Design Acquisition

"slide 735"

[4] Software Design Analysis and Concept Formation

"slide 736"

[5] Software Design Options

"slide 737"

[6] Software Design Terminology

"slide 738"

[7] Software Design Modelling

"slide 739"

[8] Software Design Verification

"slide 740"

[9] Software Design Validation

"slide 741"

[10] Software Design Release, Transfer & Maintenance

"slide 742"

[11] Software Design Documentation

"slide 743"

4.2.2 A Summary Enumeration

4.3 Software Design Information

"slide 744"

We have earlier, as mentioned above, extensively (Pages 6–24) covered the general issues of informative documents. The reader is strongly encouraged to review those pages, Sect. 1.5. Suffice it here to emphasize the following.

Current Situation: Cf. Sect. 1.6.3, Page 9

As mentioned in Sect. 1.6.3 on page 9 the context in which the software design starts must be emphasized. That context invariably includes the existence of a requirements prescription.

MORE TO COME

"slide 745"

Needs and Ideas: Cf. Sect. 1.6.4, Pages 9–10

TO BE WRITTEN

"slide 746"

Concepts and Facilities: Cf. Sect. 1.6.5, Pages 10–11

TO BE WRITTEN

"slide 747"

Scope and Span: Cf. Sect. 1.6.6, Page 11

TO BE WRITTEN

"slide 748"

Assumptions and Dependencies: Cf. Sect. 1.6.7, Pages 11–12

TO BE WRITTEN

"slide 749"

Implicit/Derivative Goals: Cf. Sect. 1.6.8, Page 12

TO BE WRITTEN

"slide 750"

Synopsis:

TO BE WRITTEN

"slide 751"

Software Development Graphs: Cf. Sect. 1.6.10, Pages 13–15

"slide 752"

TO BE WRITTEN

Resource Allocation: Cf. Sect. 1.6.11, Pages 15–16

TO BE WRITTEN

"slide 753"

Budget (and Other) Estimates: Cf. Sect. 1.6.12, Page 16

TO BE WRITTEN

"slide 754"

Standards Compliance: Cf. Sect. 1.6.13, Pages 16–19

TO BE WRITTEN

"slide 755"

Contracts and Design Briefs: Cf. Sect. 1.6.14, Pages 19–23

TO BE WRITTEN

"slide 756"

MORE TO COME

4.4 Software Design Stakeholders

"SLIDE 757"

TO BE WRITTEN

4.5 Software Design Acquisition

"SLIDE 758"

TO BE WRITTEN

4.6 Software Design Analysis and Concept Formation

"SLIDE 759"

TO BE WRITTEN

4.7 Software Design Options

"SLIDE 760"

TO BE WRITTEN

4.8 Software Design Terminology

"SLIDE 761"

TO BE WRITTEN

"slide 762"
"slide 763"

4.9 A Domain Example

"SLIDE 764"

Example. 20 – Determination of Airline Timetable Queries: *To exemplify this rough-sketch domain (to) requirements operation we first present a rough domain description, then the "more deterministic" domain requirements prescription. (i) A rough-sketch timetable-querying domain description is: There is given a further undefined notion of timetables. There is also given a concept of querying a timetable. A timetable query, abstractly speaking, denotes (i.e., stands for) a function from timetables to results. Results are not further defined.* "SLIDE 765" *(i) A rough-sketch timetable querying domain requirements description is: There are given notions of departure and arrival times, and of airports, and of airline flight numbers.* "SLIDE 766" *Determination of Airline Timetable Queries, I*

scheme *TI_TBL_2* =
  extend *TI_TBL_1* **with**
    class
      type
        T, *An*, *Fn*
    end

"SLIDE 767" *A 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.* "SLIDE 768" *Determination of Airline Timetable Queries, II*

scheme *TI_TBL_3* =
  extend *TI_TBL_2* **with**
    class
      type
        $JR' = (T \times An \times T)^*$
        $JR = \{|\ jr{:}JR' \bullet \textbf{len}\ jr \geq 2 \wedge ...\ |\}$
        $TT = Fn\ \overrightarrow{m}\ JR$
    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. "SLIDE 769" *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.

"SLIDE 770"

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 Example **??** *on page ??.*

You may skip the rest of the example, its formalisation, if your reading of this book does not include the various formalisations. "SLIDE 771" *First, we formalise the syntactic and the semantic types:*

Determination of Airline Timetable Queries, III

```
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:

"SLIDE 772" *Determination of Airline Timetable Queries, IV*

```
scheme TI_TBL_3U =
  extend TI_TBL_3 with
    class
      value
        M_q: Query → QU
        M_q(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:

"SLIDE 773"

Determination of Airline Timetable Queries, V

```
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.* "SLIDE 774" *Before we had:*

Determination of Airline Timetable Queries, VI

```
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 = Mq(q)(tt) in system(tt) end end)
    ∏ (let u:Update in let (r,tt′) = Mu(q)(tt) in system(tt′) end end)
```

"SLIDE 775"

Or, for use in Example 7:

```
  system(tt) ≡ client(tt) ∏ staff(tt)

  client: TT → Unit
  client(tt) ≡
    let q:Query in let v = Mq(q)(tt) in system(tt) end end

  staff: TT → Unit
  staff(tt) ≡
    let u:Update in let (r,tt′) = Mu(q)(tt) in system(tt′) end end
```

•

4.10 Software Design Modelling

"slide 776"

"slide 777"

4.10.1 Architectural Design

Introduction "slide 778"

The requirements prescriptions cover four areas: business process reengineering, domain requirements, interface requirements and machine requirements. Only the last three sets of requirements influence the design of the computing system, i.e., the machine. The business process reengineering prescriptions are meant to materially influence the behaviour of people using that machine. In this chapter we shall primarily focus on architectural consequences of some machine requirements.

"slide 779"

This section is predominantly based on formalisations. The reader who wishes to study this book only informally is thus left to not be able to enjoy one of the many advantages of using formal specifications. The "closer" one gets to actual program implementation, the more one has to express the software design in a programming notation. The formalisms used are, however, simple and mostly based on the CSP of RSL/CSP. We covered that subset of RSL in Vol. 1, Chap. 21. So get used to it!

Initial Domain Requirements Architecture "slide 780"

Usually, when developing a domain requirements, we can formalise, incrementally, the resulting domain requirements, not just by their properties, but we can also give model-oriented prescriptions. The same holds for some interface requirements. "slide 781"But for some other interface requirements, and for most, if not all, machine requirements, we cannot formalise the properties asked for, but one can formalise their possible, claimed implementation, that is, an architectural software design.

Model-oriented requirements prescriptions amount to partial software architecture specifications. Let us "slowly" unfold such a software architecture specification.

"slide 782"

Example. 21 – Component Diagram of a Simple Timetable System: *We refer to Example 20. We diagram that formalisation as shown in Fig. 4.1. The arrows here were not thought of as channels.*

"slide 783"

A Simple Timetable System We can model the diagram of Fig. 4.1 on the next page by:

system: TT → **Unit**
system(tt) ≡ *client(tt)* ∥ *staff(tt)*

client: TT → **Unit**

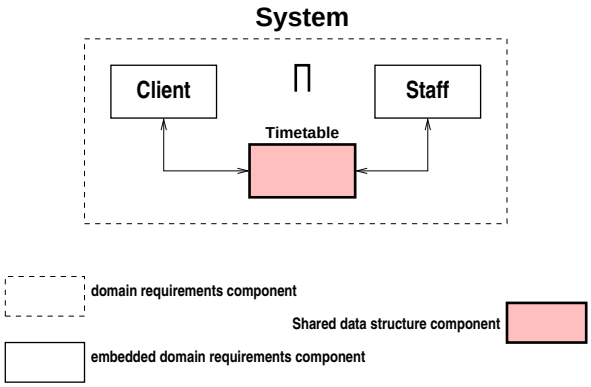


Fig. 4.1. Timetable application components

client(*tt*) ≡ **let** *q:Query* **in let** $v = \mathcal{M}_q(q)(tt)$ **in** *system*(*tt*) **end end**

staff: $TT \to$ **Unit**
staff(*tt*) ≡
  let *u:Update* **in**
  let $(r,tt') = \mathcal{M}_u(u)(tt)$ **in** *system*(tt') **end end**

"SLIDE 784"

Both the airline client and the airline staff make use of the (shared phenomenon) airline timetable. It is therefore to be considered a data structure that is shared not only between the domain and the machine, but also between client and staff. •

"SLIDE 785"

We call the component shown in Fig. 4.1, a domain requirements *component*. In this case, we may claim that it consists of three embedded such.

We now, increasingly, since software design is our subject, turn to model-oriented specifications. In this section, we almost exclusively develop and enrich process-oriented specifications.

"SLIDE 786"

Example. 22 – Formal Model of Simple Timetable System Process: *We refer to Example 21 (above), but assume arrows as designating channels. Each of the three subcomponents of Fig. 4.1 are now considered to be separately evolving behaviours, that is, processes.*

"SLIDE 787" *Simple Timetable System Processes*

channel
  ctt:QU, *ttc:VAL*, *stt:UP*, *ts:RES*
value
  system: $TT \to$ **Unit**

system(tt) ≡ *client()* ‖ *time_table(tt)* ‖ *staff()*

client: **Unit** → **out** *ct* **in** *tc* **Unit**
client() ≡ **let** *qc:Query* **in** $ctt!\mathcal{M}_q(qc)$ **end** *ttc?;client()*

staff: **Unit** → **out** *st* **in** *ts* **Unit**
staff() ≡ **let** *uc:Update* **in** $st!\mathcal{M}_u(uc)$ **end let** *res* = *ts?* **in** *staff()* **end**

time_table: TT → **in** *ctt,stt* **out** *ttc,tts* **Unit**
time_table(tt) ≡
  let *qf* = *ctt?* **in** *ttc!qf(tt); time_table(tt)* **end**
  [] **let** *uf* = *st?* **in let** *(tt′,r)=uf(tt)* **in** *ts!r; time_table(tt′)* **end end**

"SLIDE 788"

We can read the framed formulas above aloud for the reader who otherwise cannot read these formulas: There are two connections, two interfaces, between the client and the time_table, One in each direction. Similarly, there are two connections, two interfaces, between the staff and the time_table, One in each direction. The system behaviour is the parallel composition of three behaviours: client, staff and time_table. Only the time_table possesses the timetable. All three behaviours, client, staff and time_table, are cyclic: Recurse indefinitely.

"SLIDE 789"

The client behaviour sends query requests to the time_table behaviour and awaits response before recycling. The staff behaviour sends update requests to the time_table behaviour and awaits response before recycling. In either case, the client and staff behaviours — before resuming their behaviour — ignore the response.

The time_table behaviour, as an obedient server, is ready, in each round, each cycle, to engage in an event with either the client or the staff behaviours. The time_table behaviour expresses this by an external nondeterministic choice ([]). We refer to Example 7 on page 136. •

• • •

In Examples **??** and **??** we exemplified aspects of interface requirements for the example of the present chapter. One could claim, and with some justification, that what Example **??** illustrated could as well be said to constitute a software design specification. Other than this fleeting reference to interface software design, we shall not, in this chapter, illustrate interface design.

Initial Machine Requirements Architecture "SLIDE 790"

In general we can always claim that one can continue, after such software design which "implements" domain requirements concerns, with software design concerned with implementing machine requirements.

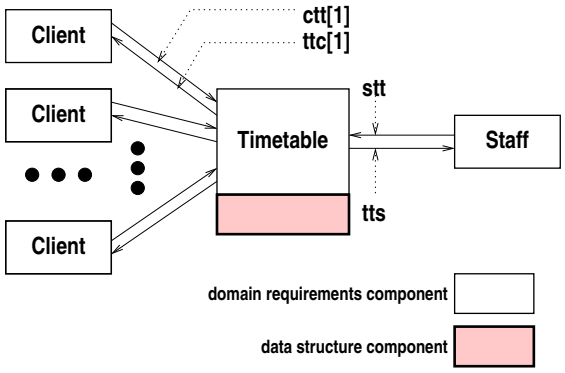


Fig. 4.2. Timetable application components

"SLIDE 791"

Example. 23 – Component Diagram and Formal Model of a Timetable System: *We shall exemplify a software design which can be said to implement domain requirements. We refer to Example 7, but now, as it is, with n client processes (Fig. 4.2).* "SLIDE 792"

This will be the last figure in Chap. **??** *in which we explicitly show data structure components.*

"SLIDE 793" *Timetable Application Components*

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*) ≡ ‖{*client*(*i*)|*i*:*CIdx*} ‖ *time_table*(*tt*) ‖ *staff*()

"SLIDE 794"

The individual processes are defined next.

Timetable Application Components

client: *i*:*CIdx* → **out** *ct*[*i*] **in** *tc*[*i*] **Unit**
client(*i*) ≡ **let** *qc*:*Query* **in** *ct*[*i*]!$\mathcal{M}_q$(*qc*) **end** *tc*[*i*]?;*client*(*i*)

staff: **Unit** → **out** *st* **in** *ts* **Unit**
staff() ≡ **let** *uc*:*Update* **in** *st*!$\mathcal{M}_u$(*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**

We refer to Example 7 on page 136. "SLIDE 795" *For the readers who cannot read the formulas we "read" them "aloud": There is an index set of client (names),* CIdx. *For each* client *there is a separate pair of channels,* ct[i] *and* tc[i], *i.e., means of communicating with the* time_table *process. This is just a generalisation of the model given in Example 22. And, as in that model, there is a pair of channels,* st *and* ts, *between the* staff *and the* time_table *process. The* system *is the parallel, comprehended composition of as many* client *processes as there are elements in* CIdx, *composed, also in parallel, with one* staff *and one* time_table *process.* "SLIDE 796" *The only difference between the model of the present example and that of Example 22 on page 176 is, first, that communications between* client *processes and the* time_table *process takes place over indexed channels, and, second, that the* time_table *process nondeterministically, externally, is ready to engage with any* client *process.* •

"SLIDE 797"

Thus we enter, in the continuing examples of this section, a stage where we are now concerned with the implementation, i.e., the software architectural issues as determined by machine requirements. But first let us exemplify an issue of analysis.

Analysis of Some Machine Requirements "SLIDE 798"

We have chosen, in this section, to focus on just a few machine requirements issues. Although it may not be realistic of actual developments, it is sufficiently illustrative of what goes on in actual software design developments concerned with the implementation of machine requirements.

The machine requirements issues selected are *performance, availability, accessibility* and *adaptive maintainability.*

Performance

"SLIDE 799"

We refer to Examples 6 and 7. The performance issue chosen was the simple one of making sure that n clients could be online simultaneously. And the software design issue that we wish to look at is how to design a machine requirements component, or a set of such components, that separate out from the time_table process the choice among n client processes and one staff process.

Availability

"SLIDE 800"

We refer to Example 10. The time_table process does not guarantee "fair" choice between handling input from clients and input from staff processes (f.

nondeterministic external choice ([]) of Example 23). The internal nondeterministic choice that we are referring to above is that between the timetable process's handling of the n:Index client process inputs and its handling of the staff process inputs. The RSL semantics of [] allows one side of the [] operator, i.e., one operand, to be selected indefinitely. "Fairness" is an issue of making sure that both process operands of [] are "inquired" as to willingness to "progress".[1] That is, both the client and the staff processes should be given a fair chance of communicating with the time_table process.

Accessibility

"SLIDE 801"

We refer to Example 9 on page 140. The current software architecture can be said to prescribe strict, mutually exclusive serialisation of client and staff processes wrt. time_table process. This may be acceptable for zero time processing, but it is not acceptable for time-consuming timetable operations (like, for example, connection queries)! "Small, quick" query processing, such as journey, could be interleaved with the processing of "large, time-consuming" queries, such as connections.

Adaptive Maintainability

"SLIDE 802"

We refer to Example 15. We focus on the direct channels between time_table and the client and staff processes. These direct channels, if also implemented as channels ("ad verbatum"), might hinder the development (i.e., refinement) of several distinct implementations of the client process.

Prioritisation of Design Decisions "SLIDE 803"

The design decisions include a prioritisation of which machine requirements shall first, then subsequently, determine program organisation design decisions. Our example prioritisation is:

- First *performance*, then
- *availability*, then
- *accessibility*, and finally
- *adaptive maintainability*.

"SLIDE 804"We do not motivate the specific prioritisation. Specific machine requirements prioritise one requirements over another. There may be various reasons for a prioritisation. These prioritisation reasons are usually given in informative documents. We shall not go into a discussion of machine requirements prioritisation.

[1] Our reasoning would be the same for the nondeterministic internal choice operator ⌈⌉.

Corresponding Designs "slide 805"

So, on one hand, there is the issue of deciding how to suggest a software architecture design (i.e., "a component or two") which implement[s] a machine requirement — a design which can, eventually, be shown, somehow, to satisfy the (usually property-oriented) machine requirements. And, on the other hand, there is the issue of choosing a way in which to represent this design decision.

"slide 806"

The emphasis in this section is on expressing designs in terms of diagrams: (i) with some boxes designating domain requirements entities and functions, i.e., domain requirements components; (ii) with other boxes designating machine requirements design choices, i.e., machine requirements components and (iii) with arrows connecting these boxes designating means of invoking functions in respective components.

Design Decision wrt. Performance

"slide 807"

We illustrated in Sect. 4.10.1 on page 179 just one aspect of the larger machine requirements concern: performance. It was based on Examples 6 and 7. We shall now illustrate a design decision which is then recorded in the form of a "boxes and arrows" diagram. Throughout this and the next design decisions (Sects. 4.10.1–4.10.1) we shall use this rather informal mode of "design and reasoning", which is based on understanding boxes as usually cyclic processes and arrows as one- or two-directional input/output event channels. That is, we shall omit the crucial specification of the protocols which monitor and control events (synchronisations and communication "along" the arrows) in (and between) processes (i.e., the boxes).

"slide 808"

"slide 809"

Example. 24 – Performance Component Design: *We refer to Fig. 4.2. We observe that it is the time_table process which performs the choice between n client requests and one staff request. We decide to "factor" this aspect — of choice — out from the time_table process, and into one machine requirements component, cli_mpx (for client multiplexor component), and one machine requirements component, cli_stf_mpx (for client/staff multiplexor component). We refer to Fig. 4.3 on the following page. For the moment we might consider the two machine requirements components as a pair of "inseparable, back-to-back" components. Informally speaking, what goes on in the boxes and on the channels is as follows: Assume a client, say client i, wishes to communicate a query to the timetable. The client multiplexor then decides between this, client i, request and possibly other such, client $j, i \neq j$, requests, and selects one, say client j. The client multiplexor now passes that request on to the client staff multiplexor. Assume that the staff, also at such a time as client requests $i, \ldots, j, \ldots, k$ are issued, issues an update request.*

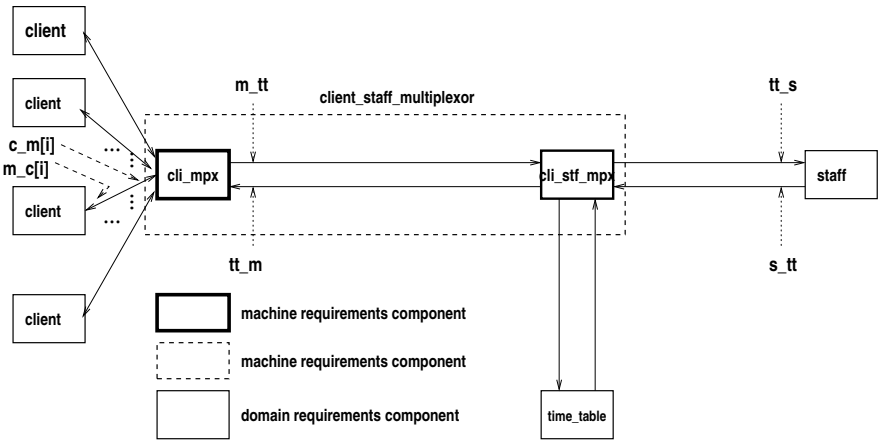


Fig. 4.3. Client/staff multiplexor components

Now the client multiplexor decides on one of these two: client j query and the staff update. Assume that the client multiplexor decides to choose the staff request. That staff request is passed on to the timetable, is serviced, and a result (response) is returned to the client staff multiplexor, which returns it to the staff. Right after this the client staff multiplexor may choose to service the client (j) query. It could choose a further staff update should such a request have been issued "right on the heels" of a former and serviced update request. Now we assume that the client staff multiplexor services the ("outstanding") client (j) query. It is passed on to the timetable and is serviced, and a result (value) is returned to the client staff multiplexor, which returns it to the client multiplexor (whereupon the client staff multiplexor is freed to service staff updates). The client multiplexor returns the result of the client j query to that client and the client multiplexor is then freed to service either "outstanding" or new client requests. Notice that at this stage of design we have chosen to let the two multiplexor processes await completion, by them, of the most recently serviced request. •

Design Decision wrt. Availability

"SLIDE 810"

Example. 25 – Availability Component Design: *First, we refer to Example 10 on page 140 and then to the discussion of Sect. 4.10.1 on page 179. To resolve nondeterminism that unfairly chooses among client query requests, as in the client multiplexor, or as among a chosen client request and a staff request, we must modify both multiplexors. Our design choice is to "equip" both multiplexors each with their own arbitration procedure embodied in an arbiter component.*

"SLIDE 811"

Thus these arbiters shall secure a "more fair" choice — perhaps "less non-deterministic" — within and between the two categories of users. A solution could be to have an internal clock (or "bit") secure alternate sampling of client *queries (to the client staff arbiter), respectively secure alternate sampling of chosen* client *and* staff *requests (to the timetable process). We refer to Fig. 4.4.*

•

"SLIDE 812"

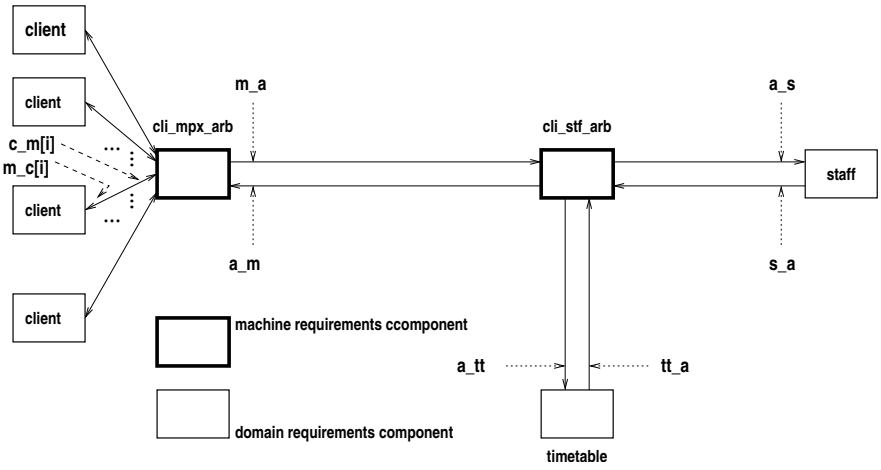


Fig. 4.4. Arbiter component

"SLIDE 813"

Please observe across the two examples just given, Example 24 and Example 25, that the boxes of a former design decision change "nature" (i.e., meaning) as a result of a subsequent design decision. It is because we wish to have the design freedom (i.e., the design options) to let this happen often, that we refrain at this thereby informal step from detailing the "inner workings" of the boxes etc. Now we need only fully specify (incl., possibly formalise) a last design decisions accumulated step.

Design Decision wrt. Accessibility

"SLIDE 814"

Example. 26 – Accessibility Component Design: *First, we refer to Example 9 and then to Sect. 4.10.1. The current software architecture (still) prescribes strict, mutually exclusive serialisation of* client *and* staff *processes wrt.*

time_table *process. This is not acceptable for time-consuming* time_table *operations. Our design decision, in this step, is therefore to prescribe that the* time_table *process is to be a time-shared process.* "SLIDE 815" *Thus the* time_table *process is to accept up to several requests, in any order, and to service each request, for example, in some "round robin" fashion, in time slices, such that zero, one or more, but a finite, small number of time slices, allocated to one particular request, produces partial, and eventually full results for that request.*

"SLIDE 816"

To thus interleave client *requests implies the passing of* client *identity — all the way to the* time_table *process. There, at the* time_table *process, they are associated with the time-shared processing, and affixed the partial or full, computed, results when these are being communicated back in order to "sort" out which partial or completed (i.e., full) results "belong to which request".*

"SLIDE 817"

This is an acceptable solution even when it comes to the design of a time_table *process which is as general as possible! Either the operating system handles the time-sharing, and that system then handles the* client *(and now also* staff*) request identities, or the timetable subsystem must!*

"SLIDE 818"

We decide therefore to make sure that two or more client *processes can be serviced in overlapping time intervals. This will be effected by a* client queue *process (*cli_q*) inserted between the* client multiplexor *processes (i.e., components) and the* client staff arbiter *process (i.e., component), and by a* client staff queue *process (*cli_stf_q*), i.e., client staff queue component, inserted between the* client staff arbiter *process (i.e., component) and the* time_table *process (i.e., component). The latter queue secures that also otherwise serially serviced* staff *updates can be time-shared and "computed" in a piecemeal fashion.* "SLIDE 819" *We refer to Fig. 4.5 on the next page. The previous client multiplexor and arbiter processes (i.e., components) have to be redefined in light of "adding" client and (whether client* i *or) staff identities to the requests being forwarded to the time_table process.* "SLIDE 820" A journey query handling by the timetable process could thus be interleaved with the handling of a connection query from another client. The former may "arrive" at the time_table process before the latter, but that process may decide to first service the latter. A more detailed specification of what goes on during the processing of requests can be given:

We first specify the orderly flows of requests from clients and staff towards the time_table process. Thereafter we specify the orderly flows of partial and/or completed results from the time_table process back to respectively client and staff processes. These two flows must themselves be interleaved and not interfere with one another.

Each client request, after having been arbiter-chosen by the client multiplexor arbiter, is annotated with its origin (client_i) and passed on to the client queue. Meanwhile the client multiplexor arbiter is freed to accept other requests.

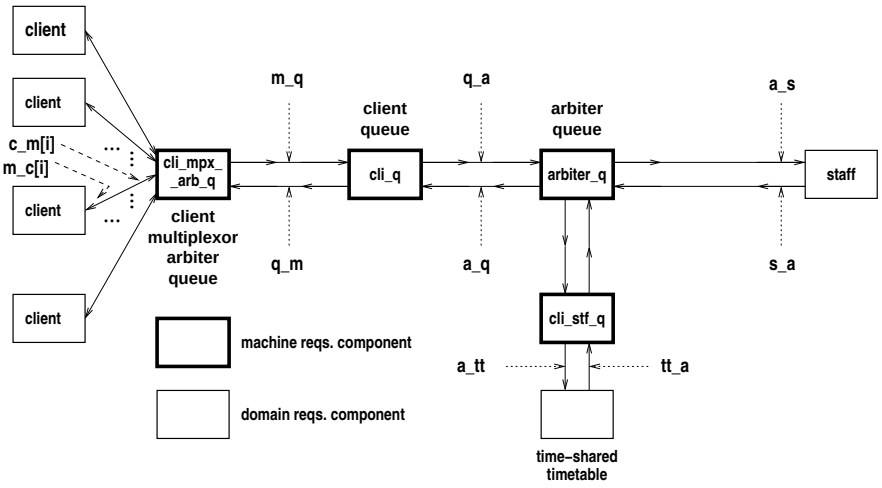


Fig. 4.5. Time-sharing interleave components

The client queue notes that a request is being made by client_i and passes that request on to the client staff arbiter, and is otherwise freed to accept further annotated client requests. The client staff arbiter selects client and staff requests and passes selected such on to the client staff queue, while being freed to arbitrate between subsequent client and staff requests. The client staff queue process notes the identity of the request and passes it on to the time_table process.

In every time-slice that the time_table process has a partial or a completed result associated with a client or a staff request, it returns that result to the client staff queue process. If the result is marked as completing a request, then the client staff queue process removes it from its list of pending requests, as now having been fully serviced by it and the time_table process, while, in any case returning the result to the client staff arbiter. That process, inspecting the identity affixed to the returned result (by the time_table process), decides where to route that result: to the staff process, or to the client staff queue process. In the former case the returned (partial or completed) result has been fully handled. In the latter case the client staff queue process inspects the returned value to see whether it represents a partial or a completed request value. If the latter, then the client staff queue process removes it from its list of pending client requests, as now having been fully serviced by it and the time_table process. In both cases the returned value is returned via the client multiplexor arbiter to the client.

The above scheme allows inspection, at any time, by a *client/staff timetable service system* (which we have not spoken of before) as to the state of pending requests. •

Design Decision wrt. Adaptability

"SLIDE 821"

Example. 27 – Adaptive Maintainability Connector Design: *First, we refer to Example 15 and then to Sect. 4.10.1. To secure that the developer does not take "white box" advantage of knowledge of how the* client, staff, time_table *and any other component processes are implemented, it is suggested to insert connectors between these and the (newly introduced)* arbiter *process. The existence of these connectors forces a "standard" ("black box") interface between connected processes.* "SLIDE 822" *The idea is that the protocol for communication between domain and machine requirements component processes, on one side, and connector processes, on the other side, is made such that previously neighbouring component processes are "effectively" shielded from one another wrt. "white box" knowledge. We refer to Fig. 4.6 on the facing page.*

The shaded circles and rounded corner boxes (both kinds without a black edge) designate these connectors. The insertion of connectors between components makes no change to the flow of messages across the network of processes. So we basically inherit the detailed, informal specification that was given for the flow of requests and results across the network of processes as given at the end of Example 26.

The meaning of the connectors is subject to a wide range of interpretations. Take an example. Let the double-arrow lines between clients and the client_multiplexor_arbiter stand for wide area communication lines. In some implementation there is trust with respect to these lines: no noise, no intrusion. The connectors on these double-arrow lines can therefore be very simple. In other implementations there is noise, but no threat of intrusion. Now these connectors need to implement some protocol that ensures uncorrupted receipts of messages. In yet other implementations there is threat of intrusion. Now the connectors must implement some encryption scheme. And so forth.
•

"SLIDE 823"

Discussion "SLIDE 824"

General

We have concluded a stage of development. From a set of requirements we have developed, informally, a software architecture, a first stage of software design. This software design is informal, and, in a sense, incomplete. It is incomplete in the sense that we have yet to specify the individual behaviours of each of the domain and machine requirements components as well as of the connector components. It is "complete" in the sense that we now have a "picture" which specifies: There are those and those processes, and there are those and those channels, and no more!

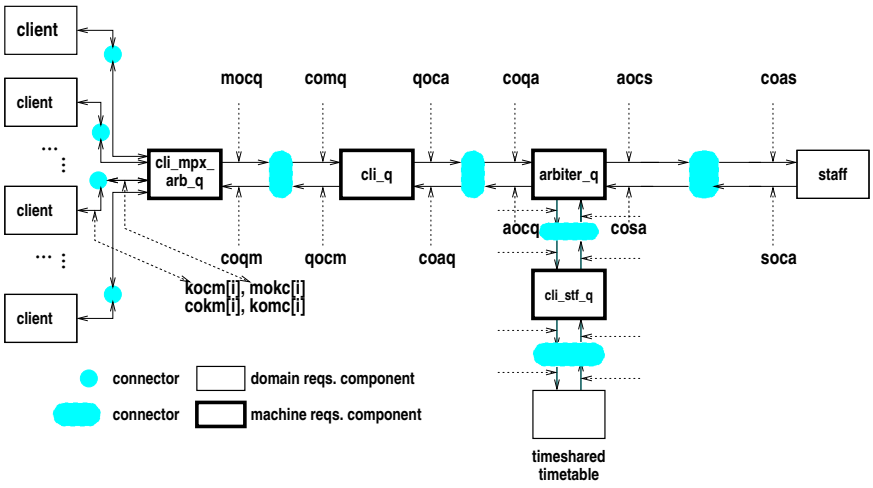


Fig. 4.6. Adaptivity connectors

"SLIDE 825"

This stage of architecture design deployed a technique — used a design approach — which was informal. It is based on "boxes and arrows". The technique is based on inserting, and perhaps renaming (i.e., redefining) "boxes and arrows". In each design decision step we presented (at the end of the corresponding examples) a detailed specification of how the network of processes was to behave. But only in the last step did we commit ourselves to a specification that eventually has to be implemented.

Principles and Techniques

"SLIDE 826"

We summarise:

Principle 10 (Software Architecture) The principles of *software architecture* are to sketch a first software design, to handle what is considered the most crucial requirements and to get the long process of software design "on the road". ∎

"SLIDE 827"

Principle 11 (Software Architecture Design) The principles of *software architecture design* are basically those of *divide and conquer*, i.e., separation of concerns, of determination of overall component structuring, and hence of determining major interfaces, so as to allow separate design groups to tackle separately the development of components. ∎

"SLIDE 828"

Principle 12 (Component) The idea of a *component* is that it serves as a suitably free-standing collection of modules, which together offer either functionalities (i.e., can perform functions), or data storage, or monitoring and/or control of processes, possibly in response to events, in order to implement a clearly identifiable (preferably small set of) requirements. ∎

"SLIDE 829"

Principle 13 (Component Design) Components emerge as a result of software architecture design. Other than that, *component design* follows all the principles of design according to which any software is designed. ∎

"SLIDE 830"

Technique 3 (Architecture Design) Major *architecture design techniques* revolve around decomposing the requirements into possibly separable major subsystems (which was not illustrated in this section); and deciding, for each such subsystem decomposition, upon definition of resulting interfaces between subsystems, their protocol of communication over these interfaces and the types of data exchanged. "SLIDE 831" Techniques within subsystems also included exploring (experimenting with) different prioritisations of related requirements in order to ascertain whether one or another prioritisation results in an architecture cum component structure (and interface) design that allows a reasonable sequence of steps of extension, each addressing subsequent, i.e., remaining (still related) requirements. ∎

Subsequent chapters will uncover additional principles and techniques.

Bibliographical Notes "SLIDE 832"

The Carnegie Mellon University group around David Garlan (G.D. Abowd, R. Allen, M. Shaw, C. Shekaran, and others) has contributed rather significantly to the clarification of many software architecture issues, notably such that relate to components and their connections: [82, 7, 2, 83, 8, 198, 3, 80, 81, 9].

"slide 833"

"slide 834"
"slide 835"
"slide 836"

4.10.2 Component Design and its Refinement

Overview Introduction "slide 837"

This section presents one large example. In it we show how a software architecture — which satisfies some initial domain requirements — is developed in steps alternating with the development of a component structure. This component structure satisfies some further machine requirements, requirements that are not really "discovered" till "halfway" through architecture design. The example also illustrates the use of *data refinement* techniques for the purpose of conquering the architectural complexity of a system.

System Complexity

That is, we are oftentimes faced with the problem of having to design a system with very many properties, too many to be grasped in any one presentation. Instead we show a technique whereby the architectures, i.e., the full variety of all properties, can be stepwise developed. From a small architectural specification — exemplifying what is considered the very basic properties — one arrives, in steps, at increasingly more complex designs. At each step a new, small set of properties is "added" to the previous description.

Sometimes software systems contain unnecessarily many, seemingly independent concepts. Occasionally a large number of such concepts are, however, necessary. Their presence is required in order to cope with varieties of domain requirements, interface requirements and, especially, machine requirements.

In all cases it is rather hard to grasp all the concepts, sort them out and interrelate them properly. In many cases this ability to dissect a software architecture into its many constituent notions is seriously hampered by opaque presentations of their interdependencies.

Proposed Remedies

"slide 838"

> You can design software in either of two ways:
> Either you make it so clear and simple such that it obviously has no bugs,
> of you make it so complex such that it has no obvious bugs.
>
> *Sir Tony Hoare*

"slide 839"

Three possibilities for "solving" the apparent complexity problem exist: two extremes, and a "middle road". These choices are either not to design such multi-concept systems at all, or go on designing them in the old "hacker" fashion. We shall sometimes choose the first extreme, sometimes the "middle road" approach outlined below, but never the second "compromise"!

Stepwise Development

"SLIDE 840"

Characterisation 107 (Stepwise Development) By *stepwise development* — other terms, used interchangeably, are *stepwise refinement*, *stepwise reification*, *stepwise transformation* — of a software design, we shall understand the following: First a model is established which exhibits, as abstractly as deemed reasonable, the intrinsic concepts and facilities for which the software was intended, that is, a model which satisfies the domain requirements. Then this model is subjected to *data* and/or *operation refinement*. "SLIDE 841"The choice of refinements is determined so as to satisfy those domain requirements which were not all taken care of in the first step, and so as to — similarly — satisfy (remaining) interface requirements, and so as to satisfy machine requirements. ■

We shall, in this section, emphasize variations of *data refinement* and *data reification* referred to by the collective term *data transformation*. A sequence of transformations may be needed. Each step introduces further properties and/or details, none, some or all of which are exploited in exposing them to an external world. The order of the steps and their nature is dictated, for example, by technological and/or product strategic considerations.

Stagewise Iteration

"SLIDE 842"

By stagewise iteration — other terms, used interchangeably, are stagewise evolution or stagewise spiralling — of a software design, we shall understand the following:

- *One or more steps of development within a stage, s, are performed.* ("Start" with stage s.)
- *Then one or more steps of development within a next stage, s', are performed.* (Forward to stage s'.)
- *Then one or more steps of development within stage s are performed.* (Back to stage s.)
- *And so on, alternating between stages $s, s', s'', \ldots, s'''$.* (Iterating, forward and backward.)

Our example exhibits stagewise iteration.

Overview of Example "SLIDE 843"

Our example is that of a file-handler system:

0. At the top level (step 0) of architecture we focus our attention on files, file names, pages and page names as *data* and the creation, and erasure, of files, and the writing, updating, reading, and deletion of pages as *operations*.
   At this step files are named and consist of named pages.

"SLIDE 844"

At the top level no concession is made to the possible storing of files and their pages in such diverse storage media as foreground (fast access, "core"), or background (slow access, "disk") storage. The decision, which is hence recorded, of eventually implementing the storing of files and pages on disk-like devices, predicates a need to be able to "look up", reasonably fast, where, on possibly several disks, files and pages are stored.

"SLIDE 845"

1., 2. In the next two steps we therefore introduce first the notions of catalogues and directories, and, subsequently, as a further step of development, abstractions of the notions of main storage and disks.

Catalogues eventually record disk addresses of file directories, one per file. Directories eventually record disk addresses of pages. Our file system at this level has one catalogue. We think, at the level of main storage and disks, of the one catalogue as always residing in main storage, whereas all directories are normally only stored on disks.

To speed up access to disk pages we operate on main storage copies of directories. The intention to operate on a file is then indicated by its opening, which is an "act" that brings a disk directory copy into main storage. The intention to not operate further on a file is then indicated by its closing, an "act" which reverses the above copying.

"SLIDE 846"

3. Hence open and close operations are introduced in step 3.

Opening and closing are file-related concepts primarily brought upon us by *efficiency* considerations. These efficiency concerns are rooted in insufficient technologies. Thus they represent machine performance requirements.

Neither at the top, nor at the second level (i.e., in steps 2 and 3) of the file-handler "architecting", did we bother about the machine requirements issue of *reliability*. We here define the reliability of our file handler as its ability to survive *crashes*.

By a "crash" we restrictively mean anything which renders main storage information (catalogue and opened directories) useless. By total "survival" we mean the ability to continue (some time) after a "crash" as if no "crash" had occurred. By "partial survival" we mean the ability to continue with at least a nonvoid subset of the files after a "crash" — with the complement set of files being clearly identified.

"SLIDE 847"

4. In the fourth step, building upon redundancies in catalogue, directory and page recordings, we therefore introduce notions of *checkpointing* files and automatic *recovery* from "crashes".

5.–6. Final steps — as presented here — hint at space management of storage and disk: We introduce free lists of unused, available disk storage, etc.

Methodology Overview "SLIDE 848"

We paraphrase the above by giving an overview of the principles and techniques to be deployed.

Principles

"SLIDE 849"

We can summarise the principles as follows:

1. *Stepwise unfolding of software architecture:* Instead of going, in "one fell swoop", from (all of) the requirements to (all of) the architecture, we decompose this stage of the development of the software architecture of a simple file handler into steps, each step taking care of one concern.
2. *Interweaving domain and machine requirements implementation:* Instead of taking care, first, of all domain requirements, we alternate between considering domain and machine requirements.
3. *Invariance:* Usually abstract type definitions, i.e., sorts, are subject to axioms which express properties of the type, i.e., of its values. These axiomatic sort properties are usually expressed in relation to (i.e., in terms of) the various functions that are applicable to (otherwise well-formed) values of the type.
   In stage- and stepwise refinement one usually represents abstract (i.e., sort) types in terms of concrete types (e.g., sets, Cartesians, lists, maps, etc.). In doing so the concrete type is usually capable of expressing "more" values than are desired, i.e., values that do not properly represent any corresponding abstract (sort) value — which was and is the idea. Hence we need to express invariance of values of a type, i.e., a subtype. We do so by defining explicit invariance predicates.
4. *Abstraction, adequacy and sufficiency:* While adhering to the above principles, we also adhere to principles of considering functions that abstract from later design steps "back" to earlier design steps, or that express the adequacy of a representation, that is, of a later design step (i.e., of a design decision), wrt. an earlier, or that express the sufficiency of a representation.
5. *Correctness:* When "performing" a step of development, from a "more" abstract to a "more" concrete design, one has to argue why the chosen step implements the abstraction. Usually a formal proof is required. Often an informal, but precise reasoning is sufficiently convincing.

Techniques

"SLIDE 850"

We summarise the techniques, referring to Sect. 4.10.2, as they are invoked in different steps:

- Intrinsic domain requirements:
  - ⋆ Step 0: Intrinsic architecture: (Sect. 4.10.2)
    files, create, erase, pages, write, update and delete

A prescription is given of what we could consider the domain requirements of the file handler. At the same time we might consider this prescription to also be a specification of the basic software architecture from which we further develop the full software architecture. Had we, instead, prescribed the domain requirements by means of sorts, of the signatures of the file handler command functions, and of axioms over these, then we could say that the specification given below is truly that of a, or the, basic software architecture.

- Machine requirements:
  - ⋆ Step 1: Catalogue (Sect. 4.10.2)
  - ⋆ Step 2: Disk (Sect. 4.10.2)

  Two steps of data refinement now follow: We choose a "more" concrete representation (of the system) than given in step 0; we define an invariance predicate; we redefine the file handler command functions; and we express adequacy, sufficiency and correctness. We do not prove correctness.
- Domain and machine requirements:
  - ⋆ Step 3: open and close commands (Sect. 4.10.2)

  We argue design decisions based on considerations of technology, and, accordingly (again) choose an "even more" concrete representation of the file-handler system than presented in step 1. We then define invariance and abstraction functions, redefine some of the file-handler command functions, and leave the expression of adequacy, sufficiency and correctness to the reader. Again we do not prove correctness.

  "SLIDE 851"
- Detailed component structure:
  - ⋆ Step 4: Crash robustness: check and crash (Sect. 4.10.2)
  - ⋆ Step 5: "Flat" storage (Sect. 4.10.2)
  - ⋆ Step 6: Space management (Sect. 4.10.2)

Step 0: Files and Pages

The next four subsections present an abstraction of a file system architecture.

A "Snapshot"

Figure 4.7 abstracts a file system of three files named f1, f2 and f3. The first file contains two pages, the second is empty and the third file contains three pages.

"SLIDE 852"

An Abstract Formal Model

"SLIDE 853"

Based on the immediately following English wording of what the type of the state of our top-level file-handler is, we "derive" informally the formal type definitions.

The sôle data structure of our file-handler consists of a set of uniquely named files. Each file consists of a set of uniquely named pages. Let *Fn, Pn* and *PAGE* denote the further unspecified types of respectively file names, page names and pages. "SLIDE 854"Then: Step 0: Architecture: Files and Pages

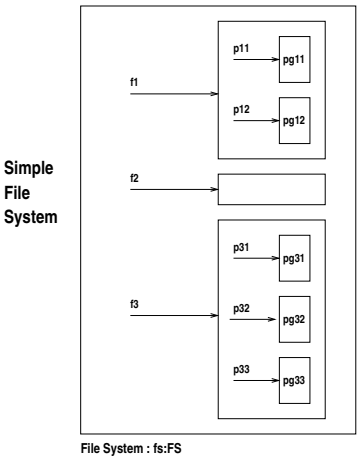


Fig. 4.7. Base file system

type
  [Step 0]
  FileSystem, File, Fn, Page, Pn, FILE, PAGE
  FS0 = Fn $\overrightarrow{m}$ FILE
  FILE = Pn $\overrightarrow{m}$ PAGE
value
  obs_Fns: FileSystem → Fn-**set**
  obs_Files: FileSystem $\xrightarrow{\sim}$ File-**set**
  obs_FS0: FileSystem $\xrightarrow{\sim}$ FS0
  obs_Fn: File $\xrightarrow{\sim}$ Fn
  obs_FILE: File $\xrightarrow{\sim}$ FILE
  obs_Pns: Page $\xrightarrow{\sim}$ Pn-**set**
  obs_Pages: File $\xrightarrow{\sim}$ Page-**set**
  obs_Pn: Page $\xrightarrow{\sim}$ Pn
  obs_PAGE: Page $\xrightarrow{\sim}$ PAGE

We have completed our first task: that of specifying the most important aspects first, namely the semantic types. We need to express an invariance: The file names of the system are those of the files of the system. And the page names of the system are those of the pages of the system.

Step 0: Architecture: Files and Pages

axiom
  ∀ fs:FileSystem •
    let fns = obs_Fns(fs), files = obs_Files(fs) **in**
    fns = { obs_Fn(f) | f:File • f ∈ files } ∧
    ∀ f:File • f ∈ files •

```
    let pns = obs_Pns(f), pages = obs_Pages(f) in
    pns = { obs_Pn(p) | p:Page • p ∈ pages }
end end
```

Abstract Versus Concrete Basic Actions

"slide 855"

To *create* an initially empty file (of no pages) we need to specify a new, hitherto unused file name. To *erase* an existing file we need to specify the name of a file already in the system. To put a page into a file we need to specify the names of the file and page, and the page itself. To *get* a page from a file we to need specify the names of the file and page. Finally, to *delete* a page we need to specify the same:

"slide 856"

Abstract Versus Concrete Basic Actions

- Create file:
  - ⋆ **Abstract:**

```
value
  crea: Fn ~→ FileSystem ~→ FileSystem
  crea(fn)(fs) as fs′
  pre: fn ∉ obs_Fns(fs)
  post: obs_FS0(fs′) = obs_FS0(fs) ∪ [ fn ↦ [ ] ]

or:

axiom
  empty: File → Bool
  empty((crea(fn)(fs))(fn)),
  ∼empty((crea(fn)(put(fn,pn,pg)(fs)))),
  undef(empty((crea(fn)(eras(fn)(fs)))))
```

"slide 857"

  - ⋆ **Concrete:**

```
value
  crea0: Fn ~→ FS0 ~→ FS0
  crea0(fn)(fs) ≡ fs ∪ [ fn ↦ [ ] ]
  pre: fn ∉ obs_Fns(fs)
```

"slide 858"

- Put file:
  - ⋆ **Abstract:**

```
value
  put: Fn × Page ~→ FileSystem ~→ FileSystem
  put(fn,pg)(fs) as fs′
  pre: fn ∈ obs_Fns(fs)
```

post: **let** cfs = obs_FS(fs), cfs′ = obs_FS(fs′),
pn = obs_Pn(pg), cpg = obs_PAGE(pg),
cfile = obs_FILE((obs_FS(fs))(fn)) **in**
cfs′ = cfs † [fn ↦ pgs † [pn ↦ cpg]] **end**

"SLIDE 859"

or:

axiom
get(fn)(put(fn,pn,pg)(fs)) = pg,
undef(get(fn)(del(fn)(fs)))

⋆ **Concrete:**
value
put0: Fn × Pn × PAGE ~→ FS ~→ FS
put0(fn,pn,pg)(fs) ≡ fs†[fn ↦ fs(fn)†[pn ↦ pg]]
pre: fn ∈ **dom** fs

Concrete Actions

"SLIDE 860"

Concrete Actions

value
eras0: Fn ~→ FS0 ~→ FS0
eras0(fn)(fs) ≡ fs \ {fn}
pre: fn ∈ **dom** fs

get0: Fn × Pn ~→ FS0 ~→ PAGE
get0(fn,pn) ≡ (fs(fn))(pn)
pre: f ∈ **dom** fs ∧ p ∈ **dom** (fs(fn))

del0: Fn × Pn ~→ FS0 ~→ FS0
del0(fn,pn)(fs) ≡ fs † [fn ↦ (fs(fn)) \ {pn}]
pre: f ∈ **dom** fs ∧ p ∈ **dom** (fs(fn))

We have completely specified the basic, major functions of a simple file handler system. The abstraction is just that: We have abstracted from any concern about how actual input of commands, including input of pages, and of how output of pages take place. We have also abstracted "away" considerations of what kind of diagnostics to use in case of erroneous input — we have only defined, in preconditions, what we mean by erroneous input. We have abstracted from any representation of files, and, in fact, the entire file system. Finally, we have not been, and shall not, in this entire example, be concerned with what pages are.

Step 1: Catalogue, Disk and Storage "SLIDE 861"

We divide the next development into three steps. First, we introduce the data notions of *catalogues* and *directories,* then the data notion of *disk,* and finally the

data notions of *storage* and *disk*. The single aim of this level is to introduce the operation notions of *open* and *close*.

Catalogue Directories

Figure 4.8 instantiates a first step of concretisation of Fig. 4.7. A catalogue and some file directories have been "inserted" as a means of obtaining access to the file pages.

"SLIDE 862"

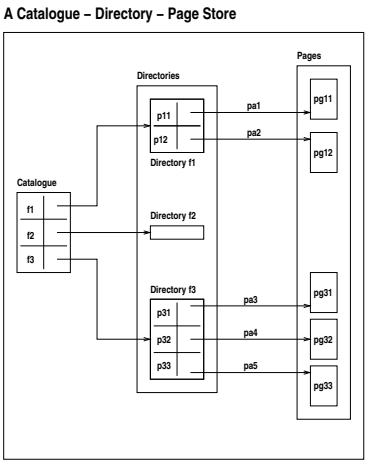


Fig. 4.8. Catalogue + directories + pages

Data Structure: "SLIDE 863"

We now adorn our type names according to the number of the step of development. The zeroth step (which was the top level) gave us FS0.

To each file in FS1 we now associate a page directory. Each directory records where pages are stored. Directories are named, and these names are recorded in a catalogue.

Catalogue + Directories + Pages

type

[Step 0]
FS0 = Fn $\xrightarrow[m]{}$ (Pn $\xrightarrow[m]{}$ PAGE)

[Step 1]
FS1 = CTLG1 × DIRS1 × PGS1

"SLIDE 864"

You may (justifiably) think of directories "translating" user-oriented page names into system-oriented page addresses, and *PGS1* to be a disk-like space within which all pages of all files are allocated. Let:

$$\left[\begin{array}{l} f_1 \mapsto \left[\begin{array}{l} p_{11} \mapsto g_{11}, \\ p_{12} \mapsto g_{12} \end{array}\right], \\ f_2 \mapsto \left[p_{21} \mapsto g_{21}\right], \\ f_3 \mapsto [\,] \end{array}\right]$$

be an abstract, *FS0*, file system. "SLIDE 865" Its counterpart in *FS1* is:

$$\left(\begin{array}{c} \left[\begin{array}{l} f_1 \mapsto d_1, \\ f_2 \mapsto d_2, \\ f_3 \mapsto d_3 \end{array}\right], \\ \left[\begin{array}{l} d_1 \mapsto \left[\begin{array}{l} p_{11} \mapsto a_{11}, \\ p_{12} \mapsto a_{12} \end{array}\right], \\ d_2 \mapsto \left[p_{21} \mapsto a_{21}\right], \\ d_3 \mapsto [\,] \end{array}\right], \\ \left[\begin{array}{l} a_{11} \mapsto g_{11}, \\ a_{12} \mapsto g_{12}, \\ a_{21} \mapsto g_{21} \end{array}\right], \end{array}\right)$$

Invariant: "SLIDE 866"

The type definitions define too much. Not all combinations of catalogues, directories and pages go together. We must require that there is a distinct directory in *DIRS1* for each file catalogued in *CTLG1;* that pages addressed in *PGS1* are actually recorded in directories; and that every page, understood as page-address, is described in exactly one directory (that is belongs to exactly one file).

"SLIDE 867"

Catalogue + Directory + Page Invariants

```
type
    [ Step 0 ]
  FS0 = Fn m→ (Pn m→ PAGE)

    [ Step 1 ]
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1
  CTLG1 = Fn m→ Dn
  DIRS1 = Dn m→ DIR1
  DIR1 = Pn m→ Pa
  PGS1 = Pa m→ PAGE

value
  inv_CTLG1: CTLG1 → Bool
  inv_CTLG_1(ctlg) ≡ card dom ctlg = card rng ctlg
```

```
inv_DIRS1: → Bool
inv_DIRS1(dirs) ≡
  card dom dirs = card rng dirs ∧
  ∀ dir:DIR1 • dir ∈ rng dirs ⇒ inv_DIR1(dir)

inv_DIR1: → Bool
inv_DIR1(dir) ≡ card dom dir = card rng dir

inv_PGS1: → Bool
inv_PGS1(pgs) ≡ card dom pgs = card rng pgs

inv_FS1: FS1 → Bool
inv_FS1(ctlg,dirs,pgs) ≡
  inv_CTLG_1(ctlg) ∧inv_DIRS1(dirs) ∧ inv_PGS1(pgs) ∧
  rng ctlg = dom dirs ∧
  ⋃ { rng dir | dir:DIR1 • dir ∈ rng dirs } = dom pgs ∧
  ∀ pa:Pa•pa ∈ dom pgs•∃! dn:Dn•dn ∈ dom dirs•pa ∈ rng dirs(dn)
```

"SLIDE 868"

Annotations:

- The more concrete catalogue is well-formed if it is a bijection: To each file name there corresponds a unique directory name.
- The collection of more concrete directories is well-formed if it is a bijection: To each directory name there corresponds not only a unique directory, but each of these directories is well-formed, i.e., is also a bijection. Directories map each page name to a unique page.
- The collection of more concrete pages is well-formed if it is a bijection: To each page address there corresponds a unique page. "SLIDE 869"

The previous three items were concerned only with the well-formedness of respective components of the overall more concrete file system. What is missing, namely the constraints that "cut across" the triplet structure is now formulated:

- The more concrete file system is well-formed:
  - ⋆ if each of its parts is well-formed,
  - ⋆ if the directory names mentioned in the catalogue correspond exactly to those mentioned in the collection of directories, and
  - ⋆ if, for every page address in the collection of pages, there exists a unique directory name in whose directory that page address is mentioned.

Abstraction

"SLIDE 870"

Given an *FS1* file system we can abstract a "corresponding" *FS0* from it. Abstraction is a function.

"SLIDE 871"

Catalogue + Directory + Page Abstraction

```
type
    [ Step 0 ]
  FS0 = Fn ~m~> (Pn ~m~> PAGE)

    [ Step 1 ]
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1
  CTLG1 = Fn ~m~> Dn
  DIRS1 = Dn ~m~> DIR1
  DIR1 = Pn ~m~> Pa
  PGS1 = Pa ~m~> PAGE

value
  abs_FS0: FS1 ~~> FS0
  abs_FS0(ctlg,dirs,pgs) ≡
    [ fn ↦ [ pn ↦ pgs((dirs(ctlg(fn)))(pn))
           | pn:Pn • pn ∈ dom dirs(ctlg(fn)) ]
    | fn:Fn • fn ∈ dom ctlg ]
  pre: inv_FS1(ctlg,dirs,pgs)
```

"SLIDE 872"

We can only retrieve (i.e., abstract) well-formed file systems.

- To abstract, i.e., to retrieve, from a more concrete file system, its abstract counterpart is
  - ⋆ for every file name, in the concrete catalogue,
  - ⋆ to reconstruct named pages:
    - · namely, for every page address in the directory for that file
    - · to map it into its page in the collection of pages.

As an aside: We could also, in addition to abstraction functions, define their "inverse", injection functions:

```
type
  A, B
value
  wf_A: A → Bool, wf_B: B → Bool
  abs_A: B ~~> A, inj_B: A ~~> B-infset
axiom
  ∀ a:A • wf_A(a) ⇒ ∀ b:B • wf_B(b) ⇒ b ∈ inj_B(a) ⇒ abs_A(b)=A
```

Actions

"SLIDE 873"

We rewrite the action functions in terms of the new semantic types:

Action Signatures: Action Signatures

```
value
  crea1: Fn → FS1 ~→ FS1
  eras1: Fn → FS1 ~→ FS1
  put1: (Fn × Pn × PAGE) → FS1 ~→ FS1
  get1: (Fn × Pn) → FS1 ~→ PAGE
  del1: (Fn × Pn) → FS1 ~→ FS1
```

"SLIDE 874"

There are five commands: crea1, eras1, put1, get1 and del1 (create, erase, put, get and delete). To create a file all the syntax that is needed is a new file name. To do the update requires the entire file system — and results in a new file system. We leave the "reading" of the remaining signatures for the reader to decipher.

Create and Erase File Actions: "SLIDE 875"

Create and Erase File Actions

```
value
  crea1(fn)(ctlg,dirs,pgs) ≡
    let d:D • d ∉ dom dirs in (ctlg ∪ [fn↦d],dirs ∪ [d↦[]],pgs) end
  pre: f ∉ dom ctlg

  eras1(fn)(ctlg,dirs,pgs) ≡
    (ctlg \ {fn},dirs \ {ctlg(fn)},pgs \ rng dirs(ctlg(fn)))
  pre: f ∈ dom ctlg
```

"SLIDE 876"

To create a named file is to "fetch" a new directory name, to let that directory name be the designation of the file name in the catalogue, to initialise the named directory to an empty such, and to not change the collection of pages.

Put Page Action: "SLIDE 877"

Put Page Action

```
value
  put1(fn,pn,pg)(ctlg,dirs,pgs) ≡
    if pn ∈ dom dirs(ctlg(fn))
      then
        (ctlg,dirs,pgs † [(dirs(ctlg(fn)))(pn) ↦ pg])
      else
        let pa:Pa • pa ∉ dom pgs in
        let dirs′ = dirs ∪ [ctlg(fn)↦(dirs(ctlg(fn)))]∪[pn↦pa],
            pgs′ = pgs ∪ [pa ↦ pg] in
        (ctlg,dirs′,pgs′) end end end
  pre: f ∈ dom ctlg
```

"SLIDE 878"

To put a page into the file system is to overwrite that named page in the collection of pages if there was already one by that name (and hence address). Otherwise it is to "fetch" a new page address, to extend the appropriate directory with the page name to page address association, and then to extend the collection of pages accordingly. The former is an update, and the latter is a write.

Get and Delete Page Actions: "SLIDE 879"

Get and Delete Page Actions

value

get1(fn,pn)(ctlg,dirs,pgs) ≡ pgs(dirs(ctlg(fn))(pn))
pre: f ∈ **dom** ctlg ∧ pn ∈ **dom**(dirs(ctlg(fn)))

del1(fn,pn)(ctlg,dirs,pgs) ≡
(ctlg,
dirs † [ctlg(fn) ↦ (dirs(ctlg(fn))) \ {pn}],
pgs \ {(dirs(ctlg(fn)))(pn)})
pre: f ∈ **dom** ctlg ∧ pn ∈ **dom**(dirs(ctlg(fn)))

"SLIDE 880"

To get a named page from a named file is to look it up in the collection of pages under the address in the directory as so directed by the catalogue. To delete a named page from a named file is to remove the page name to page address association from the directory and the page from the collection of pages.

Adequacy and Sufficiency

"SLIDE 881"

Correctness of the above realisations of the semantic actions with respect to the realisation of *FS0* in terms of *FS1* is expressed by means of the (i.e., an) abstraction function.

We "divide" our correctness concern into three parts: adequacy of chosen concrete data representation, sufficiency of the same, and correctness of each concrete action specification with respect to the corresponding abstract action specification.

Adequacy: "SLIDE 882"

Adequacy

axiom

∀ fs0:FS0, ∃ fs1:FS1 • inv_FS1(fs1) ⇒ fs0 = abs_FS0(fs1)

A concrete file system model is adequate with respect to an abstract file system model if for every abstract file system there corresponds a more concrete well-formed one which abstracts, i.e., which retrieves to the abstract file system.

Sufficiency: "slide 883"
Sufficiency

axiom
∀ fs1:FS1 • inv_FS1(fs1) ⇒ abs_FS0(fs1) ∈ FS0

A concrete file system model is sufficient with respect to an abstract file system model if every well-formed concrete file system abstracts to an abstract file system.

Correctness

"slide 884"

Comparable Results: To express correctness of concrete action specifications with respect to concrete action specifications we need to define an abstraction function on results (RES).

"slide 885"
Comparable Results

type
[Step 0]
RES0 = FS0 | PAGE
[Step 1]
RES1 = FS1 | PAGE
value
abs_RES0: RES1 $\xrightarrow{\sim}$ RES0
abs_RES0(r) ≡
if r ∈ FS1
then if inv_FS1(r) **then** abs_FS0(r) **else** undef **end**
else r **end**

"slide 886"
The abstraction of a result which is an entire concrete file system requires that that concrete file system is invariant, i.e., well-formed, and is then its abstraction. Concrete pages do not differ, in this development, from abstract pages.

The Correctness Statement: "slide 887"
Correctness

axiom
[adequacy] ∧ [sufficiency] ∧
abs_RES0(crea1(fn)fs1) = crea0(fn)fs0 ∧
abs_RES0(eras1(fn)fs1) = eras0(fn)fs0 ∧
abs_RES0(put1(fn,pn,pg)fs1) = put0(fn,pn,pg)fs0 ∧
abs_RES0(get1(fn,pn)fs1) = get0(fn,pn)fs0 ∧
abs_RES0(del1(fn,pn)fs1) = del0(fn,pn)fs0

Where '=' "extends" to: undef = undef!

"SLIDE 888"

Correctness of this step of development is now that the semantic types, i.e., concrete data representation, at this step, are adequate, that its concrete data representation is sufficient, and that every concrete operation yields a result which is comparable to that of the corresponding abstract operation. This can be diagrammed as the commutation of two algebras. See Fig. 4.9.

"SLIDE 889"

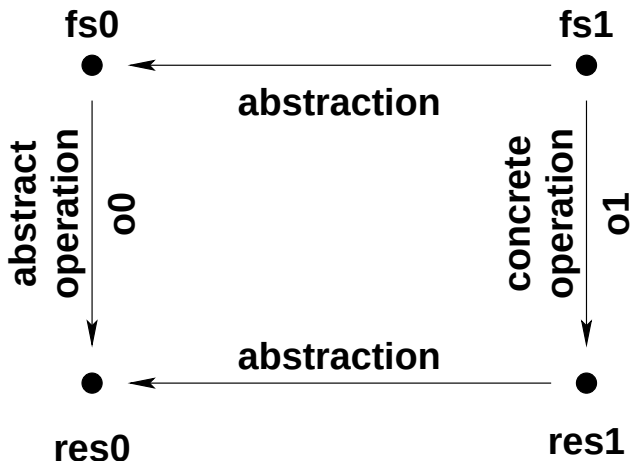


Fig. 4.9. Correctness: commutation of two algebras (FS0, FS1)

"SLIDE 890"

Let us express the correctness theorem a bit more precisely. For every pair of abstract (fs0:FS0) and concrete (fs1:FS1) file systems such that the concrete one abstracts to the abstract one, it shall be the case that for respective, and corresponding operations (o0, o1) that the results (o0(fs0), o1(fs1)) are comparable.

Step 2: Disks "SLIDE 891"

Data Refinement

The data refinement of this step involves the "gathering" (into one component of *FS2) of directories and pages, that is, of the above DIRS1 and PGS1 components of FS1, called DSK2. DIRS1* and *PGS1* are modelled as maps, and *DSK2* will hence be a "merged" type of similar maps. "SLIDE 892"Where before catalogue and directory map range types were directory names, respectively page addresses:

$$\left(\begin{bmatrix} f_1 \mapsto d_1, \\ f_2 \mapsto d_2, \\ f_3 \mapsto d_3 \end{bmatrix}, \begin{bmatrix} d_1 \mapsto \begin{bmatrix} p_{11} \mapsto a_{11}, \\ p_{12} \mapsto a_{12} \end{bmatrix}, \\ d_2 \mapsto \begin{bmatrix} p_{21} \mapsto a_{21} \end{bmatrix}, \\ d_3 \mapsto [\,] \end{bmatrix}, \begin{bmatrix} a_{11} \mapsto g_{11}, \\ a_{12} \mapsto g_{12}, \\ a_{21} \mapsto g_{21} \end{bmatrix}\right)$$

The "merged" (or "gathered") type will only have addresses in its map definition set. We think of *DSK2* as modelling "actual" disks.

Disk Type

"SLIDE 893"

A "Snapshot": Figure 4.10 is intended to show a "monolithic" state which consists of three components: catalogue, disk directories and disk pages. The catalogue is intended to be (foreground) storage-bound, whereas the directories and pages are to be disk-bound — as shown by the rectangle drawn around the latter. The formalisation captures this grouping.

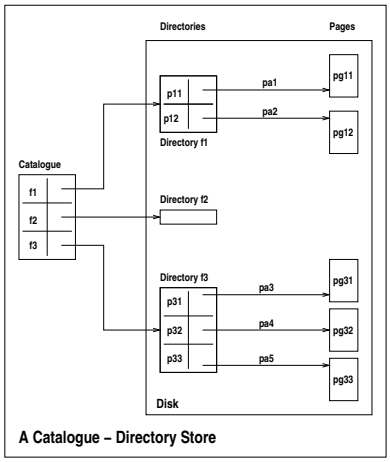


Fig. 4.10. A "snapshot"

FS0, FS1 and FS2 Types

"SLIDE 894"

Concrete Semantic Types: Concrete Semantic Types

type
  [Step 0]
  Fn, Pn, PAGE
  FS0 = Fn $\overrightarrow{m}$ (Pn $\overrightarrow{m}$ PAGE)
  [Step 1]
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1
  CTLG1 = Fn $\overrightarrow{m}$ Dn
  DIRS1 = Dn $\overrightarrow{m}$ DIR1
  DIR1 = Pn $\overrightarrow{m}$ Pa
  PGS1 = Pa $\overrightarrow{m}$ PAGE
  [Step 2]
  Adr = Dn | Pa
  FS2 = CTLG2 × DISK2
  CTLG2 = Fn $\overrightarrow{m}$ Adr
  DISK2 = Adr $\overrightarrow{m}$ (DIR2 | PAGE)
  DIR2 = Pn $\overrightarrow{m}$ Adr

that is:

type
  DISK2 = (Dn $\overrightarrow{m}$ DIR2) $\bigcup$ (Pa $\overrightarrow{m}$ PAGE)

Here addresses *Adr* (like file names, *Fn,* and page names, *Pn,* and pages, *PAGE*) are further undefined. The $\bigcup$ operator on map types is not proper RSL, but could have been so without much trouble.

Disk Type Invariant

"SLIDE 895"

Again, the type definitions define too much. In addition to the invariants ["carried over" from the very similar definitions of *FS1*], we must (first) make sure that directory addresses (listed in the catalogue) really denote directories on the disk, respectively that page addresses listed in directories really denote pages on the disk. Once this is established we can retrieve *FS1* data from such "tentatively well-formed" *FS2* data, and this abstracted data must satisfy the earlier stated constraints.

"SLIDE 896"

Disk Type Invariant

value
  inv_FS2: FS2 → **Bool**
  wf_Dirs: FS2 → **Bool**

  inv_FS2(fs2) ≡ wf_Dirs(fs2) ∧ inv_FS1(abs_FS1(fs2))

```
wf_Dirs(ctlg,disk) ≡
  ∀ a:Adr • a ∈ rng ctlg
    ⇒ a ∈ Dn ∧ disk(a) ∈ DIR2 ∧ ∀ a′:Adr • a′ ∈ rng disk(a)
      ⇒ a′ ∈ Pa ∧ disk(a′) ∈ PAGE
```

Disk Type Abstraction

"slide 897"

We leave as an exercise to narrate the abstraction function from FS2 to FS1. But here, at least, is the formalisation:

Disk Type Abstraction

```
value
  abs_FS1: FS2 ~→ FS1
  abs_FS1(ctlg,disk) ≡
    (ctlg,[ a ↦ disk(a) | a:Adr • a ∈ rng ctlg ],disk \ rng ctlg)
```

Adequacy, Sufficiency, Operations and Correctness

"slide 898"

We leave as an exercise to define adequacy and sufficiency; semantic actions: crea2, eras2, put2, get2, and del2; and correctness.

Step 3: Caches "slide 899"

Technology Considerations

We enumerate some technology constraints as they help motivate our next design decisions.

- Storage space is expensive. Disk space is less so.
- Storage access is fast. Disk access is less so.
- Hence some data are in storage; most are on disk.
- Hence accessible data must first be "opened".

We shall then see (i.e., next) our "design decision response" to the above technology constraints.

Cached Directory and Page Access

"slide 900"

We now face the reality of storages and disks. By a storage we shall understand a memory medium for which access to information is orders of magnitude faster than to information on what we shall then call disks! Access to pages (on disk) goes via catalogue and directories, where the latter are

also on disk. Thus two disk accesses per page access. (In this discussion we think of the catalogue as residing in storage.) To cut down on disk accesses we therefore decide to copy into storage the directories of those files whose pages we wish to access. In the resulting model all pages will still be thought of as stored only on the disk.

"SLIDE 901"

Figure 4.11 shows how some directories are opened, that is, are cached (hence copied) in fast access storage.

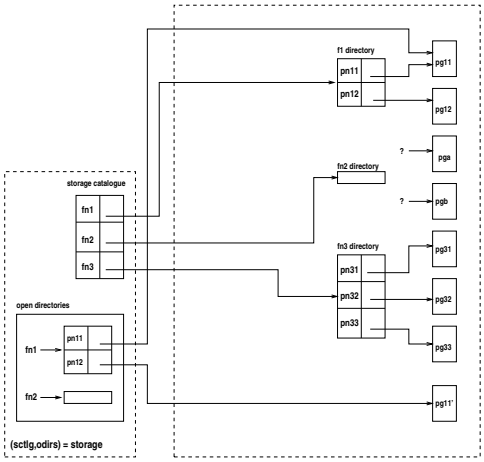


Fig. 4.11. Cached directory and page access

"SLIDE 902"

"SLIDE 903" Semantic Data Types

type

[Step 0]

Fn, Pn, PAGE

FS0 = Fn $\overrightarrow{m}$ (Pn $\overrightarrow{m}$ PAGE)

[Step 1]

Dn, Pa

FS1 = CTLG1 × DIRS1 × PGS1

CTLG1 = Fn $\overrightarrow{m}$ Dn

DIRS1 = Dn $\overrightarrow{m}$ DIR1

DIR1 = Pn $\overrightarrow{m}$ Pa

PGS1 = Pa $\overrightarrow{m}$ PAGE

[Step 2]

Adr = Dn | Pa

```
  FS2 = CTLG2 × DISK2
  CTLG2 = Fn ~m~> Adr
  DISK2 = Adr ~m~> (DIR2 | PAGE)
  DIR2 = Pn ~m~> Adr

    [Step 3]
  FS3 = STG3 × DISK3
  STG3 = CTLG2 × (Fn ~m~> DIR2)
  DISK3 = DISK2
value
  open3: Fn → FS3 ~→ FS3
  clos3: Fn → FS3 ~→ FS3
```

"SLIDE 904"

So the data system consists now of a part residing in storage and another part residing on disk. The storage part has two parts: the only catalogue (there is), and the directories of opened files. The disk part consists of all directories and all pages, merged into one map, as in the previous step.

Invariance

"SLIDE 905"

Invariance

```
value
  inv_FS3: FS3 → Bool
  wf_StgDiskOverlap: FS3 → Bool

  inv_FS3(fs3) ≡ wf_StgDiskOverlap(fs3) ∧ inv_FS2(abs_FS2(fs3))

  wf_StgDiskOverlap((ctlg,odirs),disk) ≡
    dom odirs ⊆ dom ctlg ∧ ∀ fn:Fn • fn ∈ dom ctlg
      ⇒ odirs(fn)/dom disk(ctlg(fn)) = disk(ctlg(fn))/dom odirs(fn)
```

"SLIDE 906"

The well-formedness of the new file system has two parts: First, there must be "identity" (referred to as "overlap") between those (opened) directories residing in storage and those of the same files residing on the disk. Second, the file system abstracted from the now more concrete new file system must be invariant. We see that the opened (storage-bound, or residing) directories take precedence over the similar file directories residing on disk. That is, updates on the opened directories are not propagated "back" onto the disk before closing the respective directories.

This is reflected in the abstraction function, which retrieves "more abstract" file systems (step 2) from more concrete file systems (step 3); see next.

Abstraction

"SLIDE 907"

Abstraction

```
value
  abs_FS2: FS3 ~→ FS2
  abs_FS2(stg,dsk) ≡
    (ctlg,disk † [ ctlg(fn) ↦ odirs(fn) | fn:Fn • fn ∈ dom odirs ])
```

We leave the narration of the abstraction function to the reader.

Actions

"SLIDE 908"

We leave the annotation of the more concrete (step 3) action specifications to the reader.

Actions

Open and Close Actions: "SLIDE 909"

```
type
  FS3 = STG3 × DISK3
  STG3 = CTLG2 × (Fn -m→ DIR2)
  DISK3 = Adr -m→ (DIR2 | PAGE)
  CTLG2 = Fn -m→ Adr
  DIR2 = Pn -m→ Adr

value
  open3: Fn → FS3 ~→ FS3
  clos3: Fn → FS3 ~→ FS3

  open3(fn)((ctlg,odirs),disk) ≡
    ((ctlg,odirs ∪ [ fn ↦ disk(ctlg(fn)) ]),disk)
  pre: fn ∈ dom ctlg ∧ fn ∉ dom odirs

  clos3(fn)((ctlg,odirs),disk) ≡
    ((ctlg,odirs \ {fn}),disk † [ ctlg(fn) ↦ odirs(fn) ])
  pre: fn ∈ dom ctlg ∧ fn ∈ dom odirs
```

Create and Put Actions: "SLIDE 910"

```
value
  crea3: Fn → FS3 ~→ FS3
  crea3(fn)((ctlg,odirs),disk) ≡
    let dn:Adr/Dn • a ∉ dom disk in
    ((ctlg ∪ [ fn ↦ dn ],odirs),disk ∪ [ dn ↦ [ ] ]) end
```

pre: fn ∉ **dom** ctlg

put3: Fn × Pn × PAGE → FS3 ~→ FS3
put3(fn,pn,pg)((ctlg,odirs),disk) ≡
  if pn ∈ **dom** odirs(fn)
    then
      ((ctlg,odirs),disk † [(odirs(fn))(pn) ↦ pg])
    else
      let pa:Adr/Pa • pa ∉ **dom** disk **in**
      let odirs′ = odirs † [fn ↦ odirs(fn) ∪ [pn ↦ pa]],
        disk′ = disk ∪ [pa ↦ pg] **in**
      ((ctlg,odirs′),disk′)
  end end end
  pre: f ∈ **dom** ctlg ∧ fn ∈ **dom** odirs

Erase, Get, and Delete Actions: "slide 911"

These are left as exercises!

Adequacy, Sufficiency and Correctness

"slide 912"

These are also left as exercises! Hint: Recall nondeterministic selection of Dn's and Pa's in FS2 and FS3. Therefore postulate the existence of one-to-one mapping(s) between (pairs of) Dn's, between (pairs of) Pa's and between Adr's and Dn's or Pn's.

Step 4: Storage Crashes "slide 913"

By storage crash we mean that information, i.e., data, kept by storage, as apart from being kept by a disk, is corrupted and can no longer be relied upon.

Storage and Disk

"slide 914"

The catalogue is maintained in storage and if a crash occurs it cannot be used in order to gain access to the disk. Thus, to safeguard against loss of data a copy of the catalogue is kept on disk. Every so often the storage ("master") catalogue is copied — "checkpointed" — onto the disk. When a crash occurs, the disk is considered intact, and the disk copy of the catalogue can be copied "back" to storage. Certain actions performed between the most recent checkpoint and cashe restore must be repeated.

"slide 915"

Figure 4.12 shows, relative to Fig. 4.11, the insertion of a copy on disk of the storage (hence the disk) catalogue.

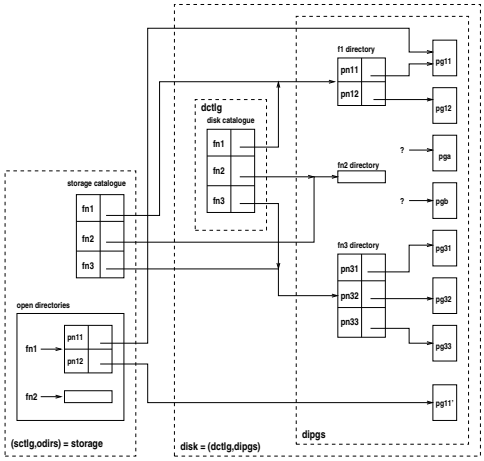


Fig. 4.12. Replicate catalogue

Concrete Semantic Types

"SLIDE 916"

Concrete Semantic Types

type

[Step 3]

FS3 = STG3 × DISK2
STG3 = CTLG2 × (Fn $\overrightarrow{m}$ DIR2)
DISK2 = Adr $\overrightarrow{m}$ (DIR2 | PAGE)
CTLG2 = Fn $\overrightarrow{m}$ Adr
DIR2 = Pn $\overrightarrow{m}$ Adr

[Step 4]

FS4 = STG3 × DISK4
DISK4 = CTLG2 × DISK2

value

inv_FS4: FS4 → **Bool**
inv_FS4(stg.disk) ≡ consSTG(stg,disk) ∧ consDISK(disk)

consSTG: FS4 → **Bool**
consDISK: DISK4 → **Bool**

Invariance

"SLIDE 917"

Well-formedness of the fourth step file system design is the conjunction of a consistent storage and consistent storage-disk pages.

Consistent Storage and Disks

"slide 918"

Consistent Storage: A consistent storage has the names of all the opened (hence storage-bound) directories be a subset of the disk directories. Further, when retrieving the disk system — from the storage directories and the additionally defined disk directories — that current disk system, restricted (/) to those pages that can be reached from storage directories (as, and hence excluding void, old pages), but extended (†) with the disk system (ds) reachable from the storage directories, shall be disk system 2 (i.e., DIR2) invariant.

"slide 919"

Consistent Storage

```
value
  consSTG: FS4 → Bool
  consSTG((ctlg,odirs),(,dipgs)) ≡
    dom odis ⊆ dom ctlg
    ∧ inv_FS2(ctlg,currentSDiPgs((ctlg,odirs),dipgs))

  currSDiPgs: (STG3×(Fn ~m~> DIR2))×DISK2 → DISK2
  currSDiPgs((stg,odirs),dipgs) ≡
    let as = currSAddrs((ctlg,odirs),dipgs) in
    let ds = [ ctlg(fn)↦odirs(fn)|fn:Fn•fn ∈ dom odirs ]
    in (disk / as) † ds end end

  currSAddrs: (STG3×(Fn ~m~> DIR2))×DISK2 ~→ Adr-set
  currSAddrs((stg,odirs),dipgs) ≡
    let das = rng ctlg,
        opas = ⋃ { rng dir | dir:DIR2 • dir ∈ rng odirs},
        cpas = ⋃ { rng dipgs(a) | a:Adr • a ∈ { ctlg(fn)
                          | fn:Fn • fn ∈ dom ctlg \ dom odirs}} in
    das ∪ opas ∪ cpas end
```

Consistent Disk: "slide 920"

The current disk system includes only those pages which can be "reached" (i.e., addressed) from the disk catalogues.

"slide 921"

Consistent Disk

```
type
  DISK4 = CTLG2 × DISK2
        = (Fn ~m~> Adr/Dn) × ((Adr/Pa) ~m~> ((Dn ~m~> Adr) | PAGE))
value
```

```
consDISK: DISK4 → Bool
consDISK(dctlg,dipgas) ≡ inv_FS2(dctlg,currDDiPgs(dctlg,dipgas))

currDDiPgs: DISK4 → Bool
currDDiPgs(dctlg,dipgs) ≡ dipgs / currDAddrs(dctlg,dipgs)

currDAddrs: DISK4 ~→ Adr-set
currDAddrs(dctlg,dipgs) ≡
  let das = rng dctlg in
  let pas = ⋃ { rng dipgs(a) | a:Adr • a ∈ das } in
  das ∪ pas end end
```

Abstractions

"slide 922"

One can abstract, i.e., retrieve to step 3 file systems either on the basis of storage catalogues, or on the basis of disk catalogues. The corresponding retrieve functions use the same restriction functions as defined for consistencies of storage, respectively disk subsystems above.

"slide 923"

Retrieval Functions

From Storage:

```
value
  abs_FS3_STG: FS4 ~→ FS3
  abs_FS3_STG((sctlg,odirs),(,dipgs)) ≡
    ((sctlg,odirs),dsk/CurrSAddrs(sctlg,dipgs))
```

From Disk:

```
value
  abs_FS3_DSK: FS4 ~→ FS3
  abs_FS3_STG(,(dctlg,dipgs)) ≡
    ((sctlg,[ ]),dipgs/CurrDAddrs(sctlg,dipgs))
```

Garbage Collection

"slide 924"

In garbage collection we delete all those pages which can no longer be "reached" from the current storage and disk directories.

Garbage Collection

```
value
  GarbColl: FS4 ~→ FS4
  GarbColl((sctlg,odirs),(dctlg,dipgs)) ≡
```

```
    let sas = currSAddrs((sctlg,odirs),dipgs),
        das = currDAdrs(dctlg,dipgs) in
    ((sctlg,odirs),(dctlg,dipgas/sas ∪ das)) end
```

New Actions

"slide 925"

Check and Crash Actions: To checkpoint a file means to update the disk with the latest version of that file's storage directory. For that a new, i.e., a "fresh" address is "fetched" and storage and disk catalogues suitably updated.

"slide 926"

Check and Crash Actions

```
value
  check: Fn → FS4 ~→ FS4
  check(fn)((sctlg,odirs),(dctlg,dipgs)) ≡
    let a:Addr • a ∉ dom dipgs in
    ((sctlg † [ fn ↦ a ],odirs),
          (dctlg † [ fn ↦ a ],
                dipgs ∪ [ a ↦ odirs(fn) ])) end
  pre: fn ∈ dom sctlg ∧ fn ∈ dom odirs

  crash: () → FS4 ~→ FS4
  crash()(,(dctlg,dipgs)) ≡ ((dctlg,[ ]),(dctlg,dipgs))
```

To crash here means to render the storage catalogues void.

Some Previous Commands

"slide 927"

Open and Close Actions: We leave it to the interested reader to "trace" the changes to the specifications of the open and close commands with respect to the file system of step 3.

"slide 928"

Open and Close Actions

```
value
  open4: Fn ~→ FS4 ~→ FS4
  open4((sctlg,opdirs),(dctlg,dipgs)) ≡
    ((sctlg,odirs ∪ [ fn ↦ dipgs(sctlg(fn)) ]),(dctlg,dipgs))
  pre: fn ∈ dom sctlg ∧ fn ∉ dom odirs

  close4: Fn ~→ FS4 ~→ FS4
  close4((sctlg,opdirs),(dctlg,dipgs)) ≡
    let a:Adr • a ∉ dom dipgs in
    ((sctlg † [ fn ↦ a ],odirs \ {fn}),(dctlg,dipgs ∪ [ a ↦ odirs(fn) ])) end
  pre: fn ∈ dom odirs
```

Put Action: "SLIDE 929"
Put Action

value
put4: Fn × Pn × PAGE $\xrightarrow{\sim}$ FS4 $\xrightarrow{\sim}$ FS4
put4(fn,pn,pg)((sctlg,opdirs),(dctlg,dipgs)) ≡
let a:Adr • a ∉ **dom** dipgs **in**
((sctlg,odirs † [fn ↦ odirs(fn) † [pn ↦ a]]),
(dctlg,dipgs ∪ [a ↦ pg])) **end**

The put action resembles the check file action.

Step 5: Flattening Storage and Disks "SLIDE 930"

"Flat" Storage and Disk

The former step of development modelled the disk as a pair consisting of a catalogue and the "previous" disk model. We now "merge" the former into the latter.

"SLIDE 931"

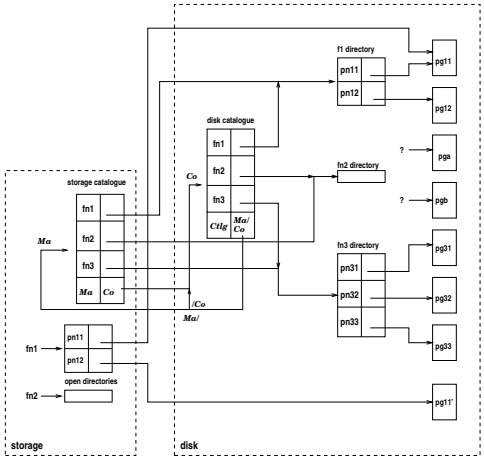


Fig. 4.13. "Flat" storage and disk

"SLIDE 932"
Figure 4.13 is really very much like Fig. 4.12. In that former figure some dashed (disk) boxes (disk catalogue and disk pages) and a fully drawn storage box (open directories) indicated separate accessible disk and storage areas. These are now "merged" into being generally addressable.

"Flat" Storage and Disk From the former models we have:

type

[Step 3]
FS3 = STG3 × DISK2
STG3 = CTLG2 × (Fn $\overrightarrow{m}$ DIR2)
DISK2 = Adr $\overrightarrow{m}$ (DIR2 | PAGE)
CTLG2 = Fn $\overrightarrow{m}$ Adr
DIR2 = Pn $\overrightarrow{m}$ Adr

[Step 4]
FS4 = STG3 × DISK4
DISK4 = CTLG2 × DISK2

"slide 933"

Defined in terms of some of the former types we get:

[Step 5]
Loc, Adr
FS5 :: STG5 × DISK5
STG5 = ({master} $\overrightarrow{m}$ SCTLG5) $\bigcup$ (Loc $\overrightarrow{m}$ DIR5)
DISK5 = ({copy} $\overrightarrow{m}$ DCTLG5) $\bigcup$ (Adr $\overrightarrow{m}$ (DIR5|PAGE))
SCTLG5 = ({master} $\overrightarrow{m}$ {copy}) $\bigcup$ (Fn $\overrightarrow{m}$ DAdr)
DCTLG5 = ({ctlg} $\overrightarrow{m}$ {master,copy}) $\bigcup$ (Fn $\overrightarrow{m}$ Adr)
DAdr = Adr × Ref
Ref == nil | Loc

"The Rest"

"slide 934"

We leave the definition of invariants, abstraction function, actions, adequacy, sufficiency, and correctness — as an exercise — to the reader.

Step 6: Disk Space Management "slide 935"

The Issue

We refer to Fig. 4.14. We can usually consider both storage and disk to both consist of (thus two) finite sets of segments. At any one time some of these pages are being used — for storage and disk catalogues, directories and file pages. And, at any one time the remaining, thus unused, segments are "free". We decide, therefore, to maintain "lists" (actually sets) of references to free segments.

"slide 936"

"slide 937"

Free segments can be allocated for new file (disk) pages, or new file (storage or disk) directories. Such allocation removes a reference to a free segment from

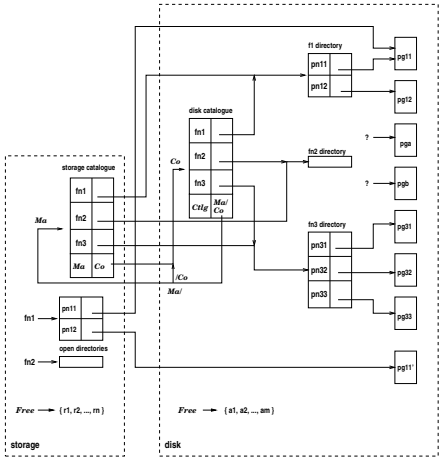


Fig. 4.14. Space management

the appropriate "free list". Whenever directories and file pages have been deleted *and* storage and disk catalogues and directories made to correspond, then garbage collection can be applied. Garbage collection now "returns" the reference to the garbage-collected segment to the "free list".

"The Rest"

"SLIDE 938"

We leave the definition of types, invariants, abstraction function, actions, adequacy, sufficiency and correctness — as an exercise — to the reader.

Discussion "SLIDE 939"

General

A long odyssey has ended. We have "slowly", in small, "measured" steps unfolded a software component design. From the basis of a "small" system component, we added, "one-by-one", additional properties. Some properties were determined by domain requirements considerations. Other properties were determined by machine requirements considerations. "SLIDE 940"Some "new, added" properties could be invoked by new commands. Other "new, added" properties are just "always" there. To properly read and grasp this chapter the reader must carefully read and make sure to understand every detailed step, its design decisions, i.e., each and every line of specification, and its micro steps. Proper understanding requires patience, and that the reader solves the posed exercises.

"SLIDE 941"

We have also, intertwined with the incremental presentation of design details, presented method principles and techniques. The stepwise unfolding is one such principle. In each step, especially in the transition from one step to the next, there were subprinciples and techniques. These include being concerned about, and hence defining invariants (i.e., well-formedness); relating "a more concrete step" to its "more abstract predecessor step", and hence defining abstraction functions; and being concerned about correctness of a stepwise refinement or extension, and hence in establishing correctness criteria. As a matter of principle, a metaprinciple, this book does not show actual proofs of correctness.

"SLIDE 942"

The major lesson of this section can be summarised: When designing systems with a great many concepts, do so in stages of refinement and extension. That is, introduce a few concepts at a time. Sketch invariants, abstraction functions, and correctness criteria. Basically try to redefine the semantic functions in every step. It is a good way to see whether one has confidence in the present step's design decisions.

Principles and Techniques

"SLIDE 943"

We summarise:

Principle 14 (Component Development, Stepwise Discovery, I) And yet another principle of *component development* is that of possibly helping to *discover* new, initially, i.e., during requirements development, unforeseen requirements. ∎

"SLIDE 944"

Principle 15 (Component Development, Stepwise Extension, II) One possibility of *component development* is that of the *stepwise unfolding* of externally observable properties. That is, of the extension of an architecture that handles some, but not all requirements, to increasingly cater for additional requirements. ∎

"SLIDE 945"

Principle 16 (Component Development, Stepwise Refinement, III) Another principle of component development is that of *stepwise refinement* or *stepwise extension:* Either making more concrete a data type while redefining operations over any such data type (refinement), or "adding" additional operations (extension). And doing this in up to several steps. ∎

"SLIDE 946"

Technique 4 (Component Development) The corresponding techniques of *component development* include development of invariants (well-formedness), abstraction (and injection) functions, adequacy and sufficiency relations, more concretised operation (action) definitions, and statements and possible proofs of correctness. ∎

Bibliographical Notes "SLIDE 947"

The file system outlined in this chapter is based on Stoy and Strachey's utterly elegant operating system OS6 [212]. Work on the current model was prompted by Abrial's approach as documented in item (4) of [6].

Exercise 27. kap4.xs.1:
Solution 27 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 28. kap4.xs.2:
Solution 28 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 29. kap4.xs.3:
Solution 29 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 30. kap4.xs.4:

Solution 30 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 31. kap4.xs.5:
Solution 31 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 32. kap4.xs.6:
Solution 32 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 33. kap4.xs.7:
Solution 33 Vol. II, Page 535, suggests a way of answering this exercise.

Exercise 34. kap4.xs.8:
Solution 34 Vol. II, Page 535, suggests a way of answering this exercise.

"SLIDE 964"

Part III

A Review of The Triptych Approach to SE

5

Closing

"slide 965"

5.1 Domains, Requirements, Software Design

"slide 966"

5.2 Process Graphs

"slide 967"

5.3 Documents

"slide 968"

5.4 Process Assessment and Improvement

"slide 969"

"slide 970"

Part IV

Administrative Appendices

A

Bibliographical Notes

Specification languages, techniques and tools, that cover the spectrum of domain and requirements specification, refinement and verification, are dealt with in Alloy: [122], ASM: [188, 189], B/event B: [4, 53], CSP [110, 194, 196, 111], DC [228, 229] (Duration Calculus), Live Sequence Charts [63, 102, 126], Message Sequence Charts [118, 119, 120], RAISE [85, 87, 33, 34, 35, 84, 41] (RSL), Petri nets [124, 173, 185, 184, 186], Statecharts [98, 99, 101, 103, 100], Temporal Logic of Reactive Systems [149, 150, 165, 180], TLA+ [131, 132, 155, 156] (Temporal Logic of Actions), VDM [44, 45, 78, 77], and Z [203, 204, 226, 108, 107]. Techniques for integrating "different" formal techniques are covered in [13, 91, 51, 49, 193]. The recent book on Logics of Specification Languages [43] covers ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM and Z.

References

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass., USA, 1996. 2nd edition.
2. G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. *SIGSOFT Software Engineering Notes*, 18(5):9–20, December 1993.
3. G.D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, Oct 1995.
4. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
5. Jean-Raymond Abrial and L. Mussat. *Event B Reference Manual (Editor: Thierry Lecomte)*, June 2001. Report of EU IST Project Matisse IST-1999-11435.

6. J.R. Abrial. (1) The Specification Language Z: Basic Library, 30 pgs.; (2) The Specification Language Z: Syntax and "Semantics", 29 pgs.; (3) An Attempt to use Z for Defining the Semantics of an Elementary Programming Language, 3 pgs.; (4) A Low Level File Handler Design, 18 pgs.; (5) Specification of Some Aspects of a Simple Batch Operating System, 37 pgs. Internal reports, Programming Research Group, April-May 1980.
7. R. Allen and D. Garlan. A formal approach to software architectures. In *IFIP Transactions A (Computer Science and Technology); IFIP Wordl Congress; Madrid, Spain*, volume vol.A-12, pages 134–141, Amsterdam, Netherlands, 1992. IFIP, North Holland.
8. R. Allen and D. Garlan. Formalizing architectural connection. In *16th International Conference on Software Engineering (Cat. No.94CH3409-0); Sorrento, Italy*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.
9. R. Allen and D. Garlan. A case study in architectural modeling: the AEGIS system. In *8th International Workshop on Software Specification and Design; Schloss Velen, Germany*, pages 6–15, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.
10. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS. Springer–Verlag, 1998.
11. Krzysztof R. Apt. *From Logic Programming to Prolog*. Prentice Hall, International Series in Computer Science, 1997. viii + 328 pages.
12. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, August 2003. ISBN 0521825830.
13. 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.
14. 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.
15. Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, US, 1996.
16. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Heidelberg, Germany, 1998.
17. John W. Backus and Peter Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–1, 1963.
18. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version I. Technical report, IBM Laboratory, Vienna, 1966.
19. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version II. Technical report, IBM Laboratory, Vienna, 1968.
20. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version III. IBM Laboratory, Vienna, 1969.
21. Claude Berge. *Théorie des Graphes et ses Applications*. Collection Universitaire de Mathematiques. Dunod, Paris, 1958. See [22].
22. Claude Berge. *Graphs*, volume 6 of *Mathematical Library*. North-Holland Publ. Co., second revised edition of part 1 of the 1973 english version edition, 1985. See [21].
23. Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

24. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, September 1996.
25. G.M. Birtwistle, O.-J.Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA* `begin`. Studentlitteratur, Lund, Sweden, 1974.
26. Dines Bjørner. Programming in the Meta-Language: A Tutorial. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language, [44]*, LNCS, pages 24–217. Springer–Verlag, 1978.
27. Dines Bjørner. Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language, [44]*, LNCS, pages 337–374. Springer–Verlag, 1978.
28. Dines Bjørner. The Vienna Development Method: Software Abstraction and Program Synthesis. In *Mathematical Studies of Information Processing*, volume 75 of *LNCS*. Springer–Verlag, 1979. Proceedings of Conference at Research Institute for Mathematical Sciences (RIMS), University of Kyoto, August 1978.
29. Dines Bjørner, editor. *Abstract Software Specifications*, volume 86 of *LNCS*. Springer–Verlag, 1980.
30. Dines Bjørner. Application of Formal Models. In *Data Bases*. INFOTECH Proceedings, October 1980.
31. Dines Bjørner. Formalization of Data Base Models. In Dines Bjørner, editor, *Abstract Software Specification, [29]*, volume 86 of *LNCS*, pages 144–215. Springer–Verlag, 1980.
32. Dines Bjørner. Documents: A Domain Analysis. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
33. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
34. 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.
35. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
36. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
37. Dines Bjørner. Transportation Systems Development. In *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation; Special Workshop Theme: Formal Methods in Avionics, Space and Transport*, ENSMA, Futuroscope, France, December 12–14 2007.
38. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2008. (This is a new journal, published by Taylor & Francis, New York and London, edited by Philip Laplante).
39. Dines Bjørner. Domain Engineering. In *BCS FACS Seminars*, Lecture Notes in Computer Science, the BCS FAC Series (eds. Paul Boca and Jonathan Bowen), pages 1–42, London, UK, 2008. Springer. To appear.

40. Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.
41. Dines Bjørner. *Software Engineering, Vol. I: The Triptych Approach, Vol. II: A Model Development*. To be submitted to Springer for evaluation, expected published 2009. This book is the basis for guest lectures at Techn. Univ. of Graz, Politecnico di Milano, University of the Saarland (Germany), etc., 2008–2009.
42. Dines Bjørner and Aser Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering. In *Festschrift for Prof. Willem Paul de Roever (Eds. Martin Steffen, Dennis Dams and Ulrich Hannemann*, volume [not known at time of submission of the current paper] of *Lecture Notes in Computer Science (eds. Martin Steffen, Dennis Dams and Ulrich Hannemann)*, pages 1–12, Heidelberg, July 2008. Springer.
43. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [189, 53, 70, 162, 96, 84, 156, 77, 107]*. EATCS Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
44. 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*. [26, 27, 28].
45. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
46. Dines Bjørner and Hans Henrik Løvengreen. Formal Semantics of Data Bases. In *8th Int'l. Very Large Data Base Conf.*, Mexico City, Sept. 8-10 1982.
47. Dines Bjørner and Hans Henrik Løvengreen. Formalization of Data Models. In *Formal Specification and Software Development, [45]*, chapter 12, pages 379–442. Prentice-Hall, 1982.
48. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
49. 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.
50. J. P. Bowen. Glossary of Z notation. *Information and Software Technology*, 37(5–6):333–334, May–June 1995.
51. 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.
52. 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: [188, 69, 163, 86, 155, 108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
53. Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages 47–152 in [43]. Springer, 2008.
54. D. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal*

Description Techniques, II (FORTE'89), pages 281–296. Elsevier Science Publishers (North-Holland), 1990.
55. Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml*. Project Cristal, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France, 2004. Preliminary translation of the book Développement d'applications avec Objective Caml [56].
56. Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. Éditions O'Reilly, Paris, France, Avril 2000. ISBN 2-84177-121-0.
57. Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, Cambridge, UK, 1998. ISBN 0-521-57183-9 (hardcover), 0-521-57681-4 (paperback).
58. David Crystal. *The Cambridge Encyclopedia of Language*. Cambridge University Press, 1987, 1988.
59. CVS. Concurrent Versions System Home Page. Electronically, on the Web: `www.cvshome.org`, 2005.
60. O.-J. Dahl, Edsger Wybe Dijkstra, and Charles Anthony Richard Hoare. *Structured Programming*. Academic Press, 1972.
61. O.-J. Dahl and Charles Anthony Richard Hoare. Hierarchical program structures. In *[60]*, pages 197–220. Academic Press, 1972.
62. O.-J. Dahl and K. Nygaard. SIMULA – an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.
63. 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.
64. C.J. Date. *An Introduction to Database Systems, I*. The Systems Programming Series. Addison Wesley, 1981.
65. C.J. Date. *An Introduction to Database Systems, II*. The Systems Programming Series. Addison Wesley, 1983.
66. Jim Davies. Announcement: Electronic version of Communicating Sequential Processes (CSP). Published electronically: `http://www.usingcsp.com/`, 2004. Announcing revised edition of [110].
67. J.W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
68. Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing - Vol. 6. World Scientific Publishing Co., Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, July 1998. 196pp, ISBN 981-02-3513-5, US$30.
69. 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: [188, 52, 163, 86, 155, 108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
70. Răzvan Diaconescu. *Logics of Specification Languages*, chapter A Methodological Guide to the CafeOBJ Logic, pages 153–240 in [43]. Springer, 2008.

71. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
72. D. J. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 244–261. VDM-Europe, Springer-Verlag, 1990.
73. R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and Meyer B, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice Hall, 1991.
74. 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. .
75. R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, Cambridge, Mass., USA, 2003. 3rd Edition.
76. 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.
77. John S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages 453–487 in [43]. Springer, 2008.
78. John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using `VDM-SL`*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.
79. FOLDOC: The free online dictionary of computing. Electronically, on the Web: `http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?ISWIM`, 2004.
80. D. Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, June 1995.
81. D. Garlan. Formal approaches to software architecture. In *Studies of Software Design. ICSE '93 Workshop. Selected Papers*, pages 64–76, Berlin, Germany, 1996. Springer-Verlag.
82. D. Garlan and M. Shaw. Experience with a course on architectures for software systems. In *Software Engineering Education. SEI Conference 1992; San Diego, CA, USA*, pages 23–43, Berlin, Germany, 199. Springer-Verlag.
83. D. Garlan and M. Shaw. *An introduction to software architecture*, pages 1–39. World Scientific, Singapore, 1993.
84. Chris George and Anne E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages 349–399 in [43]. Springer, 2008.
85. 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.
86. 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: [188, 52, 69, 163, 155, 108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
87. 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.

88. Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2002. 2nd Edition.
89. James Gosling and Frank Yellin. *The Java Language Specification*. Addison--Wesley & Sun Microsystems. ACM Press Books, 1996. 864 pp, ISBN 0-10-63451-1.
90. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
91. 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.
92. Oliver Grillmeyer. *Exploring Computer Science with Scheme*. Springer-Verlag, New York, USA, 1998.
93. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
94. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.
95. Kirsten M. Hansen. *Linking Safety Analysis to Safety Requirements*. PhD thesis, Department of Computer Science, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1996.
96. Michael R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages 299–347 in [43]. Springer, 2008.
97. Michael Reichhardt Hansen and Hans Rischel. *Functional Programming in Standard ML*. Addison Wesley, 1997.
98. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
99. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
100. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
101. 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.
102. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
103. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
104. E.C.R. Hehner. *The Logic of Programming*. Prentice-Hall, 1984.
105. E.C.R. Hehner. *a Practical Theory of Programming*. Springer-Verlag, 2nd edition, 1993. On the net: http://www.cs.toronto.edu/~hehner/aPToP/.
106. Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Microsoft •net Development Series. Addison-Wesley, 75 Arlington Street, Suite 300, Boston, MA 02116, USA, (617) 848-6000, 30 October 2003. 672 page, ISBN 0321154916.
107. Martin C. Henson, Moshe Deutsch, and Steve Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages 489–596 in [43]. Springer, 2008.

108. 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: [188, 52, 69, 163, 86, 155] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
109. Jaakko Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, Ithaca, N.Y., USA, June 1962. ASIN 0801401879.
110. Tony Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
111. Tony Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [110]. See also `http://www.usingcsp.com/`.
112. Christopher John Hogger. *Essentials of Logic Programming*. Graduate Texts in Computer Science, no.1, 310 pages. Clarendon Press, December 1990. .
113. 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.
114. IEEE CS. IEEE Standard Glossay of Software Engineering Terminology, 1990. IEEE Std.610.12.
115. IEEE: The Institute for Electrical and Electronics Engineers. The IEEE Home Page. Electronically, on the Web: `http://www.ieee.org`, 2005.
116. Inmos Ltd. Specification of instruction set & Specification of floating point unit instructions. In *Transputer Instruction Set – A compiler writer's guide*, pages 127–161. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1988.
117. ISO: The International Standards Organisation. The ISO Home Page. Electronically, on the Web: `http://www.iso.org`, 2005.
118. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
119. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
120. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
121. ITU: The International Telecommunications Union. The ITU Home Page. Electronically, on the Web: `http://www.itu.org`, 2005.
122. Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
123. Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.
124. 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.
125. Brian Kernighan and Dennis Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1989.
126. 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.

127. D.E. Knuth. *The Art of Computer Programming, Vol.1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., USA, 1968.
128. D.E. Knuth. *The Art of Computer Programming, Vol.2.: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., USA, 1969.
129. D.E. Knuth. *The Art of Computer Programming, Vol.3: Searching & Sorting*. Addison-Wesley, Reading, Mass., USA, 1973.
130. Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery (Eds.: J. Worrall and E. G. Zahar)*. Cambridge University Press, The Edinburgh Building, Shaftesbury Road, Cambridge CB2 2RU, England, 2 September 1976. ISBN: 0521290384. Published in 1963-64 in four parts in the British Journal for Philosophy of Science. (Originally Lakatos' name was Imre Lipschitz.).
131. Leslie Lamport. The Temporal Logic of Actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995.
132. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
133. J.C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *15th. Int. Symp. on Fault-tolerant computing*. IEEE, 1985.
134. 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.
135. J.A.N. Lee. *Computer Semantics*. Van Nostrand Reinhold Co., 1972.
136. J.A.N. Lee and W. Delmore. The Vienna Definition Language, a generalization of instruction definitions. In *SIGPLAN Symp. on Programming Language Definitions, San Francisco*, Aug. 1969.
137. H.S. Leonard and N. Goodman. The Calculus of Individuals and Its Uses. *Journal of Symbolic Logic*, 5:45–55, 1940.
138. Xavier Leroy and Pierre Weis. *Manuel de Référence du langage Caml*. InterEditions, Paris, France, 1993. ISBN 2-7296-0492-8.
139. Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley & Sun Microsystems. ACM Press Books, 1996. 496 pp, ISBN 0-10-63452-X.
140. W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.
141. Zhiming Liu, A.P. Ravn, E.V. Sørensen, and Chao Chen Zhou. A probabilistic duration calculus. In H. Kopetz and Y. Kakuda, editors, *Responsive Computer Systems*, volume 7 of *Dependable Computing and Fault-Tolerant Systems*, pages 30–52. Springer Verlag Wien New York, 1993.
142. J.W. Lloyd. *Foundation of Logic Programming*. Springer-Verlag, 1984.
143. P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Devt. and Res.*, 25(5):549–561, 1981.
144. P. Lucas and K. Walk. On the formal description of PL/I. *Annual Review Automatic Programming Part 3*, 6(3), 1969.
145. E.C. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
146. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, NY, USA, 1992.
147. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, New York, NY, USA, 1995.

148. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Progress*. Unpublished, Stanford University, Computer Science Department, http://theory.stanford.edu/~zm/tvors3.html, 2004.
149. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.
150. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.
151. ANSI X3.23-1974. The Cobol programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1974.
152. ANSI X3.53-1976. The PL/I programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1976.
153. ANSI X3.9-1966. The Fortran programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1966.
154. J. McCarthy and et al. *LISP 1.5, Programmer's Manual*. The MIT Press, Cambridge, Mass., USA, 1962.
155. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [188, 52, 69, 163, 86, 108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
156. Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages 401–451 in [43]. Springer, 2008.
157. Microsoft Corporation. *MCAD/MCSD Self-Paced Training Kit: Developing Web Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET*. Microsoft Corporation, Redmond, WA, USA, 2002. 800 pages.
158. Microsoft Corporation. *MCAD/MCSD Self-Paced Training Kit: Developing Windows-Based Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET*. Microsoft Corporation, Redmond, WA, USA, 2002.
159. D. Miéville and D. Vernant. *Stanisław Leśniewksi aujourd'hui*. Grenoble, October 8-10, 1992.
160. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., USA and London, England, 1990.
161. C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.
162. T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. *Logics of Specification Languages*, chapter CASL – the Common Algebraic Specification Language, pages 241–298 in [43]. Springer, 2008.
163. 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: [188, 52, 69, 86, 155, 108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
164. Peter D. Mosses, editor. CASL *Reference Manual*, volume 2960 of *LNCS, IFIP Series*. Speinger–Verlag, Heidelberg, Germnay, 2004. Part I (Summary) and Part II (Syntax): Peter Mosses; Part III (Semantics): Don Sannella, and Andrzej Tarlecki; Parts IV (Logic), V (Refinement) and VI (Libraries): Till Mossakowski.

165. Ben C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.
166. Jørgen Fischer Nilsson. Some Foundational Issues in Ontological Engineering, October 30 – Novewmber 1 2002. Lecture slides for a PhD Course in *Representation Formalisms for Ontologies*, Copenhagen, Denmark.
167. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
168. Object Management Group. *OMG Unified Modelling Language Specification*. OMG/UML, http://www.omg.org/uml/, version 1.5 edition, March 2003. www.omg.org/cgi-bin/doc?formal/03-03-01.
169. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
170. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
171. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
172. David Lorge Parnas. A technique for software module specification with examples. *Communications of the ACM*, 14(5), May 1972.
173. Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
174. Charles Petzold. *Programming Windows with C# (Core Reference)* . Microsoft Corporation, Redmond, WA, USA, 2001. 1200 pages.
175. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.
176. Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Comp. Sci. Dept., Aarhus Univ., Denmark; DAIMI-FN-19, 1981. Definitive version of this seminal report is (to be) published in a special issue of the *Journal of Logic and Algebraic Programming* (eds. Jan Bergstra and John Tucker) devoted to a workshop on SOS: Structural Operational Semantics, a Satellite Event of CONCUR 2004, August 30, 2004, London, United Kingdom. Look also for Gordon Plotkin's introductory paper for that issue: The Origins of Structural Operational Semantics.
177. Gordon D. Plotkin. Structural operational semantics. Lecture notes, Aarhus University, DAIMI FN-19. Reprinted 1991, 1981. See [179, 178].
178. Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July-December 2004. See [177, 179].
179. Gordon D. Plotkin. A structural approach operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July-December 2004. Widely disseminated since 1981 as [177]. See also [178].
180. 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. .
181. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.

182. 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.
183. Wolfgang Reif. Integrated formal methods for safety analysis. In *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain*, IFIP, Amsterdam, The Netherlands, 2004. Kluwer Academic Press.
184. Wolfang Reisig. *A Primer in Petri Net Design*. Springer Verlag, March 1992. 120 pages.
185. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.
186. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.
187. Wolfgang Reisig. On Gurevich's Theorem for Sequential Algorithms. *Acta Informatica*, 2003.
188. Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [52, 69, 163, 86, 155, 108] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
189. Wolfgang Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [43]. Springer, 2008.
190. John C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
191. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
192. P.M. Roget. *Roget's Thesaurus*. Collins, London and Glasgow, 1852, 1974.
193. 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.
194. 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.
195. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
196. 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.
197. Peter Sestoft. *Java Precisely*. The MIT Press, 25 July 2002. 100 pages (sic !), ISBN 0262692767.
198. C. Shekaran, D. Garlan, and et al. The role of software architecture in requirements engineering. In *First International Conference on Requirements Engineering (Cat. No.94TH0613-0); Colorado Springs, CO, USA*, pages 239–245, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.
199. Peter M. Simons. *Foundations of Logic and Linguistics: Problems and their Solutions*, chapter Leśniewski's Logic and its Relation to Classical and Free Logics. Plenum Press, New York, 1985. Georg Dorn and P. Weingartner (Eds.).
200. Jens Ulrik Skakkebæk, Anders Peter Ravn, Hans Rischel, and Chao Chen Zhou. Specification of embedded, real-time systems. In *Proceedings of 1992 Euromicro Workshop on Real-Time Systems*, pages 116–121. IEEE Computer Society Press, 1992.

201. Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 1982–2001.
202. John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pws Pub Co, August 17, 1999. ISBN: 0534949657, 512 pages, Amazon price: US $ 70.95.
203. 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.
204. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
205. J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.
206. J.T.J. Srzednicki and Z. Stachniak, editors. *Leśniewksi's Lecture Notes in Logic*. Dordrecht, 1988.
207. J.T.J. Srzednicki and Z. Stachniak. *Leśniewksi's Systems Protothetic*. . Dordrecht, 1998.
208. Staff of Encyclopœdia Brittanica. Encyclopœdia Brittanica. Merriam Webster/Brittanica: Access over the Web: http://www.eb.com:180/, 1999.
209. 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.
210. Staff of Oxford University Press. *The Oxford Dictionary of Quotations*. Oxford University Press, London, 1941, 1974.
211. Jess Stein, editor. *The Random House American Everyday Disctionary*. Random House, New York, N.Y., USA, 1949, 1961.
212. J.E. Stoy and C. Strachey. OS6 – an experimental operating system for a small computer, part 1: general principles and strucure, and part 2: input-output and filing system. *Computer Journal*, 15(2-3):117–124, 194–203, 1972.
213. B. Stroustrup. *C++ Programming Language*. Addison-Wesley Publishing Company, 1986.
214. S. J. Surma, J. T. Srzednicki, D. I. Barnett, and V. F. Rickey, editors. *Stanisław Leśniewksi: Collected works (2 Vols.)*. Dordrecht, Boston – New York, 1988.
215. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
216. Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 2nd edition, 29 March 1999. 512 pages, ISBN 0201342758.
217. D.A. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.
218. Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
219. F. Van der Rhee, H.R. Van Nauta Lemke, and J.G. Dukman. Knowledge based fuzzy control of systems. *IEEE Trans. Autom. Control*, 35(2):148–155, February 1990.

220. Rob van Glabbeek and Peter Weijland. Branching Time and Abstraction in Bisimulation Semantics. Electronically, on the Web: `http://-theory.stanford.edu/~rvg/abstraction/abstraction.html`, Centrum voor Wiskunde en Informatica, Postbus 94079, 1090 GB Amsterdam, The Netherlands, January 1996.
221. Peter van Roy and Seif Haridi. *Concepts, Techniques and Models of Computer Programming.* The MIT Press, Cambridge, Mass., USA, March 2004.
222. Hans van Vliet. *Software Engineering: Principles and Practice.* John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 2000. 2nd Edition.
223. Bill Venners. *Inside the Java 2.0 Virtual Machine (Enterprise Computing).* McGraw-Hill; ISBN: 0071350934, October 1999.
224. Pierre Weis and Xavier Leroy. *Le langage Caml.* Dunod, Paris, France, 1999. ISBN 2-10-004383-8, Second edition.
225. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, Cambridge, Mass., USA, 1993.
226. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.
227. Edward N. Zalta. Logic. In *The Stanford Encyclopedia of Philosophy.* Published: http://plato.stanford.edu/, Winter 2003.
228. 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.
229. Chao Chen Zhou, Charles Anthony Richard Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.
230. P.A. Lee, T. Anderson: *Fault Tolerance, Principles and Practice* (Springer 1990)
231. P.G. Bishop (ed.): *Dependability of Critical Computer Systems.* Vol. 3, Elsevier Applied Science (1988)
232. J. Dugan, S. Bavuso, M. Boyd: *Fault Trees and Markov Models for Reliability Analysis of Fault-Tolerant Digital Systems.* In: Reliability Engineering and System Safety, **39**:291–307 (1993)
233. M.R. Lyu: *Software Fault Tolerance* (Chichester, UK, 1995)
234. J.R. Taylor: *A Background to Risk Analysis.* Volume 2, Technical Report, Risø Research Center, Denmark (1979)
235. K.S. Trivedi: *Probability and Statistics with Reliability, Queueing and Computer Science Application.* Prentice Hall (1982)
236. E. Troubitsyna: *Specifying Safety-Related Hazards Formally.* Åbo Akademi University, Department of Computer Science, TUCS Technical Report No 270 (1999)
237. US Nuclear Regulatory Commission: *Fault Tree Handbook.* Washington, DC, USA (1981)

B

Indexes

B.1 Index of Concepts

B.2 Index of Domain Terms

B.3 Index of Examples

B.4 Index of Definitions

B.5 Index of Principles

B.6 Index of Techniques

B.7 Index of Tools

B.8 Index of Symbols

C

Glossary

C.1 Categories of Reference Lists

On Glossaries, Dictionaries, Encyclopædia, Ontologies, Taxonomies, Terminologies and Thesauri

An important function of glossaries, dictionaries, etc., is to make sure that terms that may seem esoteric do not remain so.

> **Esoteric:** designed for or understood by the specially initiated alone, of or relating to knowledge that is restricted to a small group, limited to a small circle
>
> *Merriam–Webster's Collegiate Dictionary [209]*

C.1.1 Glossary

According to [140] a *gloss* is "a word inserted between the lines or in the margin as an explanatory rendering of a word in the text; hence a similar rendering in a glossary or dictionary. Also, a comment, explanation, interpretation." Furthermore according to [140] a *glossary* is therefore "a collection of glosses, a list with explanations of abstruse, antiquated, dialictical, or technical terms; a partial dictionary." [50] provides a *Glossary of Z Notation.*

C.1.2 Dictionary

According to [140] a *dictionary* is "a book dealing with the words of a language, so as to set forth their orthography, pronunciation, signification, and use, their synonyms, derivation, history, or at least some of these; the words are arranged in some stated order, now, usually, alphabetical; a word book, vocabulary, lexicon. And, by extension: A book on information or reference, on any subject or branch of knowledge, the items of which are arranged alphabetically." Standard dictionaries are [209, 210, 140].

C.1.3 Encyclopædia

According to [140], an *encyclopædia* is "a circle of learning, a general course of instruction. A work containing information on all branches of knowledge, usually arranged alphabetically (1644). A work containing exhaustive information on some one art or branch of knowledge, arranged systematically." [208] is, perhaps, the most "famous" encyclopædia.

C.1.4 Ontology

By *ontology* is meant [140]: "the science or study of being; that department of metaphysics which relates to the being or essence of things, or to being in the abstract." By *an ontology* we shall mean a document which, in a systematic arrangement explains, in a logical manner, a number of abstract concepts.

C.1.5 Taxonomy

By *taxonomy* is meant [140]: "classification, especially in relation to its general laws or principles; that department of science, or of a particular science or subject, which consists in or relates to classification."

C.1.6 Terminology

By a *term* is here meant [140]: "a word or phrase used in a definite or precise sense in some particular subject, as a science or art; a technical expression." More widely: *"Any word or group of words expressing a notion or conception, or denoting an object of thought."* By *terminology* is meant [140]: "the doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature." [133] provides a terminology of *Dependable Computing and Fault Tolerance: Concepts and Terminology*.

C.1.7 Thesaurus

By *thesaurus* is, in general, meant [140]: "a 'treasury' or 'storehouse' of knowledge, as a dictionary, encyclopædia or the like. (1736)" The thesaurus [192] has set a unique standard for and "the" meaning, now, of the term 'thesaurus'.

C.2 Typography and Spelling

Some comments are in order:

- A term *definition* consists of two or three parts.

  - ⋆ The first part consists of a natural (the index) number, the term being defined and a colon (:). The term subpart is the *definiendum.*
  - ⋆ The second part is the term definition body, the *definiens.*
  - ⋆ Optional third parts — in parentheses — expand on the definiens, contrast it to other terms, or other.
- The definiendum is a one, two or three word **boldfaced term**.
- The definiens consists of free text which may contain uses of (other, or the same) defined terms.
- *Terms* written in *sans serif italicized font* stand for defined terms.
- Definiens (second part) text ending with [209] (or [140]) represents quotes.
- For reasons of cross-referencing we have spelled the terms α, β and λ as Alpha (alpha), Beta (beta) and Lambda (lambda).
- And we have rewritten the technical terms α-renaming, β-reduction and λ-calculus, conversion and expression (etc.) into Alpha-renaming, Beta-reduction and Lambda-expression, etc., while keeping the hyphens.

C.3 The Glosses

$\mathcal{A}$

1 **Abstract:** Something which focuses on essential properties. Abstract is a relation: something is abstract with respect to something else (which possesses — what is considered — inessential properties).
2 **Abstract data type:** An *abstract data type* is a set of values for which no external world or computer (i.e., data) representation is being defined, together with a set of abstractly defined functions over these data values.
3 **Abstraction:** 'The art of abstracting. The act of separating in thought; a mere idea; something visionary.'
4 **Abstraction function:** An *abstraction function* is a function which applies to *value*s of a *concrete type* and yields values of — what is said to be a corresponding — *abstract type*. (Same as *retrieve function*.)
5 **Abstract syntax:** An *abstract syntax* is a set of rules, often in the form of an *axiom system*, or in the form of a set of *sort definition*s, which defines a set of structures without prescribing a precise external world, or a computer (i.e., data) representation of those structures.
6 **Abstract type:** An *abstract type* is the same as an *abstract data type*, except that no functions over the data values have been specified.
7 **Acquirer:** The legal entity, a person, an institution or a firm which orders some *development* to take place. (Synonymous terms are *client* and *customer*.)
8 **Acquisition:** The common term means purchase. Here we mean the collection of *knowledge* (about a *domain*, about some *requirements*, or about some *software*). This collection takes place in an interaction between the

*developer*s and representatives of the *client* (*user*s, etc.). (A synonym term is *elicitation*.)

9 **Action:** By an action we shall understand something who potentially changes a *state*.

10 **Active:** By active is understood a *phenomenon* which, over *time*, changes *value*, and does so either by itself, *autonomous*ly, or also because it is "instructed" (i.e., is "bid" (see *biddable*), or "programmed" (see *programmable*) to do so). (Contrast to *inert* and *reactive*.)

11 **Actuator:** By an actuator we shall understand an electronic, a mechanical, or an electromechanical device which carries out an *action* that influences some physical *value*. (Usually actuators, together with *sensor*s, are placed in *reactive* systems, and are linked to *controller*s. Cf. *sensor*.)

12 **Adaptive:** By adaptive we mean some thing that can adapt or arrange itself to a changing *context*, a changing *environment*.

13 **Adaptive maintenance:** By adaptive maintenance we mean an update, as here, of software, to fit (to adapt) to a changing environment. (Adaptive maintenance is required when new input/output media are attached to the existing software, or when a new, underlying database management system is to be used (instead of an older such), etc. We also refer to *corrective maintenance*, *perfective maintenance*, and *preventive maintenance*.)

14 **Agent:** By an agent we mean the same as an *actor* — a human or a machine (i.e., robot). (The two terms *actor* and *agent* are here considered to be synonymous.)

15 **Algorithm:** The notion of an algorithm is so important that we will give a number of not necessarily complementary definitions, and will then discuss these.

- By an algorithm we shall understand a precise prescription for carrying out an orderly, finite set of *operation*s on a set of *data* in order to calculate (*compute*) a result. (This is a version of the classical definition. It is compatible with computability in the sense of *Turing machine*s and *Lambda-calculus*. Other terms for algorithm are: effective procedure, and abstract program.)
- Let there be given a possibly infinite set of *state*s, S, let there be given a possibly infinite set of initial states, I, where $I \subseteq S$, and let there be given a next state function $f : S \to S$. (C, where $C = (Q, I, f)$ is an initialised, *deterministic transition* system.) A sequence $s_0, s_1, \ldots, s_{i-1}, s_i, \ldots, s_m$ such that $f(s_{i-1}) = s_i$ is a *computation*. An algorithm, A, is a C with final states O, i.e.: $A = (Q, I, f, O)$, where $O \subseteq S$, such that each computation ends with a state s_m in O. (This is basically Don Knuth's definition [127]. In that definition a state is a collection of identified data, i.e., a formalised representation of information, i.e., of computable data. Thus Knuth's definition is still Turing and Lambda-calculus "compatible".)
- There is given the same definition as just above with the generalisation that a state is any association of variables to phenomena, whether

the latter are representable "inside" the computer or not. (This is basically Yuri Gurevitch's definition of an algorithm [94, 187, 188]. As such this definition goes beyond Turing machine and Lambda-calculus "compatibility". That is, captures more!)

16 **Algorithmic:** Adjective form of *algorithm*.
17 **Ambiguous:** A *sentence* is ambiguous if it is open to more than one *interpretation*, i.e., has more than one *model* and these models are not *isomorphic*.
18 **Analysis:** The resolution of anything complex into simple elements. A determination of proper components. The tracing of things to their sources; the discovery of general principles underlying concrete phenomena [140]. (In conventional mathematics analysis pertains to continuous phenomena, e.g. differential and integral calculi. Our analysis is more related to hybrid systems of both discrete and continuous phenomena, or often to just discrete ones.)
19 **Application:** By an application we shall understand either of two rather different things: (i) the application of a function to an *argument*, and (ii) the use of software for some specific purpose (i.e., the application). (See next entry for variant (ii).)
20 **Application domain:** An area of activity which some *software* is to support (or supports) or partially or fully automate (resp. automates). (We normally omit the prefix 'application' and just use the term *domain*.)
21 **Applicative:** The term applicative is used in connection with applicative programming. It is hence understood as programming where applying functions to *argument*s is a main form of expression, and hence designates function application as a main form of operation. (Thus the terms applicative and *functional* are here used synonymously.)
22 **Applicative programming:** See the term *applicative* just above. (Thus the terms applicative programming and *functional programming* are here used synonymously.)
23 **Applicative programming language:** Same as *functional programming language*.
24 **Architecture:** The structure and content of *software* as perceived by their *user*s and in the context of the *application domain*. (The term architecture is here used in a rather narrow sense when compared with the more common use in civil engineering.)
25 **Artefact:** An artificial product [140]. (Anything designed or constructed by humans or machines, which is made by humans.)
26 **Artifact:** Same term as *artefact*.
27 **Assertion:** By an assertion we mean the act of stating positively usually in anticipation of denial or objection. (In the context of *specification*s and *program*s an assertion is usually in the form of a pair of *predicate*s "attached" to the specification text, to the program text, and expressing properties that are believed to hold before any interpretation of the text;

that is, "a before" and "an after", or, as we shall also call it: a **pre-** and a **post-**condition.)

28 **Atomic:** In the context of software engineering atomic means: A *phenomenon* (a *concept*, an *entity*, a *value*) which consists of no proper subparts, i.e., no proper subphenomena, subconcepts, subentities or subvalues other than itself. (When we consider a phenomenon, a concept, an entity, a value, to be atomic, then it is often a matter of choice, with the choice reflecting a level of abstraction.)

29 **Attribute:** We use the term attribute only in connection with values of composite type. An attribute is now whether a composite value possesses a certain property, or what value it has for a certain component part. (An example is that of database (e.g., `SQL`) relations (i.e., tabular data structures): Columns of a table (i.e., a relation) are usually labelled with a name designating the attribute (type) for values of that column. Another example is that, say, of a Cartesian: $\mathsf{A} = \mathsf{B} \times \mathsf{C} \times \mathsf{D}$. A can be said to have the attributes B, C, and D. Yet other examples are $\mathsf{M} = \mathsf{A} \xrightarrow[m]{} \mathsf{B}$, $\mathsf{S} = \mathsf{A}$**-set** and $\mathsf{L} = \mathsf{A}^*$. M is said to have attributes A and B. S is said to have attribute A. L is said to have attribute A. In general we make the distinction between an entity consisting of subentities (being decomposable into proper parts, cf. *subentity*), and the entities having attributes. A person, like me, has a height attribute, but my height cannot be "composed away from me"!)

30 **Axiom:** An established rule or principle or a self-evident truth.

31 **Axiomatic specification:** A *specification* presented, i.e., given, in terms of a set of *axiom*s. (Usually an axiomatic specification also includes definitions of *sort*s and *function signature*s.)

32 **Axiom system:** Same as *axiomatic specification*.

$\mathcal{B}$

33 **B:** B stands for Bourbaki, pseudonym for a group of mostly French mathematicians which began meeting in the 1930s, aiming to write a thorough unified set-theoretic account of all mathematics. They had tremendous influence on the way mathematics has been done since. (The founding of the Bourbaki group is described in André Weil's autobiography, titled something like "memoir of an apprenticeship" (orig. Souvenirs D'apprentissage). There is a usable book on Bourbaki by J. Fang. Liliane Beaulieu has a book forthcoming, which you can sample in "A Parisian Cafe and Ten Proto-Bourbaki Meetings 1934–1935" in the Mathematical Intelligencer 15 no. 1 (1993) 27–35. From `http://www.faqs.org/faqs/-sci-math-faq/bourbaki/` (2004). Founding members were: Henri Cartan, Claude Chevalley, Jean Coulomb, Jean Delsarte, Jean Dieudonné, Charles Ehresmann, René de Possel, Szolem Mandelbrojt, André Weil. From: `http://www.bourbaki.ens.fr/` (2004). `B` also stands for a model-oriented specification language [4].)

34 **Behaviour:** By behaviour we shall understand the way in which something functions or operates. (In the context of *domain engineering* behaviour is a concept associated with *phenomena*, in particular manifest *entities*. And then behaviour is that which can be observed about the *value* of the *entity* and its *interaction* with an *environment*.)
35 **Boolean:** By Boolean we mean a data type of logical values (**true** and **false**), and a set of connectives: $\sim$, $\wedge$, $\vee$, and $\Rightarrow$. (Boolean derives from the name of the mathematician George Boole.)
36 **Boolean connective:** By a *Boolean connective* we mean either of the Boolean operators: $\wedge$, $\vee$, $\Rightarrow$ (or $\supset$), $\sim$ (or $\neg$).
37 **BPR:** See *business process reengineering*
38 **Brief:** By a brief is understood a *document*, or a part of a document which informs about a *phase* , or a *stage* , or a *step* of *development*. (A brief thus contains *information*.)
39 **Business process:** By a business process we shall understand a *behaviour* of an enterprise, a business, an institution, a factory. (Thus a business process reflects the ways in which a business conducts its affairs, and is a *facet* of the *domain*. Other facets of an enterprise are those of its *intrinsics*, *management and organisation* (a facet closely related, of course, to business processes), *support technology*, *rules and regulations*, and *human behaviour*.)
40 **Business process engineering:** By *business process engineering* we shall understand the *design*, the determination, of *business process*es. (In doing business process engineering one is basically designing, i.e., prescribing entirely new business processes.)
41 **Business process reengineering:** By *business process reengineering* we shall understand the re*design*, the change, of *business process*es. (In doing business process reengineering one is basically carrying out *change management*.)

$\mathcal{C}$

42 **Calculus:** A method of *computation* or *calculation* in a special notation. (From mathematics we know the differential and the integral calculi, and also the Laplace calculus. From metamathematics we have learned of the λ-calculus. From logic we know of the Boolean (propositional) calculus.)
43 **Capture:** The term capture is used in connection with *domain knowledge* (i.e., *domain capture*) and with *requirements acquisition*. It shall indicate the act of acquiring, of obtaining, of writing down, domain knowledge, respectively requirements.
44 **Cartesian:** By a Cartesian is understood an ordered product, a fixed grouping, a fixed composition, of *entities*. (Cartesian derives from the name of the French mathematician René Descartes.)
45 **Channel:** By a channel is understood a means of *interaction*, i.e., of *communication* and possibly of *synchronisation* between *behaviour*s. (In the

context of computing we can think of channels as being either input, or output, or both input and output channels.)

46 **Chaos:** By **chaos** we understand the totally undefined *behaviour*: Anything may happen! (In the context of computing **chaos** may, for example, be the *designation* for the never-ending, the never-terminating *process*.)

47 **Class:** By a class we mean either of two things: a **class** *clause*, as in RSL, or a set of *entities* defined by some *specification*, typically a *predicate*.

48 **Clause:** By a clause is meant an *expression*, designating a *value*, or a *statement*, designating a *state* change, or a sentential form, which designates both a value and a state change. (When we use the term clause we mean it mostly in the latter sense of both designating a value and a side effect.)

49 **Client:** By a client we mean any of three things: (i) The legal body (a person or a company) which orders the development of some software, or (ii) a *process* or a *behaviour* which *interact*s with another process or behaviour (i.e., the *server*), in order to have that server perform some *action*s on behalf of the client, or (iii) a user of some software (i.e., computing system). (We shall normally use the term customer in the first or in the second sense (i, ii).)

50 **Code:** By code we mean a *program* which is expressed in the machine language of a computer.

51 **Coding:** By coding we shall here, simply, mean the act of programming in a machine, i.e., in a computer-close language. (Thus we do not, except where explicitly so mentioned, mean the encoding of one string of characters into another, say for *communication* over a possibly faulty communication *channel* (usually with the decoding of the encoded string "back" into the original, or a similar string).)

52 **Communication:** A *process* by which *information* is exchanged between individuals (*behaviour*s, *process*es) through a common *system* of *symbol*s, *sign*s, or *protocol*s.

53 **Component:** By a component we shall here understand a set of type definitions and component local variable declarations, i.e., a component local state, this together with a (usually complete) set of modules, such that these modules together implement a set of concepts and facilities, i.e., functions, that are judged to relate to one another.

54 **Component design:** By a component design we shall understand the *design* of (one or more) *component*s. (We shall refer to 32829 for "our story" on component design.)

55 **Composite:** We say that a *phenomenon*, a *concept*, is composite when it is possible, and meaningful, to consider that phenomenon or concept as analysable into two or more subphenomena or subconcepts.

56 **Composition:** By composition we mean the way in which a *phenomenon*, a *concept*, is "put together" (i.e., composed) into a *composite phenomenon*, resp. *concept*.

57 **Compositional:** We say that two or more *phenomena* or *concepts* are compositional if it is meaningful to *compose* these phenomena and/or

concepts. (Typically a *denotational semantics* is expressed compositionally: By composing the semantics of sentence parts into the semantics of the composition of the sentence parts.)

58 **Compositional documentation:** By compositional documentation we mean a development, or a presentation (of that development), of, as here, some *description* (*prescription* or *specification*), in which some notion of "smallest", i.e., atomic phenomena and concepts are developed (resp. presented) first, then their compositions, etc., until some notion of full, complete development (etc.) has been achieved. (See also *composition*, *compositional* and *hierarchical documentation*.)

59 **Comprehension:** By comprehension we shall here mean *set*, *list* or *map* comprehension, that is, the expression, of a set, a list, respectively a map, by a predicate over the elements of the set, list or pairings of the map, that belong to the set, list, respectively the map.

60 **Computation:** See *calculation*.

61 **Compute:** Given an expression and an applicable *rule* of a *calculus*, to change the former expression into a resulting expression. (Same as *calculate*.)

62 **Computer Science:** The study and knowledge of the phenomena that can exist inside computers.

63 **Computing Science:** The study and knowledge of how to construct those phenomena that can exist inside computers.

64 **Computing system:** A combination of *hardware* and *software* that together make meaningful *computation*s possible.

65 **Concept:** An abstract or generic idea generalised from phenomena or concepts. (A working definition of a concept has it comprising two components: The *extension* and the *intension*. A word of warning: Whenever we describe something claimed to be a "real instance", i.e., a physical *phenomenon*, then even the description becomes that of a concept, not of "that real thing"!)

66 **Concept formation:** The forming, the enunciation, the *analysis*, and definition of *concepts* (on the basis, as here, of *analysis* of the *universe of discourse* (be it a *domain* or some *requirements*)). (Domain and requirements concept formation(s) is treated in Vol. 3, Chaps. 13 (Domain Analysis and Concept Formation) and 21 (Requirements Analysis and Concept Formation).)

67 **Concrete:** By concrete we understand a *phenomenon* or, even, a *concept*, whose explication, as far as is possible, considers all that can be observed about the phenomenon, respectively the concept. (We shall, however, use the term concrete more loosely: To characterise that something, being specified, is "more concrete" (possessing more properties) than something else, which has been specified, and which is thus considered "more abstract" (possessing fewer properties [considered more relevant]).)

68 **Concrete syntax:** A *concrete syntax* is a syntax which prescribes actual, computer representable *data structure*s. (Typically a *BNF Grammar* is a concrete syntax.)

69 **Concrete type:** A *concrete type* is a type which prescribes actual, computer representable *data structure*s. (Typically the type definitions of programming languages designate concrete types.)

70 **Concurrency:** By concurrency we mean the simultaneous existence of two or more *behaviour*s, i.e., two or more *process*es. (That is, a *phenomenon* is said to exhibit concurrency when one can analyse the phenomenon into two or more *concurrent* phenomena.)

71 **Concurrent:** Two (or more) *event*s can be said to occur concurrently, i.e., be concurrent, when one cannot meaningfully describe any one of these events to ("always") "occur" before any other of these events. (Thus concurrent systems are systems of two or more processes (behaviours) where the simultaneous happening of "things" (i.e., events) is deemed beneficial, or useful, or, at least, to take place!)

72 **Correct:** See next entry: *correctness*.

73 **Correctness:** Correctness is a relation between two specifications A and B: B is correct with respect to A if every property of what is specified in A is a property of B. (Compare to *conformance* and *congruence*.)

74 **Corrective maintenance:** By corrective maintenance we understand a change, predicated by a specification A, to a specification, B', resulting in a specification, B'', such that B'' satisfies more properties of A than does B'. (That is: Specification B' is in error in that it is not *correct* with respect to A. But B'' is an improvement over B'. Hopefully B'' is then correct wrt. A. We also refer to *adaptive maintenance*, *perfective maintenance*, and *preventive maintenance*.)

75 **CSP:** Abbreviation for Communicating Sequential Processes. (See [110, 194] and Chap. 21. Also, but not in this book, a term that covers constraint satisfaction problem (or programming).)

76 **Customer:** By a customer we mean either of three things: (i) the *client*, a person, or a company, which orders the development of some software, or (ii) a *client process* or a *behaviour* which *interact*s with another process or behaviour (i.e., the *server*), in order to have that server perform some *action*s on behalf of the client, or (iii) a user of some software (i.e., computing system). (We shall normally use the term customer in the third sense (iii).)

$\mathcal{D}$

77 **Data:** Data is formalised representation of information. (In our context information is what we may know, informally, and even express, in words, or informal text or diagrams, etc. Data is correspondingly the internal computer, including database representation of such information.)

78 **Database:** By a database we shall generally understand a large collection of data. More specifically we shall, by a database, imply that the data are organised according to certain data structuring and data *query* and *update* principles. (Classically, three forms of (data structured) databases can be identified: The *hierarchical*, the *network*, and the *relation*al database forms. We refer to [64, 65] for seminal coverage, and to [31, 30, 46, 47] for formalisation, of these database forms.)

79 **Database schema:** By a database schema we understand a *type definition* of the structure of the data kept in a database.

80 **Data abstraction:** Data abstraction takes place when we abstract from the particular formal representation of data.

81 **Data invariant:** By a *data* invariant is understood some property that is expected to hold for all instances of the data. (We use the term 'data' colloquially, and really should say type invariance, or variable content invariance. Then 'instances' can be equated with values. See also *constraint*.)

82 **Data refinement:** Data refinement is a relation. It holds between a pair of data if one can be said to be a "more concrete" implementation of the other. (The whole point of *data abstraction*, in earlier *phase*s, *stage*s and *step*s of *development*, is that we can later concretise, i.e., data refine.)

83 **Data reification:** Same as *data refinement*. (To reify is to render something abstract as a material or concrete thing.)

84 **Data structure:** By a data structure we shall normally understand a composition of *data value*s, for example, in the "believed" form of a linked *list*, a *tree*, a *graph* or the like. (As in contrast to an *information structure*, a data structure (by our using the term *data*) is bound to some computer representation.)

85 **Data transformation:** Same as *data refinement* and, hence, *data reification*.

86 **Data type:** By a *data type* is understood a set of *value*s and a set of *function*s over these values — whether *abstract* or *concrete*.

87 **Decidable:** A formal logic system is decidable if there is an *algorithm* which prescribes *computation*s that can determine whether any given sentence in the system is a theorem.

88 **Declaration:** A declaration prescribes the allocation of a resource of the kind declared: (i) A variable, i.e., a location in some storage; (ii) a channel between active processes; (iii) an object, i.e., a process possessing a local state; etc.

89 **Decomposition:** By a decomposition is meant the presentation of the parts of a *composite* "thing".

90 **Definiendum:** The left-hand side of a *definition*, that which is to be defined.

91 **Definiens:** The right-hand side of a *definition*, that which is defining "something".

92 **Definite:** Something which has specified limits. (Watch out for the four terms: *finite*, *infinite*, *definite* and *indefinite*.)

93 **Definition:** A definition defines something, makes it conceptually "manifest". A definition consists of two parts: a *definiendum*, normally considered the left-hand part of a definition, and a *definiens*, normally considered the right-hand part (the body) of a definition.

94 **Definition set:** By a definition set we mean, given a *function*, the set of *value*s for which the function is defined, i.e., for which, when it is *applied* to a member of the definition set yields a proper value. (Cf., *range set*.)

95 **Denotation:** A direct specific meaning as distinct from an implied or associated idea [209]. (By a denotation we shall, in our context, associate the idea of mathematical functions: That is, of the *denotational semantics* standing for functions.)

96 **Denotational:** Being a *denotation*.

97 **Denotational semantics:** By a denotational semantics we mean a *semantics* which to *atomic* syntactical notions associate simple mathematical structures (usually *function*s, or *set*s of *trace*s, or *algebra*s), and which to *composite* syntactical notions prescribe a semantics which is the *functional composition* of the denotational semantics of the *composition* parts.

98 **Denote:** Designates a mathematical meaning according to the principles of *denotational semantics*. (Sometimes we use the looser term designate.)

99 **Dependability:** Dependability is defined as the property of a *machine* such that reliance can justifiably be placed on the service it delivers [182]. (See definition of the related terms: *error*, *failure*, *fault* and *machine service*.)

100 **Dependability requirements:** By *requirements* concerning dependability we mean any such requirements which deal with either *accessibility* requirements, or *availability* requirements, or *integrity* requirements, or *reliability* requirements, or *robustness* requirements, or *safety* requirements, or *security* requirements.

101 **Describe:** To describe something is to create, in the mind of the reader, a *model* of that something. The thing, to be describable, must be either a physically manifest *phenomenon*, or a concept derived from such phenomena. Furthermore, to be describable it must be possible to create, to formulate a mathematical, i.e., a formal description of that something. (This delineation of description is narrow. It is too narrow for, for example, philosophical or literary, or historical, or psychological discourse. But it is probably too wide for a *software engineering*, or a *computing science* discourse. See also *description*.)

102 **Description:** By a description is, in our context, meant some text which designates something, i.e., for which, eventually, a mathematical *model* can be established. (We readily accept that our characterisation of the term 'description' is narrow. That is: We take as a guiding principle, as a dogma, that an informal text, a *rough sketch*, a *narrative*, is not a description unless one can eventually demonstrate a mathematical model that somehow relates to, i.e., "models" that informal text. To further paraphrase our concern about "describability", we now state that a de-

scription is a description of the *entities*, *function*s, *event*s and *behaviour*s of a further designated universe of discourse: That is, a description of a *domain*, a *prescription* of *requirements*, or a *specification* of a *software design*.)

103 **Design:** By a design we mean the *specification* of a *concrete artefact*, something that can either be physically manifested, like a chair, or conceptually demonstrated, like a software program.

104 **Designate:** To designate is to present a reference to, to point out, something. (See also *denote* and *designation*.)

105 **Designation:** The relation between a *syntactic* marker and the semantic thing signified. (See also *denote* and *designate*.)

106 **Deterministic:** In a narrow sense we shall say that a behaviour, a process, a set of actions, is deterministic if the outcome of the behaviour, etc., can be predicted: Is always the same given the same "starting conditions", i.e., the same initial *configuration* (from which the behaviour, etc., proceeds). (See also *nondeterministic*.)

107 **Developer:** The person, or the company, which constructs an *artefact*, as here, a *domain description*, or a *requirements prescription*, or a *software design*.

108 **Development:** The set of actions that are carried out in order to construct an *artefact*.

109 **Diagram:** A usually two-dimensional drawing, a figure. (Sometimes a diagram is annotated with informal and *formal* text.)

110 **Dialogue:** A "conversation" between two *agent*s (men or machines). (We thus speak of man-machine dialogues as carried out over *CHI*s (*HCI*s).)

111 **Dictionary:** See Sect. C.1.2

112 **Didactics:** Systematic instruction based on a clear conceptualisation of the bases, of the foundations, upon which what is being instructed rests. (One may speak of the didactics of a field of knowledge, such as, for example, software engineering. We believe that the present three volume book represents such a clearly conceptualised didactics, i.e., a foundationally consistent and complete basis.)

113 **Directed graph:** A directed graph is a *graph* all of whose *edge*s are directed, i.e., are *arrow*s.

114 **Directory:** A collection of directions. (We shall here take the more limited view of a directory as being a list of names of, i.e., references to *resource*s.)

115 **Discrete:** As opposed to *continuous*: consisting of distinct or unconnected elements [209].

116 **Disjunction:** Being separated, being disjoined, decomposed. (We shall mostly think of disjunction as the (meaning of the) logical connective "or": $\vee$.)

117 **Document:** By a document is meant any text, whether informal or *formal*, whether *informative*, *descriptive* (or *prescriptive*) or *analytic*. (Descriptive documents may be *rough sketch*es, *terminologies*, *narrative*s, or *formal*. Informative documents are not *descriptive*. Analytic documents

"describe" relations between documents, *verification* and *validation*, or describe properties of a document.)

118 **Documentation requirements:** By documentation requirements we mean requirements which state which kinds of documents shall make up the deliverable, what these documents shall contain and how they express what they contain.

119 **Domain:** Same as *application domain*; hence see that term for a characterisation. (The term domain is the preferred term.)

120 **Domain acquisition:** The act of acquiring, of gathering, *domain knowledge*, and of analysing and recording this knowledge.

121 **Domain analysis:** The act of analysing recorded *domain knowledge* in search of (common) properties of phenomena, or relating what may be considered separate phenomena.

122 **Domain capture:** The act of gathering *domain knowledge*, of collecting it — usually from domain *stakeholder*s.

123 **Domain description:** A textual, informal or formal document which describes the domain. (Usually a domain description is a set of documents with many parts recording many facets of the domain: The *intrinsics*, *business process*es, *support technology*, *management and organisation*, *rules and regulations*, and the *human behaviour*s.)

124 **Domain description unit:** By a domain description unit we understand a short, "one- or two-liner", possibly *rough-sketch description* of some property of a *domain phenomenon*, i.e., some property of an *entity*, some property of a *function*, of an *event*, or some property of a *behaviour*. (Usually domain description units are the smallest textual, sentential fragments elicited from domain *stakeholder*s.)

125 **Domain determination:** Domain determination is a *domain requirements facet*. It is an operation performed on a *domain description* cum *requirements prescription*. Any *nondeterminism* expressed by either of these specifications which is not desirable for some required software design must be made deterministic (by this *requirements engineer* performed operation). (Other domain requirements facets are: *domain projection*, *domain instantiation*, *domain extension* and *domain fitting*.)

126 **Domain development:** By domain development we shall understand the *development* of a *domain description*. (All aspects are included in development: *domain acquisition*, domain *analysis*, domain *model*ling, domain *validation* and domain *verification*.)

127 **Domain engineer:** A domain engineer is a *software engineer* who performs *domain engineering*. (Other forms of *software engineer*s are: *requirements engineer*s and *software design*ers (cum *programmer*s).)

128 **Domain engineering:** The engineering of the development of a *domain description*, from identification of *domain stakeholder*s, via *domain acquisition*, *domain analysis* and *domain description* to *domain validation* and *domain verification*.

129 **Domain extension:** Domain extension is a *domain requirements facet*. It is an operation performed on a *domain description* cum *requirements prescription*. It effectively extends a *domain description* by entities, functions, events and/or behaviours conceptually possible, but not necessarily humanly feasible in the domain. (Other domain requirements facets are: *domain projection*, *domain determination*, *domain instantiation* and *domain fitting*.)

130 **Domain facet:** By a domain facet we understand one amongst a finite set of generic ways of analysing a domain: A view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain. (We consider here the following domain facets: *business process*, *intrinsics*, *support technology*, *management and organisation*, *rules and regulations*, and *human behaviour*.)

131 **Domain fitting:** Domain fitting is a *domain requirements facet*. It is an operation performed on a *domain description* cum *requirements prescription*. It effectively combines one *domain description* (cum *domain requirements*) with another [*domain description*, respectively *domain requirements*]. (Other domain requirements facets are: *domain projection*, *domain determination*, *domain instantiation* and *domain extension*.)

132 **Domain initialisation:** Domain initialisation is an *interface requirements facet*. It is an operation performed on a *requirements prescription*. For an explanation see *shared data initialisation* (its 'equivalent'). (Other *interface requirements facet*s are: *shared data refreshment*, *computational data+control*, *man-machine dialogue*, *man-machine physiological* and *machine-machine dialogue requirements*.)

133 **Domain instantiation:** Domain instantiation is a *domain requirements facet*. It is an operation performed on a *domain description* (cum *requirements prescription*). Where, in a domain description certain *entities* and *function*s are left undefined, domain instantiation means that these entities or functions are now instantiated into constant *value*s. (Other requirements facets are: *domain projection*, *domain determination*, *domain extension* and *domain fitting*.)

134 **Domain knowledge:** By domain knowledge we mean that which a particular group of people, all basically engaged in the "same kind of activities", know about that domain of activity, and what they believe that other people know and believe about the same domain. (We shall, in our context, strictly limit ourselves to "knowledge", staying short of "beliefs", and we shall similarly strictly limit ourselves to assume just one "actual" world, not any number of "possible" worlds. More specifically, we shall strictly limit our treatment of domain knowledge to stay clear of the (albeit very exciting) area of reasoning about knowledge and belief between people (and agents) [109, 76].)

135 **Domain projection:** Domain projection is a *domain requirements facet*. It is an operation performed on a *domain description* cum *requirements prescription*. The operation basically "removes" from a description defini-

tions of those *entities* (including their *type definition*s), *functions*, *events* and *behaviours* that are not to be considered in the *requirements*. (The removed phenomena and concepts are thus projected "away". Other domain requirements facets are: *domain determination*, *domain instantiation*, *domain extension* and *domain fitting*.)

136 **Domain validation:** By domain validation we rather mean: '*validation* of a domain description', and by that we mean the informal assurance that a description purported to cover the *entities*, *function*s, *event*s and *behaviour*s of a further designated domain indeed does cover that domain in a reasonably representative manner. (Domain validation is, necessarily, an informal activity: It basically involves a guided reading of a domain description (being validated) by *stakeholder*s of the domain, and ends in an evaluation report written by these domain *stakeholder* readers.)

137 **Domain verification:** By domain verification we mean *verification* of claimed properties of a domain description, and by that we mean the formal assurance that a description indeed does possess those claimed properties. (The usual principles, techniques and tools of verification apply here.)

138 **Domain requirements:** By domain *requirements* we understand such requirements — save those of *business process reengineering* — which can be expressed solely by using professional terms of the *domain*. (Domain requirements constitute one requirements *facet*. Others requirements facets are: *business process reengineering*, *interface requirements* and *machine requirements*.)

139 **Domain requirements facet:** By *domain requirements* facets we understand such domain requirements that basically arise from either of the following operations on *domain description*s (cum *requirements prescription*s): *domain projection*, *domain determination*, *domain extension*, *domain instantiation* and *domain fitting*.

$\mathcal{E}$

140 **Elaborate:** See next: *elaboration*.

141 **Elaboration:** The three terms *elaboration*, *evaluation* and *interpretation* essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*, or as a function from configurations to *value*s. Given that configuration typically consists of *static environment*s and *dynamic state*s (or *storage*s), we use the term elaboration in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to pairs of states and values.

142 **Elicitation:** To elicit, to extract. (See also: *acquisition*. We consider elicitation to be part of acquisition. Acquisition is more than elicitation. Elicitation, to us, is primarily the act of extracting information, i.e., knowledge. Acquisition is that plus more: Namely the preparation of what and how to elicit and the postprocessing of that which has been elicited — in

preparation of proper analysis. Elicitation applies both to domain and to requirements elicitation.)

143 **Embedded:** Being an integral part of something else. (When something is embedded in something else, then that something else is said to surround the embedded thing.)

144 **Embedded system:** A *system* which is an integral part of a larger system. (We shall use the term embedded system primarily in the context of the larger, 'surrounding' system being *reactive* and/or *hard real time*.)

145 **Engineer:** An engineer is a person who "walks the bridge" between science and technology: (i) Constructing, i.e., designing, *technology* based on scientific insight, and (ii) analysing technology for its possible scientific content.

146 **Engineering:** Engineering is the design of *technology* based on scientific insight, and the analysis of technology for its possible scientific content. (In the context of this glossary we single out three forms of engineering: *domain engineering*, *requirements engineering* and *software design*; together we call them *software engineering*. The technology constructed by the *domain engineer* is a *domain description*. The technology constructed by the *requirements engineer* is a *requirements prescription*. The technology constructed by the *software design*er is *software*.)

147 **Enrichment:** The addition of a property to something already existing. (We shall use the term enrich in connection with a collection (i.e., a `RSL` **scheme** or a `RSL` **class**) — of definitions, declaration and axioms — being '**extend**ed **with**' further such definitions, declaration and axioms.)

148 **Entity:** By an entity we shall loosely understand something fixed, immobile, static — although that thing may move, but after it has moved it is essentially the same thing, an entity. (We shall take the narrow view of an entity, being in contrast to a *function*, and an *event*, and a *behaviour*; that entities "roughly correspond" to what we shall think of as *value*s, i.e., as *information* or *data*. We shall further allow entities to be either *atomic* or *composite*, i.e., in the latter case having decomposable subentities (cf. *subentity*). Finally entities may have nondecomposable *attribute*s.)

149 **Enumerable:** By enumerable we mean that a set of elements satisfies a *proposition*, i.e., can be logically characterised.

150 **Enumeration:** To list, one after another. (We shall use the term enumeration in connection with the syntactic expression of a "small", i.e., definite, number of elements of a(n enumerated) *set*, *list* or *map*.)

151 **Environment:** A context, that is, in our case (i.e., usage), the ("more static") part of a *configuration* in which some syntactic entity is *elaborated*, *evaluated*, or *interpreted*. (In our "metacontext", i.e., that of software engineering, environments, when deployed in the elaboration (etc.) of, typically, specifications or programs, record, i.e., list, associate, identifiers of the specification or program text with their meaning.)

152 **Epistemology:** The study of knowledge. (Contrast, please, to *ontology*.)

153 **Error:** An error is an action that produces an incorrect result. An error is that part of a *machine state* which is "liable to lead to subsequent failure". An error affecting the *machine service* is an indication that a *failure* occurs or has occurred [182]. (An error is caused by a *fault*.)
154 **Evaluate:** See next: *evaluation*.
155 **Evaluation:** The three terms *elaboration*, *evaluation* and *interpretation* essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*, or as a function from configurations to *value*s. Given that configuration typically consists of *static environment*s and *dynamic state*s (or *storage*s), we use the term evaluation in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to values.
156 **Event:** Something that occurs instantaneously. (We shall, in our context, take events as being manifested by certain *state* changes, and by certain *interaction*s between *behaviour*s or *process*es. The occurrence of events may "trigger" actions. How the triggering, i.e., the *invocation* of *functions* are brought about is usually left implied, or unspecified.)
157 **Expression:** An expression, in our context (i.e., that of software engineering), is a syntactical entity which, through *evaluation*, designates a *value*.
158 **Extension:** We shall here take extension to be the same as *enrichment*. (The extension of a *concept* is all the individuals falling under the concept [166].)
159 **Extensional:** Concerned with objective reality [209]. (Please observe a shift here: We do not understand the term extensional as 'relating to, or marked by extension in the above sense, but in contrast to *intensional*.)

$\mathcal{F}$

160 **Facet:** By a facet we understand one amongst a finite set of generic ways of analysing and presenting a *domain*, a *requirements* or a *software design*: a view of the universe of discourse, such that the different facets cover conceptually different views, and such that these views together cover that universe of discourse. (Examples of domain facets are *intrinsics*, *business process*es, *support technology*, *management and organisation*, *rules and regulations* and *human behaviour*. Examples of requirements facets are *business process reengineering*, *domain requirements*, *interface requirements* and *machine requirements*. Examples of software design facets are *software architecture*, *component design*, *module design*, etc.)
161 **Failure:** A *fault* may result in a failure. A *machine* failure occurs when the delivered *machine service* deviates from fulfilling the machine function, the latter being what the machine is aimed at [182]. (A failure is thus something relative to a *specification*, and is due to a *fault*. Failures are concerned with such things as *accessibility*, *availability*, *reliability*, *safety* and *security*.)

162 **Fault:** The adjudged (i.e., the 'so judged') or hypothesised cause of an *error* [182]. (An *error* is caused by a fault, i.e., faults cause errors. A software fault is the consequence of a human *error* in the development of that software.)

163 **Fault tree:** A fault tree is a tree with nodes of alternating kinds: event and logic nodes. The fault tree root is an event node and so are all the leaf nodes. Event nodes label (undesirable) events (or states of a computing system). Logic nodes designate combinators like conjunction, disjunction, etc. (See the definitions of branch, event, fault, node, root, state and tree [items 88, 270, 276, 464, 614, 679, 750, Appendix B, Vol. 1].)

164 **Fault tree analysis:** A form of safety analysis that assesses computing systems safety to provide failure statistics and sensitivity analyses that indicate the possible effect of critical failures. (In the technique known as fault tree analysis, an undesired effect is taken as the root ("top event") of a tree of logic. Then, each situation that could cause that effect is added to the tree as a series of logic expressions. When fault trees are labelled with actual numbers about failure probabilities, which are often in practice unavailable because of the expense of testing, computer programs can calculate failure probabilities from fault trees. See the definition of hazard analysis.)

165 **Finite:** Of a fixed number less than infinity, or of a fixed structure that does not "flow" into perpetuity as would any *information structure* that just goes on and on. (Watch out for the four terms: *finite*, *infinite*, *definite* and *indefinite*.)

166 **Flowchart:** A diagram (a chart), for example of circles (input, output), annotated (square) boxes, annotated diamonds and infixed arrows, that shows step by step flow through an algorithm.

167 **Formal:** By formal we shall, in our context (i.e., that of software engineering), mean a language, a system, an argument (a way of reasoning), a program or a specification whose syntax and semantics is based on (rules of) mathematics (including mathematical logic).

168 **Formal definition:** Same as *formal description*, *formal prescription* or *formal specification*.

169 **Formal development:** Same as the standard meaning of the composition of *formal* and *development*. (We usually speak of a spectrum of development modes: *systematic development*, *rigorous development*, and formal development. Formal software development, to us, is at the "formalistic" extreme of the three modes of development: Complete *formal specification*s are always constructed, for all (phases and) stages of development; all *proof obligation*s are expressed; and all are discharged (i.e., proved to hold).)

170 **Formal description:** A *formal description* of something. (Usually we use the term formal description only in connection with *formalisation* of *domain*s.)

171 **Formalisation:** The act of making a formal specification of something elsewhere informally specified; or the document which results therefrom.

172 **Formal method:** By a formal method we mean a *method* whose techniques and tools[1] are *formal*ly based. (It is common to hear that some notation is claimed to be that of a formal method — where it then turns out that few, if any, of the building blocks of that notation have any formal foundation. This is especially true of many diagrammatic notations. `UML` is a case in point — much is presently being done to formalise subsets of `UML` [168].)

173 **Formal prescription:** Same as *formal definition* or *formal specification*. (Usually we use the term formal prescription only in connection with *formalisation* of *requirements*.)

174 **Formal specification:** A *formalisation* of something. (Same as *formal definition*, *formal description* or *formal prescription*. Usually we use the term formal specification only in connection with *formalisation* of *software design*s.)

175 **Function:** By a function we understand something which when *applied* to a *value*, called an *argument*, yields a value called a *result*. (Functions can be modelled as sets of (argument, result) pair — in which case applying a function to an argument amounts to "searching" for an appropriate pair. If several such pairs have the same argument (value), the function is said to be *nondeterministic*. If a function is applied to an argument for which there is no appropriate pair, then the function is said to be partial; otherwise it is a total function.)

176 **Function activation:** When, in an operational, i.e., computational ("mechanical") sense, a function is being applied, then some resources have to be set aside in order to carry out, to handle, the application. This is what we shall call a function activation. (Typically a function activation, for conventional *block-structured* languages (like `C#, Java, Standard ML` [106, 197, 97]), is implemented by means (also) of a stack-like data structure: Function invocation then implies the stacking (pushing) of a stack activation on that stack, i.e., the *activation stack* (a circular reference!). Elaboration of the function definition body means that intermediate values are pushed and popped from the topmost activation element, etc., and that completion of the function application means that the top stack activation is popped.)

177 **Functional:** A function whose arguments are allowed themselves to be functions is called a functional. (The *fix point* (finding) function is a functional.)

[1] Tools include specification and programming languages as such, as well as all the software tools relating to these languages (editors, syntax checkers, theorem provers, proof assistants, model checkers, specification and program (flow) analysers, interpreters, compilers, etc.).

178 **Functional programming:** By functional programming we mean the same as *applicative programming*: In its barest rendition functional programming involves just three things: definition of functions, functions as ordinary *value*s, and *function application* (i.e., *function invocation*). (Most current functional programming languages (`Haskell, Miranda, Standard ML`) go well beyond just providing the three basic building blocks of functional programming [216, 217, 160].)

179 **Functional programming language:** By a functional programming language we mean a *programming language* whose principal values are functions and whose principal operations on these values are their creation (i.e., definition), their application (i.e., invocation) and their composition. (Functional programming languages of interest today, 2005, are (alphabetically listed): `CAML` [57, 55, 56, 224, 138], `Haskell` [216], `Miranda` [217], `Scheme` [1, 92, 75] and `SML` (Standard ML) [160, 97]. `LISP 1.5` was a first functional programming language [154].)

180 **Function application:** The act of applying a function to an argument is called a function application. (See 'comment' field of *function activation* just above.)

181 **Function definition:** A *function definition*, as does any definition, consists of a *definiens* and a *definiendum*. The definiens is a *function signature*, and the definiendum is a clause, typically an expression. (Cf. *Lambda-function*s.)

182 **Function invocation:** Same as *function application*. (See parenthesized remark of entry 176 (*function activation*).)

183 **Function signature:** By a function signature we mean a text which presents the name of the function, the types of its argument values and the type(s) of its result value(s).

$\mathcal{G}$

184 **Generator function:** To speak of a generator function we need first introduce the concept of a *sort* "of interest". A generator function is a function which when applied to arguments of some kind, i.e., types, yields a value of the type of the sort "of interest". (Typically the sort "of interest" can be thought of as the state (a stack, a queue, etc.).)

185 **Glossary:** See Sect. C.1.1.

186 **Grand state:** "Grand state" is a colloquial term. It is meant to have the same meaning as *configuration*. (The colloquialism is used in the context of, for example, praising a software engineer as "being one who really knows how to design the grand state for some universe of discourse" being specified.)

187 **Grouping:** By grouping we mean the ordered, finite collection, into a *Cartesian*, of mathematical structures (i.e., *value*s).

$\mathcal{H}$

188 **Hardware:** By hardware is meant the physical embodiment of a computer: its electronics, its boards, the racks, cables, button, lamps, etc.

189 **HCI:** Abbreviation for human computer interface. (Same as *CHI*, and same as *man-machine* interface.)

190 **Human behaviour:** By human behaviour we shall here understand the way a human follows the enterprise *rules and regulations* as well as interacts with a *machine*: dutifully honouring specified (machine *dialogue*) *protocol*s, or negligently so, or sloppily not quite so, or even criminally not so! (Human behaviour is a *facet* of the *domain* (of the enterprise). We shall thus model human behaviour also in terms of it failing to react properly, i.e., humans as *nondeterministic agent*s! Other facets of an enterprise are those of its *intrinsics*, *business process*es, *support technology*, *management and organisation*, and *rules and regulations*.)

$\mathcal{I}$

191 **Identification:** The pointing out of a relation, an association, between an *identifier* and that "thing", that *phenomenon*, it *designate*s, i.e., it stands for or identifies.

192 **Identifier:** A name. (Usually represented by a string of alphanumeric characters, sometimes with properly infixed "-"s or "_"s.)

193 **Imperative:** Expressive of a command [209]. (We take imperative to more specifically be a reflection of *do this, then do that*. That is, of the use of a *state*-based programming approach, i.e., of the use of an *imperative programming language*. See also *indicative*, *optative*, and *putative*.)

194 **Imperative programming:** Programming, *imperative*ly, "with" references to *storage location*s and the updates of those, i.e., of *state*s. (Imperative programming seems to be the classical, first way of programming digital computers.)

195 **Imperative programming language:** A programming language which, significantly, offers language constructs for the creation and manipulation of variables, i.e., *storage*s and their *location*s. (Typical imperative programming languages were, in "ye olde days", `Fortran, Cobol, Algol 60, PL/I, Pascal, C,` etc. [153, 151, 17, 152, 17, 125]. Today programming languages like `C++, Java, C#,` etc. [213, 197, 106] additionally offer *module* cum *object* "features".)

196 **Implementation:** By an implementation we understand a computer program that is made suitable for *compilation* or *interpretation* by a *machine*. (See next entry: *implementation relation*.)

197 **Implementation relation:** By an *implementation* relation we understand a logical relation of *correctness* between a *software design specification* and an *implementation* (i.e., a computer program made suitable for *compilation* or *interpretation* by a *machine*).

198 **Incomplete:** We say that a *proof system* is incomplete if not all true sentences are provable.

199 **Incompleteness:** Noun form of the *incomplete* adjective.

200 **Inconsistent:** A set of *axiom*s is said to be inconsistent if, by means of these, and some *deduction rule*s, one can *prove* a property and its negation.

201 **Indefinite:** Not definite, i.e., of a fixed number or a specific property, but it is not known, at the point of uttering the term 'indefinite', what that number or property is. (Watch out for the four terms: *finite*, *infinite*, *definite* and *indefinite*.)

202 **Indicative:** Stating an objective fact. (See also *imperative*, *optative* and *putative*.)

203 **Inert:** A *dynamic phenomenon* is said to be inert if it cannot change *value* of its own volition, i.e., by itself, but only through the *interaction* between that *phenomenon* and a change-instigating *environment*. An inert phenomenon only changes value as the result of external stimuli. These stimuli prescribe exactly which new value they are to change to. (Contrast to *active* and *reactive*.)

204 **Infinite:** As you would think of it: not finite! (Watch out for the four terms: *finite*, *infinite*, *definite* and *indefinite*.)

205 **Informal:** Not formal! (We normally, by an informal specification mean one which may be precise (i.e., unambiguous, and even concise), but which, for example is expressed in natural, yet (domain specific) professional language — i.e., a language which does not have a precise semantics let alone a formal *proof system*. The `UML` notation is an example of an informal language [168].)

206 **Informatics:** The confluence of (i) *application*s, (ii) *computer science*, (iii) *computing science* [i.e., the art [127, 128, 129] (1968–1973), craft [190] (1981), discipline [71] (1976), logic [104] (1984), practice [105] (1993–2004), and science [90] (1981) of programming], (iv) *software engineering* and (v) *mathematics*.

207 **Information:** The communication or reception of knowledge. (By information we thus mean something which, in contrast to *data*, informs us. No computer representation is, let alone any efficiency criteria are, assumed. Data as such does, i.e., bit patterns do, not 'inform' us.)

208 **Information structure:** By an information structure we shall normally understand a composition of more "formally" represented (i.e., structured) *information*, for example, in the "believed" form of *table*, a *tree*, a *graph*, etc. (In contrast to *data structure*, an information structure does not necessarily have a computer representation, let alone an "efficient" such.)

209 **Informative documentation:** By informative documentation we understand texts which *inform,* but which do not (essentially) describe that which a *development* is to develop. (Informative documentation is balanced by *descriptive* and *analytic documentation* to make up the full documentation of a *development*.)

210 **Infrastructure:** According to the World Bank: *'Infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share tech-*

nical and economic features (such as economies of scale and spillovers from users to nonusers). We shall use the term as follows: Infrastructures are concerned with supporting other systems or activities. Computing systems for infrastructures are thus likely to be distributed and concerned in particular with supporting communication of information, control, people and materials. Issues of (for example) openness, timeliness, security, lack of corruption, and resilience are often important. (Winston Churchill is quoted to have said, during a debate in the House of Commons, in 1946: *... The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms as 'infra-structures'.*)

211 **Input:** By input we mean the *communication* of *information* (*data*) from an outside, an *environment*, to a *phenomenon* "within" our universe of discourse. (More colloquially, and more generally: Input can be thought of as *value*(s) transferred over *channel*(s) to, or between *process*es. Cf. *output*. In a narrow sense we talk of input to an *automaton* (i.e., a *finite state automaton* or a *pushdown automaton*) and a *machine* (here in the sense of, for example, a *finite state machine* (or a *pushdown machine*)).)

212 **Instance:** An individual, a thing, an *entity*. (We shall usually think of an 'instance' as a *value*.)

213 **Instantiation:** 'To represent (an abstraction) by a concrete *instance*' [209]. (We shall sometimes be using the term 'instantiation' in lieu of a *function invocation* on an *activation stack*.)

214 **Installation manual:** A *document* which describes how a *computing system* is to be installed. (A special case of 'installation' is the downloading of *software* onto a *computing system*. See also *training manual* and *user manual*.)

215 **Intangible:** Not *tangible*.

216 **Integrity:** By a *machine* having integrity we mean that that machine remains unimpaired, i.e., has no faults, errors and failures, and remains so even in the situations where the environment of the machine has faults, errors and failures. (Integrity is a *dependability requirement*.)

217 **Intension:** Intension indicates the internal content of a term. (See also *in intension*. The intension of a *concept* is the collection of the properties possessed jointly by all conceivable individuals falling under the concept [166]. The intension determines the *extension* [166].)

218 **Intensional:** Adjective form of *intension*.

219 **Interact:** The term interact here addresses the phenomenon of one *behaviour* acting in unison, simultaneously, *concurrent*ly, with another behaviour, including one behaviour influencing another behaviour. (See also *interaction*.)

220 **Interaction:** Two-way reciprocal action.

221 **Interface:** Boundary between two disjoint sets of communicating phenomena or concepts. (We shall think of the systems as *behaviour*s or *pro-*

*cess*es, the boundary as being *channel*s, and the communications as *input*s and *output*s.)

222 **Interface requirements:** By interface requirements we understand the expression of expectations as to which software-software, or software-hardware *interface* places (i.e., *channel*s), *input*s and *output*s (including the *semiotics* of these input/outputs) there shall be in some contemplated *computing system*. (Interface requirements can often, usefully, be classified in terms of *shared data initialisation requirements*, *shared data refreshment requirements*, *computational data+control requirements*, *man-machine dialogue requirements*, *man-machine physiological requirements* and *machine-machine dialogue requirements*. Interface requirements constitute one requirements *facet*. Other requirements facets are: *business process reengineering*, *domain requirements* and *machine requirements*.)

223 **Interface requirements facet:** See *interface requirements* for a list of facets: *shared data initialisation*, *shared data refreshment*, *computational data+control*, *man-machine dialogue*, *man-machine physiological* and *machine-machine dialogue requirements*.

224 **Interpret:** See next: *interpretation*.

225 **Interpretation:** The three terms *elaboration*, *evaluation* and *interpretation* essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*, or as a function from configurations to *value*s. Given that configuration typically consists of *static environment*s and *dynamic state*s (or *storage*s), we use the term interpretation in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to states.

226 **Interpreter:** An interpreter is an *agent*, a *machine*, which performs *interpretation*s.

227 **Intrinsics:** By the intrinsics of a *domain* we shall understand those phenomena and concepts of a domain which are basic to any of the other facets, with such a domain intrinsics initially covering at least one specific, hence named, *stakeholder* view. (Intrinsics is thus one of several *domain facet*s. Others include: *business process*es, *support technology*, *management and organisation*, *rules and regulations*, and *human behaviour*.)

228 **Invariant:** By an invariant we mean a property that holds of a *phenomenon* or a *concept*, both before and after any *action* involving that phenomenon or a concept. (A case in point is usually an *information* or a *data structure*: Assume an action, say a repeated one (e.g., a while loop). We say that the action (i.e., the while loop) preserves an invariant, i.e., usually a *proposition*, if the proposition holds true of the *state* before and the state after any *interpretation* of the while loop. Invariance is here seen separate from the *well-formedness* of an *information* or a *data structure*. We refer to the explication of *well-formedness*!)

$\mathcal{K}$

229 **Keyword:** A significant word from a title or document. (See *KWIC*.)
230 **Knowledge:** What is, or what can be known. The body of truth, information, and principles acquired by mankind [209]. (See *epistemology* and *ontology*. *A priori knowledge:* Knowledge that is independent of all particular experiences. *A posteriori knowledge:* Knowledge, which derives from experience alone.)

$\mathcal{L}$

231 **Label:** Same as named *program point*.
232 **Language:** By a language we shall understand a possibly infinite set of *sentence*s which follow some *syntax*, express some *semantics* and are uttered, or written down, due to some *pragmatics*.
233 **Law:** A law is a rule of conduct prescribed as binding or enforced by a controlling authority. (We shall take the term law in the specific sense of law of Nature (cf., Ampére's Law, Boyle's Law, the conservation laws (of mass-energy, electric charge, linear and angular momentum), Newton's Laws, Ohm's Law, etc.), and laws of Mathematics (cf. "law of the excluded middle" (as in logic: a proposition must either be true, or false, not both, and not none)).)
234 **Lemma:** An auxiliary *proposition* used in the demonstration of another proposition. (Instead of proposition we could use the term *theorem*.)
235 **Link:** A link is the same as a *pointer*, an *address* or a *reference*: something which refers to, i.e., designates something (typically something else).
236 **Linguistics:** The study and knowledge of the *syntax*, *semantics* and *pragmatics* of *language*(s).
237 **List:** A list is an ordered sequence of zero, one or more not necessarily distinct entities.
238 **Literal:** A term whose use in software engineering, i.e., programming, shall mean: an identifier which denotes a constant, or is a keyword. (Usually that identifier is emphasised. Examples of `RSL` literals are: **Bool, true, false, chaos, if, then, else, end, let, in,** and the numerals $0, 1, 2., ..., 1234.5678$, etc.)
239 **Live Sequence Chart:** The Live Sequence Chart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [63, 102, 126].)
240 **Location:** By a location is meant an area of *storage*.
241 **Logic:** The principles and criteria of validity of inference and deduction, that is, the mathematics of the formal principles of reasoning. (We refer to Vol. 1, Chap. 9 for our survey treatment of mathematical logic.)
242 **Logic programming:** Logic programming is programming based on an interpreter which either performs deductions or inductions, or both. (In logic programming the chief values are those of the Booleans, and the chief forms of expressions are those of propositions and predicates.)

243 **Logic programming language:** By a *logic programming* language is meant a language which allows one to express, to prescribe, *logic programming*. (The classical logic programming language is `Prolog` [142, 112].)

244 **Loose specification:** By a loose specification is understood a specification which either *underspecifies* a problem, or specifies this problem *nondeterministically*.

$\mathcal{M}$

245 **Machine:** By the machine we understand the *hardware* plus *software* that implements some *requirements*, i.e., a *computing system*. (This definition follows that of M.A. Jackson [123].)

246 **Machine requirements:** By *machine requirements* we understand *requirements* put specifically to, i.e., expected specifically from, the *machine*. (We normally analyse machine requirements into *performance requirements*, *dependability requirements*, *maintenance requirements*, *platform requirements* and *documentation requirements*.)

247 **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 *interact*s with it [182].

248 **Maintenance:** By maintenance we shall here, for software, mean change to *software*, i.e., its various *document*s, due to needs for (i) adapting that software to new *platform*s, (ii) correcting that software due to observed software errors, (iii) improving certain performance properties of the *machine* of which the software is part, or (iv) avoiding potential problems with that machine. (We refer to subcategories of maintenance: *adaptive maintenance*, *corrective maintenance*, *perfective maintenance* and *preventive maintenance*.)

249 **Maintenance requirements:** By *maintenance requirements* we understand requirements which express expectations on how the *machine* being desired (i.e., required) is expected to be maintained. (We also refer to *adaptive maintenance*, *corrective maintenance*, *perfective maintenance* and *preventive maintenance*.)

250 **Management and organisation:** By management and organisation we mean those *facet*s of a *domain* which are representative of relations between the various management levels of an enterprise, and between these and non-management staff, i.e., "blue-collar" workers. (As such, management and organisation is about formulating strategical, tactical and operational goals for the enterprise, of communicating and "translating" these goals into action to be done by management and staff, in general, and to "backstop" when "things do not 'work out'", i.e., handling complaints from "above" and "below". Other facets of an enterprise are those of its *intrinsics*, *business process*es, *support technology*, *rules and regulations* and *human behaviour*.)

251 **Man-machine dialogue:** By man-machinedialogues we understand actual instantiations of *user* interactions with *machine*s, and machine interactions with users: what input the users provide, what output the machine initiates, the interdependencies of these inputs/outputs, their temporal and spatial constraints, including response times, input/output media (locations), etc. (

252 **Man-machine dialogue requirements:** By man-machine dialogue requirements we understand those *interface requirements* which express expectations on, i.e., mandates the *protocol* according to which *user*s are to interact with the *machine*, and the machine with the users. (See *man-machine dialogue*. For other *interface requirements* see *computational data+control requirements*, *shared data initialisation requirements*, *shared data refreshment requirements*, *man-machine physiological requirements* and *machine-machine dialogue requirements*.))

253 **Man-machine physiological requirements:** By man-machine physiological requirements we understand those *interface requirements* which express expectations on, i.e., mandates, the form and appearance of ways in which the *man-machine dialogue* utilises such physiological devices as visual display screens, keyboards, "mouses" (and other tactile instruments), audio microphones and loudspeakers, television cameras, etc. (See also *computational data+control requirements*, *shared data initialisation requirements*, *shared data refreshment requirements*, *man-machine dialogue requirements* and *machine-machine dialogue requirements*.)

254 **Map:** A map is like a *function*, but is here thought of as an *enumerable* set of pairs of argument/result values. (Thus the *definition set* of a map is usually decidable, i.e., whether an entity is a member of a definition set of a map or not can usually be decided.)

255 **Mereology:** The theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole. (Mereology is often considered a branch of *ontology*. Leading investigators of mereology were Franz Brentano, Edmund Husserl, Stanislaw Lesniewski [199, 145, 159, 206, 207, 214] and Leonard and Goodman [137].)

256 **Meta-IV:** `Meta-IV` stands for the fourth metalanguage (for programing language definition conceived at the IBM Vienna Laboratory in the 1960s and 1970s). (`Meta-IV` is pronounced meta-four.)

257 **Metalanguage:** By a metalanguage is understood a *language* which is used to explain another language, either its *syntax*, or its *semantics*, or its *pragmatics*, or two or all of these! (One cannot explain any language using itself. That would lead to any interpretation of what is explained being a valid solution, in other words: Nonsense. `RSL` thus cannot be used to explain `RSL`. Typically formal specification languages are metalanguages: being used to explain, for example, the semantics of ordinary programming languages.)

258 **Metalinguistic:** We say that a language is used in a metalinguistic manner when it is being deployed to explain some other language. (And we

also say that when we examine a language, like we could, for example, examine RSL, and when we use a subset of RSL to make that analysis, then that subset of RSL is used metalinguistically (wrt. all of RSL).)

259 **Metaphysics:** We quote from: http://mally.stanford.edu/: "Whereas physics is the attempt to discover the laws that govern fundamental concrete objects, metaphysics is the attempt to discover the laws that systematize the fundamental abstract objects presupposed by physical science, such as natural numbers, real numbers, functions, sets and properties, physically possible objects and events, to name just a few. The goal of metaphysics, therefore, is to develop a formal ontology, i.e., a formally precise systematization of these abstract objects. Such a theory will be compatible with the world view of natural science if the abstract objects postulated by the theory are conceived as patterns of the natural world." (Metaphysics may, to other scientists and philosophers, mean more or other, but for software engineering the characterisation just given suffices.)

260 **Method:** By a method we shall here understand a set of *principle*s for selecting and using a number of *technique*s and *tool*s in order to construct some *artefact*. (This is our leading definition — one that sets out our methodological quest: to identify, enumerate and explain the principles, the techniques and, in cases, the tools — notably where the latter are specification and programming languages. (Yes, languages are tools.))

261 **Methodology:** By methodology we understand the study and knowledge of *method*s, one, but usually two or more. (In some dialects of English, methodology is confused with method.)

262 **Model:** A model is the mathematical meaning of a description (of a domain), or a prescription (of requirements), or a specification (of software), i.e., is the meaning of a specification of some universe of discourse. (The meaning can be understood either as a mathematical function, as for a *denotational semantics* meaning, or an *algebra* as for an *algebraic semantics* or a *denotational semantics* meaning, etc. The essence is that the model is some mathematical structure.)

263 **Model-oriented:** A specification (description, prescription) is said to be model-oriented if the specification (etc.) *denote*s a *model*. (Contrast to *property-oriented*.)

264 **Model-oriented type:** A type is said to be model-oriented if its specification *designate*s a *model*. (Contrast to *property-oriented type*.)

265 **Modularisation:** The act of structuring a text using *module*s.

266 **Module:** By a module we shall understand a clearly delineated text which denotes either a single complex quantity, as does, usually, an *object*, or a possibly empty, possibly infinite set of *model*s of objects. (The RSL module concept is manifested in the use of one or more of the RSL *class* (**class ... end**), *object* (**object** identifier **class ... end**, etc.), and *scheme* (**scheme** identifier **class ... end**), etc., constructs. We refer to [62, 61, 25] and to [172, 171] for original, early papers on modules.)

267 **Module design:** By module design we shall understand the *design* of (one or more) *module*s.

268 **Monotonic:** A function, $f : A \to B$, is monotonic, if for all a, a' in the definition set A of f, and some ordering relations, $\sqsubseteq$, on a and B, we have that if $a \sqsubseteq a'$ then $f(a) \sqsubseteq f(a')$.

269 **Multi-dimensional:** A composite (i.e., a non*atomic*) *entity* is a multi-dimensional *entity* if some relations between properly contained (i.e., constituent) subentities (cf. *subentity*) can only be described by both forward and backward references, and/or with recursive references. (This is in contrast to *one-dimensional* entities.)

270 **Multimedia:** The use of various forms of input/output media in the man-machine interface: Text, two-dimensional graphics, voice (audio), video, and tactile instruments (like "mouse").

$\mathcal{N}$

271 **Name:** A name is syntactically (generally an expression, but usually it is) a simple alphanumeric identifier. Semantically a name denotes (i.e., designates) "something". Pragmatically a name is used to uniquely identify that "something". (Shakespeare: Romeo: "What's in a name?" Juliet to Romeo: "That which we call a rose by any other name would smell as sweet.")

272 **Naming:** The action of allocating a unique name to a value.

273 **Narrative:** By a narrative we shall understand a document text which, in precise, unambiguous language, introduces and describes (prescribes, specifies) all relevant properties of entities, functions, events and behaviours, of a set of phenomena and concepts, in such a way that two or more readers will basically obtain the same idea as to what is being described (prescribed, specified). (More commonly: Something that is narrated, a story.)

274 **Natural language:** By a natural language we shall understand a language like Arabic, Chinese, English, French, Russian, Spanish, etc. — one that is spoken today, 2005, by people, has a body of literature, etc. (In contrast to natural languages we have (i) professional languages, like the languages of medical doctors, or lawyers, or skilled craftsmen like carpenters, etc.; and we have (ii) formal languages like software specification languages, programming languages, and the languages of first-order predicate logics, etc.)

275 **Network:** By a network we shall understand the same as a directed, but not necessarily *acyclic graph*. (Our only use of it here is in connection with network *databases*.)

276 **Node:** A point in some *graph* or *tree*.

277 **Nondeterminate:** Same as *nondeterministic*.

278 **Nondeterministic:** A property of a specification: May, on purpose, i.e., deliberately have more than one meaning. (A specification which is am-

biguous also has more than one meaning, but its ambiguity is of overriding concern: It is not 'nondeterministic' (and certainly not 'deterministic'!).)

279 **Nondeterminism:** A *nondeterministic* specification models nondeterminism.

280 **Notation:** By a notation we shall usually understand a reasonably precisely delineated language. (Some notations are textual, as are programming notations or specification languages; some are diagrammatic, as are, for example, *Petri net*s, *statechart*s, *live sequence chart*s, etc.)

281 **Noun:** Something, a name, that refers to an *entity*, a quality, a *state*, an *action*, or a *concept*. Something that may serve as the subject of a *verb*. (But beware: In English many nouns can be "verbed", and many verbs can be "nouned"!)

$\mathcal{O}$

282 **Object:** An instance of the *data structure* and *behaviour* defined by the object's *class*. Each object has its own *value*s for the instance *variable*s of its class and can respond to the *function*s defined by its class. (Various *specification language*s, `object Z` [54, 72, 73], `RSL`, etc., each have their own, further refined, meaning for the term 'object', and so do *object-oriented programming language* (viz., `C++` [213], `Java` [15, 89, 139, 223, 10, 197], `C#` [174, 158, 157, 106] and so on).)

283 **Object-oriented:** We say that a program is *object-oriented* if its main structure is determined by a *modularisation* into a *class*, that is, a cluster of *type*s, *variable*s and *procedure*s, each such set acting as a separate *abstract data type*. Similarly we say that a *programming language* is object-oriented if it specifically offers language constructs to express the appropriate *modularisation*. (Object-orientedness became a mantra of the 1990s: Everything had to be object-oriented. And many programming problems are indeed well served by being structured around some object-oriented notion. The first *object-oriented programming language* was `Simula 67` [25].)

284 **Observer:** By an observer we mean basically the same as an *observer function*.

285 **Observer function:** An observer function is a *function* which when "applied" to an *entity* (a *phenomenon* or a *concept*) yields subentities or attributes of that entity (without "destroying" that entity). (Thus we do not make a distinction between functions that observe subentities (cf. *subentity*) and functions that observe *attribute*s. You may wish to make distinctions between the two kinds of observer function. You can do so by some simple *naming* convention: assign names the prefix `obs_` when you mean to observe subentities, and `attr_` when you mean to observe attributes. Vol. 3 Chap. 5 introduces these concepts.)

286 **Ontology:** In philosophy: A systematic account of Existence. To us: An explicit formal specification of how to represent the phenomena, concepts and other entities that are assumed to exist in some area of interest (some

universe of discourse) and the relationships that hold among them. (Further clarification: An ontology is a catalogue of *concept*s and their relationships — including properties as relationships to other concepts. See Sect. C.1.4.)

287 **Operation:** By an operation we shall mean a *function*, or an *action* (i.e., the effect of function *invocation*). (The context determines which of these two strongly related meanings are being referred to.)

288 **Operational:** We say that a *specification* (a *description*, a *prescription*), say of a *function*, is operational if what it explains is explained in terms of how that thing, **how** that phenomenon, or concept, operates (rather than by **what** it achieves). (Usually operational definitions are *model oriented* (in contrast to *property oriented*).)

289 **Operational abstraction:** Although a definition (a *specification*, a *description*, or a *prescription*) may be said, or claimed, to be *operational*, it may still provide *abstraction* in that the *model-oriented* concepts of the definition are not themselves directly representable or performable by humans or computers. (This is in contrast to *denotational abstraction*s or *algebra*ic (or *axiom*atic) *abstraction*s.)

290 **Operational semantics:** A *definition* of a *language semantics* that is *operational*. (See also *structural operational semantics*.)

291 **Operation transformation:** To speak of *operation reification* one must first be able to refer to an abstract, usually *property-oriented*, specification of the operation. Then, by operation *transformation* we mean a *specification* which is, somehow, *calculate*d from the abstract specification. (Three nice books on such calculi are: [161, 24, 16].)

292 **Optative:** Expressive of wish or desire. (See also *imperative*, *indicative*, and *putative*.)

293 **Organisation:** By organisation we shall here, in a narrow sense, only mean the administrative or functional structure of an enterprise, a public or private administration, or of a set of services, as for example in a consumer/retailer/wholesaler/producer/distributor market, or in a financial services industry, etc.

294 **Organisation and management:** The composite term organisation and management applies in connection with *organisation*s as outlined just above. The term then emphasises the relations between the organisation and its management. (For more, see *management and organisation*.)

295 **Output:** By output we mean the *communication* of *information* (*data*) to an outside, an *environment*, from a *phenomenon* "within" our universe of discourse. (More colloquially, and more generally: output can be thought of as *value*(s) transferred over *channel*(s) from, or between, *process*es. Cf. *input*. In a narrow sense we talk of output from a *machine* (e.g., a *finite state machine* or a *pushdown machine*).)

296 **Overloaded:** The concept of 'overloaded' is a concept related to *function symbol*s, i.e., *function name*s. A function name is said to be overloaded if there exists two or more distinct *signature*s for that function name.

(Typically overloaded function symbols are '+', which applies, possibly, in some notation, to addition of integers, addition of reals, etc., and '=', which applies, possibly, in some notation, to comparison of any pair of *value*s of the same *type*.)

$\mathcal{P}$

297 **Paradigm:** A philosophical and theoretical framework of a scientific school or discipline within which theories, laws and generalizations and the experiments performed in support of them are formulated; a philosophical or theoretical framework of any kind. (Software engineering is full of paradigms: Object-orientedness is one.)

298 **Parallel programming language:** A *programming language* whose major kinds of concepts are *process*es, process *composition* [putting processes in parallel and *nondeterministic* {internal or external} choice of process *elaboration*], and synchronisation and communication between processes. (A main example of a practical parallel programming language is `occam` [116], and of a specificational 'programming' language is `CSP` [110, 194, 196]. Most recent *imperative programming language*s (Java, C#, etc.) provide for programming constructs (e.g., threads) that somehow mimic parallel programming.)

299 **Perfective maintenance:** By perfective maintenance we mean an update, as here, of software, to achieve a more desirable use of resources: time, storage space, equipment. (We also refer to *adaptive maintenance*, *corrective maintenance* and *preventive maintenance*.)

300 **Performance:** By performance we, here, in the context of computing, mean quantitative figures for the use of computing resources: time, storage space, equipment.

301 **Performance requirements:** By performance requirements we mean *requirements* which express *performance* properties (desiderata).

302 **Phase:** By a phase we shall here, in the context of software development, understand either the *domain development* phase, the *requirements development* phase, or the *software design* phase.

303 **Phenomenon:** By a phenomenon we shall mean a physically manifest "thing". (Something that can be sensed by humans (seen, heard, touched, smelled or tasted), or can be measured by physical apparatus: Electricity (voltage, current, etc.), mechanics (length, time and hence velocity, acceleration, etc.), chemistry, etc.)

304 **Phenomenology:** Phenomenology is the study of structures of consciousness as experienced from the first-person point of view [227].

305 **Platform:** By a platform, we shall, in the context of computing, understand a *machine*: Some computer (i.e., hardware) equipment and some software systems. (Typical examples of platforms are: `Microsoft Windows` running on an `IBM ThinkPad Series T` model, or `Trusted Solaris` op-

erating system with an Oracle Database $10g$ running on a Sun Fire E25K Server.)

306 **Platform requirements:** By platform requirements we mean *requirements* which express *platform* properties (desiderata). (There can be several platform requirements: One set for the platform on which software shall be developed. Another set for the platform(s) on which software shall be utilised. A third set for the platform on which software shall be demonstrated. And a fourth set for the platform on which software shall be maintained. These platforms need not always be the same.)

307 **Portability:** Portability is a concept associated with *software*, more specifically with the *program*s (or *data*). Software is (or files, including *data base* records, are) said to be portable if it (they), with ease, can be "ported" to, i.e., made to "run" on, a new *platform* and/or compile with a different compiler, respectively different database management system.

308 **Post-condition:** The concept of post-condition is associated with function application. The post-condition of a function f is a predicate p_{o_f} which expresses the relation between argument a and result r values that the function f defines. If a represent argument values, r corresponding result values and f the function, then $f(a) = r$ can be expressed by the post-condition predicate p_{o_f}, namely, for all applicable a and r the predicate p_{o_f} expresses the truth of $p_{o_f}(a, r)$. (See also *pre-condition*.)

309 **Postfix:** The concept of postfix is basically a syntactic one, and is associated with operator/operand expressions. It is one about the displayed position of a unary (i.e., a monadic) operator with respect to its operand (expression). An expression is said to be in postfix form if a monadic operator is shown, is displayed, after the expression to which it applies. (Typically the factorial operator, say !, is shown after its operand expression, viz. 7!.)

310 **Pragmatics:** Pragmatics is the (i) study and (ii) practice of the factors that govern our choice of language in social interaction and the effects of our choice on others. (We use the term pragmatics in connection with the use of language, as complemented by the *semantics* and *syntax* of language.)

311 **Pre-condition:** The concept of pre-condition is associated with function application where the function being applied is a partial function. That is: for some arguments of its definition set the function yields **chaos**, that is, does not terminate. The pre-consition of the function is then a predicate which expresses those values of the arguments for which the function application terminates, that is, yields a result value. (See *weakest pre-condition*.)

312 **Predicate:** A predicate is a truth-valued expression involving terms over arbitrary values, well-formed formula relating terms and with *Boolean connective*s and *quantifier*s.

313 **Predicate logic:** A predicate logic is a language of *predicate*s (given by some *formal syntax*) and a *proof system*.

314 **Presentation:** By presentation we mean the syntactic *document*ation of the results of some *development*.

315 **Prescription:** A prescription is a specification which prescribes something designatable, i.e., which states what shall be achieved. (Usually the term 'prescription' is used only in connection with *requirements* prescriptions.)

316 **Preventive maintenance:** By preventive maintenance — of a *machine* — we mean that a set of special tests are performed on that *machine* in order to ascertain whether the *machine* needs *adaptive maintenance*, and/or *corrective maintenance*, and/or *perfective maintenance*. (If so, then an update, as here, of software, has to be made in order to achieve suitable *integrity* or *robustness* of the *machine*.)

317 **Principle:** An accepted or professed rule of action or conduct, ..., a fundamental doctrine, right rules of conduct, ... [211]. (The concept of principle, as we bring it forth, relates strongly to that of *method*. The concept of principle is "fluid". Usually, by a method, some people understand an orderliness. Our definition puts the orderliness as part of overall principles. Also, one usually expects analysis and construction to be efficient and to result in efficient artifacts. Also this we relegate to be implied by some principles, techniques and tools.)

318 **Procedure:** By a procedure we mean the same as a *function*. (Same as *routine* or *subroutine*.)

319 **Process:** By a process we understand a sequence of actions and events. The events designate interaction with some environment of the process.

320 **Program:** A program, in some *programming language*, is a formal text which can be subject to *interpretation* by a computer. (Sometimes we use the term *code* instead of program, namely when the program is expressed in the machine language of a computer.)

321 **Programmable:** An *active dynamic phenomenon* has the programmable (active dynamic) attribute if its *action*s (hence *state* changes) over a future time interval can be accurately prescribed. (Cf. *autonomous* and *biddable*.)

322 **Programmer:** A person who does *software design*.

323 **Program organisation:** By program organisation we loosely mean how a *program* (i.e., its text) is structured into, for example, *module*s (eg., *class*es), *procedure*s, etc.

324 **Programming:** The act of constructing *program*s. From [79]:

> *1: The art of debugging a blank sheet of paper (or, in these days of on-line editing, the art of debugging an empty file). 2: A pastime similar to banging one's head against a wall, but with fewer opportunities for reward. 3: The most fun you can have with your clothes on (although clothes are not mandatory).*

325 **Programming language:** A language for expressing *program*s, i.e., a language with a precise *syntax*, a *semantics* and some textbooks which provides remnants of the *pragmatics* that was originally intended for that programming language. (See next entry: *programming language type*.)

326 **Programming language type:** With a *programming language* one can associate a *type*. Typically the name of that type intends to reveal the type of a main paradigm, or a main data type of the language. (Examples are: *functional programming language* (major data type is functions, major operations are definition of functions, application of functions and composition of functions), *logic programming language* (major kinds of expressions are ground terms in a Boolean algebra, propositions and predicates), *imperative programming language* (major kinds of language constructs are declaration of assignable variables, and assignment to variables, and a more or less indispensable kind of data type is references [locations, addresses, pointers]), and *parallel programming language*.)

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

328 **Proof:** A *proof* of a theorem, ϕ, from a set, Γ, of sentences of some *formal proposition*al or *predicate* language, $\mathcal{L}$, is a finite sequence of sentences, ϕ_1, ϕ_2, ..., ϕ_n, where $\phi = \phi_1$, where $\phi_n = \textbf{true}$, and in which each ϕ_i is either an *axiom* of $\mathcal{L}$, or a member of Γ, or follows from earlier ϕ_j's by an *inference rule* of $\mathcal{L}$.

329 **Proof obligation:** A clause of a program may only be (dynamically) well-defined if the values of clause parts lie in certain ranges (viz. no division by zero). We say that such clauses raise proof obligations, i.e., an obligation to prove a property. (Classically it may not be statically (i.e., compile time) checkable that certain expression values lie within certain *subtype*s. Discharging a proof may help ensure such constraints.)

330 **Proof rule:** Same as *inference rule* or *axiom*.

331 **Proof system:** A *consistent* and (relative) *complete* set of *proof rule*s.

332 **Property:** A quality belonging and especially peculiar to an individual or thing; an *attribute* common to all members of a class. (Hence: "Not a property owned by someone, but a property possessed by something".)

333 **Property-oriented:** A specification (description, prescription) is said to be property-oriented if the specification (etc.) expresses *attribute*s. (Contrast to *model oriented*.)

334 **Proposition:** An expression in language which has a truth value.

335 **Pure functional programming language:** A *functional programming language* is said to be pure if none of its constructs designates *side-effects*.

336 **Putative:** Commonly accepted or supposed, that is, assumed to exist or to have existed. (See also *imperative*, *indicative* and *optative*.)

$\mathcal{Q}$

337 **Quality:** Specific and essential character. (Quality is an *attribute*, a *property*, a characteristic (something has character).)

338 **Quantification:** The operation of quantifying. (See *quantifier*. The x (the y) is quantifying expression $\forall x{:}X{\cdot}P(x)$ (respectively $\exists y{:}Y{\cdot}Q(y)$).)

339 **Quantifier:** A marker that quantifies. It is a prefixed operator that binds the variables in a logical formula by specifying their possible range of *value*s. (Colloquially we speak of the **universal** and the **existential** quantifiers, $\forall$, respectively $\exists$. Typically a quantified expression is then of either of the forms $\forall x{:}X{\cdot}P(x)$ and $\exists y{:}Y{\cdot}Q(y)$. They 'read': For all quantities x of type X it is the case that the predicate $P(x)$ holds; respectively: There exists a quantity y of type Y such that the predicate $Q(y)$ holds.)

340 **Quantity:** An indefinite *value*. (See the *quantifier* entry: The quantities in $P(x)$ (respectively $Q(y)$) are of type X (respectively Y). y is indefinite in that it is one of the quantities of Y, but which one is not said.)

$\mathcal{R}$

341 **RAISE:** `RAISE` stands for Rigorous Approach to Industrial Software Engineering. (`RAISE` refers to a method, `The RAISE Method` [87], a specification language, `RSL` [85], and "comes" with a set of tools.)

342 **Range:** The concept of range is here used in connection with functions. Same as *range set*. See next entry.

343 **Range set:** Given a *function*, its range set is that set of *value*s which is yielded when the function is *applied* to each member of its *definition set*.

344 **Reactive:** A *phenomenon* is said to be reactive if the phenomenon performs *action*s in response to external stimuli. Thus three properties must be satisfied for a system to be of reactive dynamic attribute: (i) An interface must be definable in terms of (ii) provision of input stimuli and (iii) observation of (state) reaction. (Contrast to *inert* and *active*.)

345 **Reactive system:** A *system* whose main phenomena are chiefly *reactive*. (See the *reactive* entry just above.)

346 **Real time:** We say that a *phenomenon* is real time if its behaviour somehow must guarantee a response to an external event within a given time. (Cf. *hard real time* and *soft real time*.)

347 **Reasoning:** Reasoning is the ability to *infer*, i.e., to make *deduction*s or *induction*s. (Automated reasoning is concerned with the building and use of computing systems that automate this process. The overall goal is to mechanise different forms of reasoning.)

348 **Reengineering:** By reengineering we shall, in a narrow sense, only consider the reengineering of business processes. Thus, to us, reengineering is the same as *business process reengineering*. (Reengineering is also used in the wider sense of a major change to some already existing engineering *artefact*.)

349 **Reference:** A reference is the same as an *address*, a *link*, or a *pointer*: something which refers to, i.e., designates something (typically something else).

350 **Refinement:** Refinement is a *relation* between two *specification*s: One specification, D, is said to be a refinement of another specification, S, if all the properties that can be observed of S can be observed in D. Usually this is expressed as $D \sqsubseteq S$. (Set-theoretically it works the other way around: in $D \supseteq S$, D allows behaviours not accounted for in S.)

351 **Refutable assertion:** A refutable assertion is an assertion that might be refuted (i.e., convincingly shown to be false). (Einstein's theory of relativity, in a sense, refuted Newton's laws of mechanics. Both theories amount to assertions.)

352 **Refutation:** A refutation is a statement that (convincingly) refutes an assertion. (Lakatos [130] drew a distinction between refutation (evidence that counts against a theory) and rejection (deciding that the original theory has to be replaced by another theory). We can still use Newton's theory provided we stay within certain boundaries, within which that theory is much easier to handle than Einstein's theory.)

353 **Reification:** The result of a *reify* action. (See also *data reification*, *operation reification* and *refinement*.)

354 **Reify:** To regard (something *abstract*) as a material or *concrete* thing. (Our use of the term is more *operational*: To take an *abstract* thing and turn it into a less abstract, more *concrete* thing.)

355 **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*. (Cf. *dependability* [being dependable].) (Reliability is a *dependability requirement*. Usually reliability is considered a *machine* property. As such, reliability is (to be) expressed in a *machine requirements* document.)

356 **Renaming:** By renaming we mean *Alpha-renaming*. (Renaming, in this sense, is a concept of the *Lambda-calculus*.)

357 **Representation abstraction:** By *representation abstraction* of [typed] values we mean a specification which does not hint at a particular data (structure) model, that is, which is not implementation biased. (Usually a representation abstraction (of data) is either *property oriented* or is *model oriented*. In the latter case it is then expressed, typically, in terms of mathematical entities such as sets, Cartesians, lists, maps and functions.)

358 **Requirements:** A condition or capability needed by a user to solve a problem or achieve an objective [114].

359 **Requirements acquisition:** The gathering and enunciation of *requirements*. (Requirements acquisition comprises the activities of preparation, requirements *elicitation* (i.e. *requirements capture*) and preliminary requirements evaluation (i.e., requirements vetting).)

360 **Requirements analysis:** By *requirements analysis* we understand a reading of requirements acquisition (rough) prescription units, (i) with the aim of forming concepts from these requirements prescription units, (ii) as well as with the aim of discovering inconsistencies, conflicts and incompletenesses within these requirements prescription units, and (iii) with the aim of evaluating whether a requirements can be objectively shown to hold, and if so what kinds of tests (etc.) ought be devised.

361 **Requirements capture:** By requirements capture we mean the act of eliciting, of obtaining, of extracting, requirements from *stakeholder*s. (For practical purposes requirements capture is synonymous with *requirements elicitation*.)

362 **Requirements definition:** Proper *definition*al part of a *requirements prescription*.

363 **Requirements development:** By requirements development we shall understand the *development* of a *requirements prescription*. (All aspects are included in development: *requirements acquisition*, requirements *analysis*, requirements *model*ling, requirements *validation* and requirements *verification*.)

364 **Requirements elicitation:** By requirements elicitation we mean the actual extraction of *requirements* from *stakeholder*s.

365 **Requirements engineer:** A requirements engineer is a *software engineer* who performs *requirements engineering*. (Other forms of *software engineer*s are *domain engineer*s and *software design*ers (cum *programmer*).)

366 **Requirements engineering:** The engineering of the development of a *requirements prescription*, from identification of *requirements stakeholder*s, via *requirements acquisition*, *requirements analysis*, and *requirements prescription* to requirements *validation* and requirements *verification*.

367 **Requirements facet:** A requirements facet is a view of the requirements — "seen from a *domain description*" — such as *domain projection*, *domain determination*, *domain instantiation*, *domain extension*, *domain fitting* or *domain initialisation*.

368 **Requirements prescription:** By a *requirements prescription* we mean just that: the prescription of some requirements. (Sometimes, by requirements prescription, we mean a relatively complete and consistent specification of all requirements, and sometimes just a *requirements prescription unit*.)

369 **Requirements prescription unit:** By a *requirements prescription* unit we understand a short, "one or two liner", possibly *rough sketch*, *prescription* of some property of a *domain requirements*, an *interface requirements*, or a *machine requirements*. (Usually requirements prescription units are

the smallest textual, sentential fragments elicited from requirements *stakeholder*s.)

370 **Requirements specification:** Same as *requirements prescription* — the preferred term.

371 **Requirements validation:** By requirements validation we rather mean the *validation* of a *requirements prescription*.

372 **Retrieval:** Used here in two senses: The general (typically *database*-oriented) sense of 'the retrieval [the fetching] of data (of obtaining information) from a repository of such'. And the special sense of 'the retrieval of an abstraction from a concretisation', i.e., abstracting a concept from a phenomenon (or another, more operational concept). (See the next entry for the latter meaning.)

373 **Retrieve function:** By a *retrieve function* we shall understand a function that applies to *values* of some *type*, the "more concrete, operational" type, and yields *values* of some *type* claimed to be more *abstract*. (Same as *abstraction function*.)

374 **Rigorous:** Favoring rigor, i.e., being precise.

375 **Rigorous development:** Same as the composed meaning of the two terms *rigorous* and *development*. (We usually speak of a spectrum of development modes: *systematic development*, rigorous development and *formal development*. Rigorous software development, to us, "falls" somewhere between the two other modes of development: (Always) complete *formal specification*s are constructed, for all (phases and) stages of development; some, but usually not all *proof obligation*s are expressed; and usually only a few are discharged (i.e., proved to hold).)

376 **Risk:** The Concise Oxford Dictionary [140] defines risk (noun) in terms of a hazard, chance, bad consequences, loss, etc., exposure to mischance. Other characterisations of the term risk are: someone or something that creates or suggests a hazard, and possibility of loss or injury.

377 **Robustness:** A *system* is robust — in the context of a *machine* being *dependable* — if it retains all its *dependability* attributes (i.e., properties) after *failure* and after *maintenance*. (Robustness is (thus) a *dependability requirement*.)

378 **Root:** A root is a *node* of a *tree* which is not a sub*tree* of a larger, *embedding* (*embedded*) tree.

379 **Rough sketch:** By a rough sketch — in the context of *descriptive software development documentation* — we shall understand a *document* text which describes something which is not yet consistent and complete, and/or which may still be too concrete, and/or overlapping, and/or repetitive in its descriptions, and/or with which the describer has yet to be fully satisfied.

380 **Route:** Same as *path*.

381 **Routine:** Same as *procedure*.

382 **RSL:** `RSL` stands for the `RAISE` [87] Specification Language [85]. ()

383 **Rule:** A regulating principle. (We use the concept of rules in several different contexts: *rewrite rule*, *rule of grammar* and *rules and regulations*.)

$\mathcal{S}$

384 **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. (Safety is a *dependability requirement*. Usually safety is considered a *machine* property. As such safety is (to be) expressed in a *machine requirements document*.)

385 **Safety critical:** A *system* whose *failure* may cause injury or death to human beings, or serious loss of property, or serious disruption of services or production, is said to be safety critical.

386 **Script:** By a domain script we shall understand the structured, almost, if not outright, formally expressed, wording of a rule or a regulation (cf. *rules and regulations*) that has legally binding power, that is, which may be contested in a court of law.

387 **Secure:** To properly define the concept of secure, we first assume the concept of an authorised user. Now, a *system* is said to be secure if an un-authorised user, when supposedly making use of that system, (i) is not able to find out what the system does, (ii) is not able to find out how it does 'whatever' it does do, and (iii), after some such "use", does not know whether he/she knows! (The above characterisation represents an unattainable proposition. As a characterisation it is acceptable. But it does not hint at ways and means of implementing secure systems. Once such a system is believed implemented the characterisation can, however be used as a guide in devising tests that may reveal to which extent the system indeed is secure. Secure systems usually deploy some forms of authorisation and encryption mechanisms in guarding access to system functions.)

388 **Security:** When we say that a *system* exhibits security we mean that it is *secure*. (Security is a *dependability requirement*. Usually security is considered a *machine* property. As such security is (to be) expressed in a *machine requirements* document.)

389 **Selector:** By a selector (a selector function) we understand a function which is applicable to *values* of a certain, defined, composed *type*, and which yields a proper component of that value. The function itself is defined by the *type definition*.

390 **Semantics:** Semantics is the study and knowledge [incl. specification] of meaning in language [58]. (We make the distinction between the *pragmatics*, the semantics and the *syntax* of languages. Leading textbooks on semantics of programming languages are [67, 93, 191, 195, 215, 225].)

391 **Semantic function:** A semantics function is a function which when applied to *syntactic values* yields their *semantic values*.

392 **Semantic type:** By a semantic type we mean a *type* that defines *semantic values*.

393 **Semiotics:** Semiotics, as used by us, is the study and knowledge of *pragmatics*, *semantics* and *syntax* of language(s).

394 **Sensor:** A sensor can be thought of as a piece of *technology* (an electronic, a mechanical or an electromechanical device) that senses, i.e., measures, a physical *value*. (A sensor is in contrast to an *actuator*.)

395 **Sentence:** (i) A word, clause, or phrase or a group of clauses or phrases forming a syntactic unit which expresses an assertion, a question, a command, a wish, an exclamation, or the performance of an action, that in writing usually begins with a capital letter and concludes with appropriate end punctuation, and that in speaking is distinguished by characteristic patterns of stress, pitch and pauses; (ii) a mathematical or logical statement (as an equation or a proposition) in words or symbols [209].

396 **Sequential:** Arranged in a sequence, following a linear order, one after another.

397 **Sequential process:** A process is sequential if all its observable actions can be, or are, ordered in sequence.

398 **Set:** We understand a set as a mathematical entity, something that is not mathematically defined, but is a concept that is taken for granted. (Thus by a set we understand the same as a collection, an aggregation, of distinct entities. Membership (of an entity) of a set is also a mathematical concept which is likewise taken for granted, i.e., undefined.)

399 **Set theoretic:** We say that something is set theoretically understood or explained if its understanding or explanation is based on *sets*.

400 **Shared data:** See *shared phenomenon*.

401 **Shared data initialisation:** By shared data initialisation we understand an *operation* that (initially) creates a *data structure* that reflects, i.e., models, some *shared phenomenon* in the *machine*. (See also *shared data refreshment*.)

402 **Shared data initialisation requirements:** *Requirements* for *shared data initialisation*. (See also *computational data+control requirements*, *shared data refreshment requirements*, *man-machine dialogue requirements*, *man-machine physiological requirements*, and *machine-machine dialogue requirements*.)

403 **Shared data refreshment:** By shared data refreshment we understand a *machine operation* which, at prescribed intervals, or in response to prescribed events updates an (originally initialised) *shared data* structure. (See also *shared data initialisation*.)

404 **Shared data refreshment requirements:** *Requirements* for *shared data refreshment*. (See also *computational data+control requirements*, *shared data initialisation requirements*, *man-machine dialogue requirements*, *man-machine physiological requirements*, and *machine-machine dialogue requirements*.)

405 **Shared information:** See *shared phenomenon*.

406 **Shared phenomenon:** A shared phenomenon is a phenomenon which is present in some *domain* (say in the form of facts, *knowledge* or *information*) and which is also represented in the *machine* (say in the form of *data*). (See also *shared data* and *shared information*.)

407 **Side effect:** A language construct that designates the modification of the state of a system is said to be a side-effect-producing construct. (Typical side effect constructs are assignment, input and output. A *programming language* "without side effects" is said to be a *pure functional programming language*.)

408 **Sign:** Same as *symbol*.

409 **Signature:** See *function signature*.

410 **Soft real time:** By soft real time we mean a *real time* property where the exact, i.e., absolute timing, or time interval, is only of loose, approximate essence. (Cf., *hard real time*.)

411 **Software:** By software we understand not only the code that when "submitted" to a computer enables desired computations to take place, but also all the documentation that went into its development (i.e., its *domain description*, *requirements specification*, its complete *software design* (all stages and steps of *refinement* and *transformation*), the *installation manual*, *training manual*, and the *user manual*).

412 **Software component:** Same as *component*.

413 **Software architecture:** By a software architecture we mean a first kind of specification of software — after requirements — one which indicates **how** the software is to handle the given requirements in terms of *software components* and their interconnection — though without detailing (i.e., designing) these software components.

414 **Software design:** By software design we shall understand the determination of which *components*, which *modules* and which *algorithms* shall implement the *requirements* — together with all the *documents* that usually make up properly documented *software*. (Software design entails *programming*, but programming is a "narrower" field of activity than software design in that programming usually excludes many documentation aspects.)

415 **Software design specification:** The *specification* of a *software design*.

416 **Software development:** To us, software development includes all three phases of *software development*: *domain development*, *requirements development* and *software design*.

417 **Software development project:** A *software* development project is a planning, research and development project whose aim is to construct *software*.

418 **Software engineer:** A software engineer is an *engineer* who performs one or more of the functions of *software engineering*. (These functions include *domain engineering*, *requirements engineering* and *software design* (incl. *programming*).)

419 **Software engineering:** The confluence of the science, logic, discipline, craft and art of *domain engineering*, *requirements engineering* and *software design*.

420 **Sort:** A sort is a collection, a structure, of, at present, further unspecified entities. (That is, same as an *algebraic type*. When we say "at present, further unspecified", we mean that the (values of the) sort may be subject to constraining axioms. When we say "a structure", we mean that "this set" is not necessarily a *set* in the simple sense of mathematics, but may be a collection whose members satisfy certain interrelations, for example, some *partially ordered set*, some *neighbourhood set* or other.)

421 **Sort definition:** The *definition* of a *sort*. (Usually a sort definition consists of the (introduction of) a type name, some (typically *observer function* and *generator function*) *signatures*, and some *axioms* relating sort *values* and *functions*.)

422 **Source program:** By a source program we mean a *program* (text) in some *programming language*. (The term source is used in contrast to target: the result of compiling a source text for some target *machine*.)

423 **Span:** Span is here used, in contrast to *scope*, more specifically in the context of the degree to which a project *scope* and *span* extend: Scope being the "larger, wider" delineation of what a project "is all about", *span* being the "narrower", more precise extent.

424 **Specification:** We use the term 'specification" to cover the concepts of *domain description*s, *requirements prescription*s and *software design*s. More specifically a specification is a *definition*, usually consisting of many definitions.

425 **Specification language:** By a specification language we understand a *formal language* capable of expressing *formal specifications*. (We refer to such formal specification languages as: `ASM` [188], `B & eventB` [4, 5, 52], `CASL` [23, 164, 163], `CafeOBJ` [68, 69], `RSL` [85, 86], `VDM-SL` [44, 78] and `Z` [203, 205, 226, 108].)

426 **Stage:** (i) By a development stage we shall understand a set of development activities which either starts from nothing and results in a complete phase documentation, or which starts from a complete phase documentation of stage kind, and results in a complete phase documentation of another stage kind. (ii) By a development stage we shall understand a set of development activities such that some (one or more) activities have created new, externally conceivable (i.e., observable) properties of what is being described, whereas some (zero, one or more) other activities have refined previous properties. (Typical development stages are: *domain intrinsics*, *domain support technologies*, *domain management and organisation*, *domain rules and regulations*, etc., and *domain requirements*, *interface requirements*, and *machine requirements*, etc.)

427 **Stakeholder:** By a *domain* (*requirements*, *software design*)[2] stakeholder we shall understand a person, or a group of persons, "united" somehow in their common interest in, or dependency on the domain (requirements, software design); or an institution, an enterprise, or a group of such, (again) characterised (and, again, loosely) by their common interest in, or dependency on the domain (requirements, software design). (The three stakeholder groups usually overlap.)

428 **Stakeholder perspective:** By a *stakeholder* perspective we shall understand the, or an, understanding of the *universe of discourse* shared by the specifically identified stakeholder group — a view that may differ from one stakeholder group to another stakeholder group of the same universe of discourse.

429 **State:** By a state we shall, in the context of computer *programs*, understand a summary of past *computations*, and, in the context of *domains*, a suitably selected set of *dynamic entities*.

430 **Statechart:** The Statechart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [98, 99, 101, 103, 100].)

431 **Statement:** We shall take the rather narrow view that a statement is a *programming language* construct which *denotes* a *state*-to-state function. (Pure expressions are then programming language constructs which denote state-to-value functions (i.e., with no *side effect*), whereas "impure" expressions, also called clauses, denote state-to-state-and-value functions.)

432 **Step:** By a development step we shall understand a refinement of a domain description (or a requirements prescription, or a software design specification) module, from a more abstract to a more concrete description (or a more concrete requirements prescription, or a more concrete software design specification).

433 **Stepwise development:** By a stepwise development we shall understand a *development* that undergoes *phases*, *stages* or *steps* of development, i.e., can be characterised by pairs of two adjoining *phase steps*, a last *phase step* and a (first) next *phase step*, or two adjoining *stage steps*.

434 **Stepwise refinement:** By a stepwise refinement we understand a pair of adjoining *development steps* where the transition from one *step* to the next *step* is characterised by a *refinement*. (Refinement is thus always stepwise refinement.)

435 **Structure:** The term 'structure' is understood rather loosely. Normally we shall understand a structure as a mathematical structure, such as an *algebra*, or a *predicate logic*, or a *Lambda-calculus*, or some defined abstraction (a *scheme* or a *class*). (Set theory is a (mathematical) structure. So are `RSL`'s Cartesian, list and map data types.)

[2] These three areas of concern form three *universes of discourse*.

436 **Subentity:** A subentity is a proper part of a (thus) non-*atomic entity*. (Do not confuse a subentity of an entity with an *attribute* of that entity (or of that subentity).)

437 **Subtype:** To speak of a subtype we must first be able to speak of a *type*, i.e., colloquially, a (suitably structured) set of *value*s. A subtype of a type is then a (suitably structured) and proper subset of the values of the type. (Usually we shall, in RSL, think of a predicate, p, that applies to all members of the type, T, and singles out a proper subset whose elements satisfy the predicate: $\{a \mid a : T \cdot p(a)\}$.)

438 **Support technology:** By a support technology we understand a *facet* of a *domain*, one which reflects its (current) dependency on mechanical, electro-mechanical, electronic and other technologies (i.e., tools) in order to carry out its *business process*es. (Other facets of an enterprise are those of its *intrinsics*, *business process*es, *management and organisation*, *rules and regulations* and *human behaviour*.)

439 **Synopsis:** By a synopsis we shall understand a composition of *informative documentation* and *rough-sketch description* of some project.

440 **Syntax:** By syntax we mean (i) the ways in which words are arranged to show meaning (cf. *semantics*) within and between sentences, and (ii) rules for forming *syntactically correct* sentences. (See also *regular syntax*, *context-free syntax*, *context-sensitive syntax* and *BNF* for specifics.)

441 **System:** A regularly interacting or interdependent group of phenomena or concepts forming a whole, that is, a group of devices or artificial objects or an organization forming a network especially for producing something or serving a common purpose. (This book will have its own characterisation of the concept of a system (commensurate, however, with the above encircling characterisation); cf. Vol. 2, Sect. 9.5's treatment of system.)

442 **Systematic development:** Systematic development of software is *formal development "lite"!* (We usually speak of a spectrum of development modes: systematic development, *rigorous development*, and *formal development*. Systems software development, to us, is at the "informal" extreme of the three modes of development: *formal specification*s are constructed, but maybe not for all stages of development; and usually no proof obligations are expressed, let alone proved. The three volumes of this series of textbooks in software engineering can thus be said to expound primarily the systematic approach.)

443 **Systems engineering:** By systems engineering we shall here understand computing systems engineering: The confluence of developing *hardware* and *software* solutions to *requirements*.

$\mathcal{T}$

444 **Taxonomy:** See Sect. C.1.5.

445 **Technique:** A procedure, an approach, to accomplish something.

446 **Technology:** We shall in these volumes be using the term technology to stand for the results of applying scientific and engineering insight. This, we think, is more in line with current usage of the term IT, information technology.

447 **Temporal:** Of or relating to time, including sequence of time, or to time intervals (i.e., durations).

448 **Temporal logic:** A(ny) *logic* over *temporal phenomena*. (We refer to Vol. 2, Chap. 15 for our survey treatment of some temporal logics.)

449 **Term:** From [140]: A word or phrase used in a definite or precise sense in some particular subject, as a science or art; a technical expression. More widely: any word or group of words expressing a notion or conception, or denoting an object of thought. (Thus, in RSL, a term is a *clause*, an *expression*, a *statement*, which has a *value* (statements have the **Unit** value).)

450 **Terminal:** By a terminal we shall mean a terminal *symbol* which (in contrast to a *nonterminal* symbol) designates something specific.

451 **Termination:** The concept of termination is associated with that of an *algorithm*. We say that an algorithm, when subject to *interpretation* (colloquially: 'execution'), may, or may not terminate. That is, may halt, or may "go on forever, forever looping". (Whether an algorithm terminates is *undecidable*.)

452 **Terminology:** By terminology is meant ([140]): The doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature.

453 **Test:** A test is a means to conduct *testing*. (Typically such a test is a set of data values provided to a program (or a specification) as values for its *free variables*. *Testing* then evaluates the program (resp., interprets (symbolically) the specification) to obtain a result (value) which is then compared with what is (believed to be) the, or a, correct result. See Vol. 3, Sects. 14.3.2, 22.3.2 and 29.5.3 for treatments of the concept of test.)

454 **Testing:** Testing is a systematic effort to refute a claim of correctness of one (e.g., a concrete) specification (for example a program) with respect to another (the abstract) specification. (See Vol. 3, Sects. 14.3.2, 22.3.2, and 29.5.3 for treatments of the concept of testing.)

455 **Theorem:** A theorem is a *sentence* that is *provable* without assumptions, that is "purely" from *axioms* and *inference rules*.

456 **Theorem prover:** A mechanical, i.e., a computerised means for *theorem proving*. (Well-known theorem provers are: PVS [169, 170] and HOL/Isabelle [167].)

457 **Theorem proving:** The act of *proving theorems*.

458 **Theory:** A formal theory is a *formal language*, a set of *axioms* and *inference rules* for *sentences* in this language, and is a set of *theorems* proved about sentences of this language using the axioms and inference rules. A mathematical theory leaves out the strict formality (i.e., the *proof* system)

requirements and relies on mathematical proofs that have stood the social test of having been scrutinised by mathematicians.

459 **Thesaurus:** See Sect. C.1.7.

460 **Time:** Time is often a notion that is taken for granted. But one may do well, or better, in trying to understand time as some point set that satisfies certain axioms. Time and space are also often related (via [other] physically manifest "things"). Again their interrelationship needs to be made precise. (In comparative concurrency semantics one usually distinguishes between linear time and branching time semantic equivalences [220]. We refer to our treatment of time and space in Vol. 2 Chap. 5, to Johan van Benthem's book *The Logic of Time* [218], and to Wayne D. Blizard's paper *A Formal Theory of Objects, Space and Time* [48].)

461 **Token:** Something given or shown as an identity. (When, in RSL, we define a *sort* with no "constraining" axioms, we basically mean to define a set of tokens.)

462 **Tool:** An instrument or apparatus used in performing an operation. (The tools most relevant to us, in software engineering, are the *specification* and *programming language*s as well as the *software* packages that aid us in the development of (other) software.)

463 **Training manual:** A *document* which can serve as a basis for a (possibly self-study) course in how to use a *computing system*. (See also *installation manual* and *user manual*.)

464 **Transaction:** General: A communicative action or activity involving two *agent*s that reciprocally influence each other. (Special: The term transaction has come to be used, in computing, notably in connection with the use of database management systems (DBMS, or similar multiuser systems): A transaction is then a unit of interaction with a DBMS (etc.). To further qualify as being a transaction, it must be handled, by the DBMS (etc.), in a coherent and reliable way independent of other transactions.)

465 **Transformation:** The operation of changing one configuration or expression into another in accordance with a precise rule. (We consider the results of *substitution*, of *translation* and of *rewriting* to be transformations of what the *substitution*, the *translation* and the *rewriting* was applied to.)

466 **Transition:** Passage from one state, stage, subject or place to another; a movement, development, or evolution from one form, stage or style to another [209].

467 **Transition rule:** A *rule*, of such a form that it can specify how any of a well-defined class of *states* of a *machine* may make *transitions* to another state, possibly *nondeterministically* to any one of a well-defined number of other states. (The seminal 1981 report *A Structural Approach to Operational Semantics*, by Gordon D. Plotkin [176], set a de facto standard for formulating transition rules (exploring their theoretical properties and uses).)

468 **Translate:** See *translation*.

469 **Translation:** An act, process or instance of translating, i.e., of rendering from one language into another.
470 **Translator:** Same as a *compiler*.
471 **Triptych:** An ancient Roman writing tablet with three waxed leaves hinged together; a picture (as an altarpiece) or carving in three panels side by side [209]. (The trilogy of the *phases* of *software development*, *domain engineering*, *requirements engineering* and *software design* as promulgated by this trilogy of volumes!)
472 **Tuple:** A grouping of values. (Like 2-tuplets, quintuplets, etc. Used extensively, at least in the early days, in the field of relational databases — where a tuple was like a row in a relation (i.e., table).)
473 **Type:** Generally a certain kind of set of *value*s. (See *algebraic type*, *model-oriented type*, *programming language type* and *sort*.)
474 **Type check:** The concept of type check arises from the concepts of *function signatures* and function *arguments*. If arguments are not of the appropriate type then a type check yields an *error* result. (By appropriate *static typing* of *declarations* of *variables* of a *programming language* or a *specification language* one can perform static type checking (i.e., at *compile time*).)
475 **Type constructor:** A type constructor is an operation that applies to *types* and yields a *type*. (The type constructors of RSL include the power set constructors: **-set** and **-infset**, the Cartesian constructor: $\times$, the list constructors: * and $^\omega$, the map constructor: $\overrightarrow{m}$, the total and partial function space constructors: $\rightarrow$ and $\xrightarrow{\sim}$, the union type constructor: $|$, and others.)
476 **Type definition:** A type definition semantically associates a *type name* with a *type*. Syntactically, as, for example, in RSL, a type definition is either a *sort* definition or is a *definition* whose right-hand side is a *type expression*.
477 **Type expression:** A type expression semantically denotes a *type*. Syntactically, as, for example, in RSL, a type expression is an expression involving *type names* and *type constructors*, and, rarely, *terminals*.
478 **Type name:** A type name is usually just a simple *identifier*.
479 **Typing:** By typing we mean the association of *types* with *variables*. (Usually such an association is afforded by pairing a *variable identifier* with a *type name* in the variable *declaration*. See also *dynamic typing* and *static typing*.)

$\mathcal{U}$

480 **UML:** Universal Modelling Language. A hodgepodge of notations for expressing requirements and designs of computing systems. (Vol. 2, Chaps. 10, and 12–14 outlines our attempt to "UML"-ize formal techniques.)
481 **Underspecify:** By an underspecified expression, typically an identifier, we mean one which for repeated occurrences in a specification text always

yields the same value, but what the specific value is, is not knowable. (Cf. *nondeterministic* or *loose specification*.)

482 **Undecidable:** A formal logic system is undecidable if there is no *algorithm* which prescribes *computation*s that can determine whether any given sentence in the system is a theorem.

483 **Universe of discourse:** That which is being talked about; that which is being discussed; that which is the subject of our concern. (The four most prevalent universes of discourse of this book, this series of volumes on software engineering, are: *software development methodology*, *domains*, *requirements* and *software design*.)

484 **Update:** By an update we shall understand a change of value of a variable, including also the parts, or all, of a *database*.

485 **Update problem:** By the update problem we shall understand that data stored in a *database* usually reflect some state of a domain, but that changes in the external state of that domain are not always properly, including timely, reflected in the database.

486 **User:** By a user we shall understand a person who uses a *computing system*, or a *machine* (i.e., another computing system) which *interfaces* with the former. (Not to be confused with *client* or *stakeholder*.)

487 **User-friendly:** A "lofty" term that is often used in the following context: *"A computing system, a machine, a software package, is required to be user-friendly"* — without the requestor further prescribing the meaning of that term. Our definition of the term user-friendly is as follows: A *machine* (software + hardware) is said to be user-friendly (i) if the *shared phenomena* of the application *domain* (and *machine*) are each implemented in a transparent, one-to-one manner, and such that no IT jargon, but common application *domain terminology* is used in their (i.1) accessing, (i.2) *invocation* (by a human *user*), and (i.3) display (by the machine); i.e., (ii) if the *interface requirements* have all been carefully expressed (commensurate, in further detailed ways: ..., with the user psyche) and correctly implemented; and (iii) if the machine otherwise satisfies a number of *performance* and *dependability requirements* that are commensurate, in further detailed ways: ..., with the user psyche.

488 **User manual:** A *document* which a regular user of a *computing system* refers to when in doubt concerning the use of some features of that system. (See also *installation manual* and *training manual*.)

$\mathcal{V}$

489 **Valid:** A *predicate* is said to be *valid* if it is true for all *interpretation*s. (In this context think of an interpretation as a *binding* of all *free variable*s of the predicate expression to *value*s; cf. *satisfiable*.)

490 **Validation:** (Let, in the following *universe of discourse* stand consistently for either *domain*, *requirements* or *software design*.) By universe of discourse validation we understand the assurance, with universe of discourse

stakeholders, that the specifications produced as a result of universe of discourse acquisition, universe of discourse analysis and *concept formation*, and universe of discourse domain *modelling* are commensurate with how the stakeholder views the universe of discourse. (*Domain* and *requirements validation* is treated in Vol. 3, Chaps. 14 and 22.)

491 **Valuation:** Same as *evaluation*.

492 **Value:** From (assumed) Vulgar Latin *valuta*, from feminine of *valutus*, past participle of Latin *valere* to be of worth, be strong [209]. (Commensurate with that definition, value, to us, in the context of programming (i.e., of software engineering), is whatever mathematically founded *abstraction* can be captured by our *type* and *axiom systems*. (Hence numbers, truth values, *tokens*, sets, Cartesians, lists, maps, functions, etc., of, or over, these.))

493 **Variable:** (i) From Latin *variabilis*, from *variare* to vary; (ii) able or apt to vary; (iii) subject to variation or changes [209]. (Commensurate with that definition, a variable, to us, in the context of programming (i.e., of software engineering), is a *placeholder*, for example, a *storage location* whose *contents* may change. A variable, further, to us, has a name, the variable's identifier, by which it can be referred.)

494 **VDM:** `VDM` stands for the `Vienna Development Method` [44, 45]. (`VDM-SL` (SL for Specification Language) was the first formal specification language to have an international standard: `VDM-SL, ISO/IEC 13817-1: 1996.` The author of this book coined the name `VDM` in 1974 while working with Hans Bekič, Cliff B. Jones, Wolfgang Henhapl and Peter Lucas, on what became the `VDM` description of `PL/I`. The IBM Vienna Laboratory, in Austria, had, in the 1960s, researched and developed semantics descriptions [18, 19, 20, 144] of `PL/I`, a programming language of that time. "JAN" (John A.N.) Lee [135] is believed to have coined the name `VDL` [136, 143] for the notation (the Vienna Definition Language) used in those semantics definitions. So the letter M follows, lexicographically, the letter L, hence `VDM`.)

495 **VDM–SL:** `VDM-SL` stands for the `VDM` Specification Language. (See entry `VDM` above. Between 1974 and the late 1980s `VDM-SL` was referred to by the acronym `Meta-IV`: the fourth metalanguage (for language definition) conceived at the IBM Vienna Laboratory during the 1960s and 1970s.)

496 **Verification:** By verification we mean the process of determining whether or not a specification (a description, a prescription) fulfills a stated property. (That stated property could (i) either be a property of the specification itself, or (ii) that the specification relates, somehow, i.e., is correct with respect to some other specification.)

497 **Verify:** Same, for all practical purposes, as *verification*.

498 **Well-formedness:** By well-formedness we mean a concept related to the way in which *information* or *data structure* definitions may be given. Usually these are given in terms of *type definition*s. And sometimes it is not possible, due to the *context-free* nature of type definitions. (Well-formedness is here seen separate from the *invariant* over an *information* or a *data structure*. We refer to the explication of *invariant*!)

499 **Wildcard:** A special symbol that stands for one or more characters. (Many operating systems and applications support wildcards for identifying files and directories. This enables you to select multiple files with a single specification. Typical wildcard designators are * (asterisk) and _ (underscore).)

500 **Word:** A speech sound or series of speech sounds or a character or series of juxtaposed characters that symbolizes and communicates a meaning without being divisible into smaller units capable of independent use [209].

501 **Z:** Z stands for Zermelo (Frankel), a set theoretician. (Z also stands for a model-oriented specification language [203, 204, 226, 108, 107].)

D

Time and Space

"SLIDE 971"

D.1 van Benthem's Theory of Time

The following is taken from Johan van Benthem [218]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering ($<$, $=$, $>$) relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the application at hand, while also fitting our intuition.

"SLIDE 972"

Time is transitive: If $p<p'$ and $p'<p''$ then $p<p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time. One could designate a time axis as beginning at some time, that is, having no predecessor times. And one can designate a time axis as ending at some time, that is, having no successor times. General, past and future successors (predecessors, respectively successors in daily talk) can be defined. Time can be dense: Given any two times one can always find a time between them. Discrete time can be defined.

"SLIDE 973"

axiom

[TRANS: Transitivity] $\forall$ p,p′,p″:P • p < p′ < p″ ⇒ p < p″

[IRREF: Irreflexitivity] $\forall$ p:P • p $\not<$ p

[LIN: Linearity] $\forall$ p,p′:P • (p=p′ $\vee$ p<p′ $\vee$ p>p′)

[L−LIN: Left Linearity]
$\forall$ p,p′,p″:P • (p′<p $\wedge$ p″<p) ⇒ (p′<p″ $\vee$ p′=p″ $\vee$ p″<p′)

[BEG: Beginning] ∃ p:P • ∼∃ p′:P • p′<p

[END: Ending] ∃ p:P • ∼∃ p′:P • p<p′

"slide 974"

[SUCC: Successor]
  [PAST: Predecessors] ∀ p:P,∃ p′:P • p′<p
  [FUTURE: Successor] ∀ p:P,∃ p′:P • p<p′

[DENS: Dense] ∀ p,p′:P (p<p′ ⇒ ∃ p″:P • p<p″<p′)

[DENS: Converse Dense] ≡ [TRANS: Transitivity]
  ∀ p,p′:P (∃ p″:P • p<p″<p′ ⇒ p<p′)

[DISC: Discrete]
  ∀ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p<p″ ∧ ∼∃ p‴:P • (p<p‴<p″))) ∧
  ∀ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p″<p′ ∧ ∼∃ p‴:P • (p″<p‴<p′)))

A strict partial order, SPO, is a point structure satisfying TRANS and IRREF. TRANS, IRREF and SUCC imply infinite models. TRANS and SUCC may have finite, "looping time" models.

D.2 Blizard's Theory of Time-Space

"slide 975"

We shall present an axiom system (Wayne D. Blizard, 1980, [48]) which relates abstracted entities to spatial points and time. Let $A, B, \ldots$ stand for entities, $p, q, \ldots$ for spatial points; and t, τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p, q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator applicable, pairwise to entities, spatial locations and times, respectively. A^t_p expresses that entity A is at location p at time t. We omit (obvious) typings of A, B, P, Q, and T. The suffix prime, $'$, designates the time successor function. Thus t' designates the next time after t.

"SLIDE 976"

$$
\begin{array}{lrll}
(I) & \forall A \forall t \exists p & : & A_p^t \\
(II) & (A_p^t \wedge A_q^t) & \supset & p = q \\
(III) & (A_p^t \wedge B_p^t) & \supset & A = B \\
(IV) & (A_p^t \wedge A_p^{t'}) & \supset & t = t' \\
(V\ i) & \forall p, q & : & N(p,q) \supset p \neq q \qquad \text{Irreflexivity} \\
(V\ ii) & \forall p, q & : & N(p,q) = N(q,p) \qquad \text{Symmetry} \\
(V\ iii) & \forall p \exists q, r & : & N(p,q) \wedge N(p,r) \wedge q \neq r \qquad \text{No isolated pts.} \\
(VI\ i) & \forall t & : & t \neq t' \\
(VI\ ii) & \forall t & : & t' \neq 0 \\
(VI\ iii) & \forall t & : & t \neq 0 \supset \exists \tau : t = \tau' \\
(VI\ iv) & \forall t, \tau & : & \tau' = t' \supset \tau = t \\
(VII) & A_p^t \wedge A_q^{t'} & \supset & N(p,q) \\
(VIII) & A_p^t \wedge B_q^t \wedge N(p,q) & \supset & \sim (A_q^{t'} \wedge B_p^{t'})
\end{array}
$$

- (II–IV,VII, VIII): The axioms are universally 'closed', that is, we have omitted the usual $\forall A, B, p, q, t$s.
- (I): For every entity, A, and every time, t, there is a location, p, at which A is located at time t.
- (II): An entity cannot be in two locations at the same time.
- (III): Two distinct entities cannot be at the same location at the same time.
- (IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture, and could be questioned.*
- (V): These three axioms define N.
- (V i): Same as $\forall p :\sim N(p,p)$. "Being a neighbour of", is the same as "being distinct from".
- (V ii): If p is a neighbour of q, then q is a neighbour of p.
- (V iii): Every location has at least two distinct neighbours.
- (VI): The next four axioms determine the time successor function $'$.
- (VI i): A time is always distinct from its successor: Time cannot rest. There are no time fix points.
- (VI ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.
- (VI iii): Every nonbegin time has an immediate predecessor.
- (VI iv): The time successor function $'$ is a one-to-one (i.e., a bijection) function.
- (VII): The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time t', then p and q are neighbours.
- (VIII): *No "switching":* If entities A and B occupy neighbouring locations at time t the it is not possible for A and B to have switched locations at the next time t'.

"SLIDE 977"

Discussion of the *Blizard* Model of Space/Time

Except for axiom (IV) the system applies to systems of entities that "sometimes" rest, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities "occupy more" space volume than others, then we may suitably "repair" the notion of the point space P (etc.), however, this is not shown here.

D.3 Discussion

"SLIDE 978"

Dines Bjørner

SOFTWARE ENGINEERING

Volume II: A Model Development

March 26, 2008. Compiled December 17, 2008, 15:56

To be submitted, late 2008, for evaluation, to

Springer
Berlin Heidelberg New York
Hongkong London
Milan Paris Tokyo

Document History

- Version 1 released March 26, 2008:
  - ⋆ A first "vastly incomplete" draft of this document was conceived March 26, 2008 and essential parts of Appendices F–K and N–P were "lifted" from [40].
- Version 2 released April 19, 2008:
  - ⋆ On Sunday April 6, 2008, a complete reorganisation of the material assembled and written and rewritten by then took place — resulting in basically the current structure.
  - ⋆ A week, April 6–11, 2008, was then spent on "fattening" the syntactic structure of this textbook. The Pre- and Postlude appendices were added to Parts V and VI.
  - ⋆ "Serious", tentatively "concluding draft" work on Chap. 1 started on Saturday April 12, 2008.
  - ⋆ Work on Chap. 1 and Appendix E progressed significantly during the week of April 12–19, 2008.
- Version 3 released May 7, 2008:
  - ⋆ Draft copy notice inserted.
  - ⋆ Cross-referecing between Vol. 1 text and Vol. 1 slides pages. So far no check has been made for "synchroneity".
  - ⋆ Thus Vol. 1 text margin numbers refer to Vol. 2 slide numbers.
  - ⋆ Notes on 'A Possible (12 week) Course Plan' inserted into Preface, pages 11–14.
  - ⋆ Lecture plan inserted as first four slides: 2–6 incl.
- Version 6 released July 20, 2008:
  - ⋆ Worked on Appendix H.
  - ⋆ I am, as of today, July 20, 2008, not happy with Sect. H.2.
  - ⋆ It's treatment of 'states' is too long-winded.
  - ⋆ I believe my ideas on Sect. H.4 will change the former sections.
  - ⋆ I am starting on Sect. H.4 on page 399 tomorrow, July 21, 2008.

Contents

VOLUME I: THE TRIPTYCH METHOD

Part I Opening

Part V

Domain Engineering

E

Prelude Domain Engineering Actions

"SLIDE 980"

By 'prelude domain engineering actions' we mean those which precede domain modelling proper. That is establishment and initial editing of informative documents (Sect. E.1), stakeholder identification (and first contacts) (Sect. E.2), domain acquisition (Sect. E.3), domain analysis and concept formation (Sect. E.4), rough sketching of domain business processes (Sect. E.5), and establishment and initial editing of domain terminology (Sect. E.6).

E.1 Informative Domain Documents

"SLIDE 981"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 1, Sect. 1.6 (Page 7).

Recall, from Page 7, that the information documents are composed from:

1 Project Name and Date — Sect. E.1.1 on the following page
2 Project Place(s) ('where') — Sect. E.1.2 on the next page
3 Partners ('whom') — Sect. E.1.2 on the following page
4 Project: Background and Outlook
   (a) Current Situation — Sect. E.1.3 on page 319
   (b) Needs and Ideas — Sect. E.1.4 on page 320
   (c) Concepts and Facilities — Sect. E.1.5 on page 321
   (d) Scope and Span — Sect. E.1.6 on page 322
   (e) Assumptions and Dependencies — Sect. E.1.7 on page 323
   (f) Implicit/Derivative Goals — Sect. E.1.8 on page 324
   (g) Synopsis — Sect. E.1.9 on page 325 "slide 982"
5 Project Plan
   (a) Software Development Graph — Sect. 1.6.10 on page 13
   (b) Resource Allocation — Sect. E.1.11 on page 327
   (c) Budget Estimate — Sect. E.1.12 on page 328
   (d) Standards Compliance — Sect. E.1.13 on page 329
6 Contracts and Design Briefs — Sect. E.1.14 on page 330
7 Logbook — Sect. E.1.15 on page 331

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 1, Sect. 1.6 (Page 7). "slide 983"

E.1.1 Project Name and Dates

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.1, Page 8.

Project Name and Dates

- **Project Name:** *TransDOM:* A model of a transport domain
- **Dates:** Summer 2008 – fall 2009

E.1.2 Project Partners and Places

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.2, Page 8.

Project Partners and Places

The following information is purely speculative. It is, however, indicative of the kind of partners that a project like *A Model of A Transport Domain* ought engage.

- **Client:**
  - ⋆ *Institution:* Vejdirektoratet (Directorate of Roads) vd@vd.dk,
  - ⋆ *Address:* Niels Juels Gade 13, P.O.Box 9018, DK-1022 København K, Denmark

  Vejdirektoratet is part of the Danish ministry of transport

PG.:985

- **Developer:** DTU Transport, www.dtf.dtu.dk
  - ⋆ *Contact:* Prof., Dr. Otto Anker Nielsen, oan@transport.dtu.dk
  - ⋆ *Institution:* Bygningstorvet 116 Vest, Technical University of Denmark.
  - ⋆ *Address:* DK-2800 Kgs. Lyngby
- **Project Consultant:**
  - ⋆ *Person:* Dines Bjørner bjorner@gmail.com
  - ⋆ *Address:* Fredsvej 11, DK-2840 Holte, Denmark
- **Funding Agency:** None.

PG.:986

- **Project Audience:**
  - ⋆ **ENPC:**
    - · *Institution:* École Nationale des Ponts et Chaussées
    - · *Address:* 6–8 Avenue Blaise Pascal, Cité Descartes, 77455 Champs-sur-Marne, Marne la Vallée, Cedex 2, France. www.enpc.fr/english/int_index.htm
  - ⋆ **BTAC:**
    - · *Institution:* British Transport Advisory Commission
    - · *Address:* PO Box 9108, Maldon, Essex, CM9 5HG, UK. www.btac.org.uk

PG.:987

- ⋆ **DLR:**
  - · *Contact Person:* Dr.-ing. Michael Meyer zu Hörste
  - · *Institution:* Deutsches Zentrum für Luft- und Raumfahrt (DLR),
  - · *Address:* Lilienthalplatz 7, D-38108 Braunschweig, Germany. www.dlr.de
- ⋆ **SafeTRANS:** (Safety in Transportation Systems) www.safetrans-de.org
  - · *Contact Person:* Prof. Dr. Werner Damm,
  - · *Institution:* CvO Universität Oldenburg ,
  - · *Address:* OFFIS, Escherweg 2, DE-26121 Oldenburg, Germany

"slide 988"

E.1.3 Current Situation

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.3, Page 9.

Current Situation

The current situation in the infrastructure of transportation (road, rail, air traffic and, to some extent also shipping), can be summarised as follows:

- **Congestion:** Roads, rail tracks and air lanes are are rapidly reaching saturation wrt. what current technology support can monitor and control.

  PG.:989
- **Interfaces:** The interfaces between different kinds of person and freight transport modes (road, rail, air, sea) are not well understood: transfers from one mode to another mode is often a bottleneck for efficient transport time and economic transitions.

  PG.:990
- **Training:** Quality of training of new staff in the oftentimes intricate properties of transport systems is hap-hazard: trainers are usually second-rate staff members of current transport sub-systems; their training material is inconsistent and incomplete, lacks proper abstractions and must hence often delve into far too concrete case studies; and new staff members drop out and leave their working place to seek other vocations.

  PG.:991
- **Intelligent Transport:** This, 'intelligent transport' is currently a buzzword. It covers over real-time, "just-in-time" scheduling and allocation of transport resources (roads, rail tracks, air lanes, and their vehicles (cars, trains, aircraft)) and the monitoring and control of actual traffic. But too little is known of the transport domain to assess whether proposals for software for intelligent transport are feasible and will satisfy expectations.

  PG.:992
- **Safe Transport:** Traffic signals, whether for road, rail or air traffic, are oftentimes posing safety hazards: not only is the software development not based on a sufficiently thorough knowledge of the domain, but the software

is either not the right software for the problem at hand, or the software is not right, i.e., is incorrect, or both !

PG.:993

- **Confusion and Bewilderment:** The sum total of the above and many other facets not mentioned here, is that responsible management, who themselves cannot be guaranteed to have an objective, clear understanding of their own domain, is exposed: do not know whether they have to prerequisite knowledge to make day-to-day operational, or tactical, or strategical decisions: how to clearly formulate and manage the business processes and how to institute necessary re-engineerings of these business processes including acquiring the right software and get that software right !

"SLIDE 994"

E.1.4 Needs and Ideas

Needs

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.4, Page 9.

Needs

There is thus a clear need to clarify what is meant by transportation:

- how the transport nets of different modality and capacity (roads, rail tracks, air lanes, etc.), interact with their users (automobiles, trains, aircraft and ships);
- how scheduling and allocation of one modality of transport resources interact with other modalities of transport resources;
- how monitoring and control of one modality of traffic interact with other modalities of traffic;
- etcetera.

"slide 995"

Ideas

Ideas

A clarification is hoped to be achieved:

- by conducting a research-oriented study of transportation in general;
- by augmenting this study with a number of studies of
  - ⋆ road transportation (including road traffic),
  - ⋆ rail transportation (i.e., railways, including train traffic),
  - ⋆ air traffic (including the "the players":
    - · airports,
    - · airlines,
    - · civil aviation authorities,
    - · GAO, etcetera.

  - ⋆ shipping, for example
    - · container line industry
      - · containers,
      - · container vessels,
      - · container terminal ports,
      - · shipping companies, etc.
    - · passenger and automobile boats and ferries, and
    - · pleasure cruise industry.

PG.:996

- and by letting these study projects be based on domain research and engineering — along the lines suggested in this book (and previously in [33, 34, 35]) —
- aimed at establishing
  - ⋆ a generic domain model of conceptual transport as well as
  - ⋆ instantiated domain models of each of the different transport modes
  - ⋆ such that all models are both informally narrated,
  - ⋆ formally specified, and
  - ⋆ related to existing operations research models of transport.

"SLIDE 997"

E.1.5 Concepts and Facilities

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.5, Page 10.

Concepts and Facilities

The following is a brief list of the facilities (observable phenomena) normally considered when dealing with the transport infrastructure:

- The facilities of
  - ⋆ road nets, rail nets, air lane and airline nets, and shipping nets
  - ⋆ are "lifted" into concepts of
    - · hubs being road intersections, rail switches, crossovers, etc., airports and harbours, and
    - · links being street segments, rail tracks, air and shipping lanes (all between adjacent hubs).

PG.:998

- The facilities of
  - ⋆ automobiles, trains, aircraft and vessels
  - ⋆ are lifted into the concepts of vehicles.

PG.:999

- Etcetera for facilities and concepts such as:
  - ⋆ monitoring and control of traffic and of traffic signals;
  - ⋆ booking and tracing of transport (incl. passenger travel, ticketing and freight transport);

- ⋆ scheduling and allocation of resources (net, vehicle, passenger, freight);
- ⋆ monitoring and control of vehicles, individual, in traffic queues and in convoys;
- ⋆ the loading and unloading of passengers or freight on and from vehicles and the transfer of passengers or freight between vehicles;
- ⋆ etcetera, etcetera.

"SLIDE 1000"

E.1.6 Scope and Span

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.6, Page 11.

Scope

The scope of the project is the entire transportation infrastructure:

- the public and private road system, from country lanes to freeways and toll-roads, vehicles, drivers, signalling, toll plazas, etc.;
- the public and private rail system: the possibly (usually) shared net, the train stations, trains, traffic, passengers, travel inquiries, ticketing, freight services (including freight tracing), etc.;
- the public and private airline and (usually public) air traffic system, aircraft, passengers, airports, luggage handling, etc.
- the public and private (line and tramp) shipping, vessels, containers, container terminal ports, etc.

"slide 1001"

Span

- The span of the project is the transport net, vehicles and their traffic, but restricted to
  - ⋆ road and
  - ⋆ rail

  transport. Besides covering
  - ⋆ freeways and toll-roads, vehicles, drivers, signalling, toll plazas, etc., and
  - ⋆ shared net, the train stations, trains, traffic, passengers, travel inquiries, ticketing, freight services (including freight tracing), etc.
- The span includes all the interfaces between road and rail transport:
  - ⋆ roads connection to train stations,
  - ⋆ same level rail track and road crossings,
  - ⋆ combined ticket for road/rail (for example auto train) travel,
  - ⋆ etcetera.

"slide 1002"

E.1.7 Assumptions and Dependencies

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.7, Page 11.

Assumptions

The assumptions are that a domain knowledge reference group can be established, that is, that there is, somewhere, the domain knowledge that can and must be harnessed, that is:

- **Accessibility:** that the people with that knowledge can be accessed, i.e., that they are known, that is, identifiable. These people are from both the client, the Danish 'Directorate of Roads' (DoR), from within the developer 'DTU Transport', and from amongst the 'Project Audience' (ENPC, BTAC, DLR, SafeTRANS).

PG.:1003

- **The Domain Stakeholder Reference Group:** In particular the following persons are identified:

  - ◇ *DoR:*
    - Mr NN_1
    - Mrs NN_2
    - Miss NN_3
  - ◇ *DTU Transport:*
    - Mr NN_4
    - Mrs NN_5
    - Miss NN_6
  - ◇ *ENPC:*
    - Mr NN_7
    - Mrs NN_8
    - Miss NN_9
  - ◇ *BTAC:*
    - Mr NN_{10}
    - Mrs NN_{11}
    - Miss NN_{12}
  - ◇ *DLR:*
    - Mr NN_{13}
    - Mrs NN_{14}
    - Miss NN_{15}
  - ◇ *SafeTRANS:*
    - Mr NN_{16}
    - Mrs NN_{17}
    - Miss NN_{18}

PG.:1004

- **Availability:** It must be guaranteed that these people are available on a suitable basis — to be negotiated.
- **Professionalism:** As a group they represent or are aware of the essential knowledge of the domain.
- **Communicable:** The domain engineers must be able to communicate with this reference group, i.e., in own, natural languages.
- **Cooperative:** They must be co-operative.

A trial period of 2 months shall be set aside to check that the Domain Knowledge reference group can be securely established.

"slide 1005"

The borderline between 'assumptions' and 'dependencies' is not sharp.

Dependencies

- **Funding:** The project cannot start without assured funding. See the budget item Sect. E.1.12.
- **Quality of R&D Staff:** The involved developers and members of the domain stakeholder reference group must be certified by their management and by

the project board to possess both necessary and sufficient knowledge of their respective domains.
- **Stability of R&D Staff:** The developer staff, the consultant and the domain stakeholder reference group members must be stable throughout the entire project period.
- **Seriousness of Management:** The TRANSDOM project management must be guaranteed to set aside sufficient monthly monitoring review and advisory time as well as dispatch their control in a regular and timely fashion.

E.1.8 Implicit/Derivative Goals

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.8, Page 12.

Implicit/Derivative Goals

- **Improved Management:** It is expected that the TRANSDOM project results, once studied by all relevant transportation infrastructure (tactical and operational) management teams, will lead to a further professionalisation of their management.
- **Trained Staff:** It is expected that the resulting TRANSDOMain description can serve as a basis for developing educational, teaching and training material (on-line demos and paper documents); that such material will be used in future training courses for (possibly existing and) newly hired staff; and that staff trained in this manner will improve the daily operations within the transportation infrastructure.

PG.:1007

- **Basis for (Software) Requirements:** The existence of a TRANSDOMain description does not imply any requirements prescriptions, but once such a TRANSDOMain description exists one can more easily embark on any number of specific software requirements prescriptions.
  - ⋆ **Software Product Family:** So, a 'normative' TRANSDOMain description can give rise to a whole family of specific software requirements prescriptions and requirements for specific optical/electromechanical and electronic as well as requirements for other sensor and actor, equipment related to the software requirements.
  - ⋆ **Open-endedness:** Furthermore these requirements can be "fitted" to one another so that they can eventually be interfaced with one another in an all-encompassing system of software for the transportation infrastructure. They are open-ended in that new requirements can be developed long time after completions of first software packages and still guaranteeing "fitness".

PG.:1008

- **Improved Public Image:** The image of a highly competent and sophisticated transportation infrastructure industry should emerge from the TRANSDOM during the project and for some time after its completion. This should be the result of a carefully planned and executed information campaign in a diversity of media: the press, radio, TV, at scientific and technological events and at commercial exhibits trade shows and fairs. Such an improved image should further the ease with which the transportation infrastructure industry can recruit competent and motivated staff.

  PG.:1009
- **Input to 'Normative Standardisation' Agencies:** Nationally, regionally and internationally there are several transportation agencies:
  - ⋆ The *International Road and Transport Union* www.iru.org
  - ⋆ The *International Union of Railways* (Union Internationale des Chemins de fer) www.uic.asso.fr
  - ⋆ The *International Union of combined Rail/Road transport companies* www.uirr.com
  - ⋆ The *International Maritime Organization (IMO)*, a UN body www.imo.org
  - ⋆ The *International Civil Aviation Organization (ICAO)* www.icao.int
  - ⋆ The *ISO: International Organisation for Standardization* www.iso.org/iso/home.htm
  - ⋆ The *CENELEC: European Committee for Electrotechnical Standardization* /www.cenelec.eu/Cenelec/Homepage.htm
  - ⋆ Etcetera

  It is hoped that the result of the TRANSDOM project can help these organisations attain a higher level of interaction — etcetera.

"SLIDE 1010"

E.1.9 Synopsis

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.9, Page 13.

Synopsis

- **Aims:** The TRANSDOM project aims at researching and developing an as complete, both informal and formal description of a, or "the", road and rail transportation domain.

  PG.:1011
- **Objectives:** The TRANSDOM project objective is to establish a 'normative' model of the road and rail transportation domain, one
  - ⋆ that can become an input to national, regional and international transportation 'bodies' (, etcetera);

  - ⋆ that can serve as a basis for training a largest variety of staff of the transportation infrastructure (including its providers of transportation equipment);
  - ⋆ that can serve as a basis for operations analysis R&D into traffic models, etcetera; and
  - ⋆ that can serve as a basis for research and development of
    - · software for the support of transportation,and
    - · sensor and actuator equipment for vehicle and traffic monitoring and control.

PG.:1012

- **A Road and Rail Transportation Infrastructure Rough Sketch:**
  - ⋆ *Some Entities:* roads (street segments and intersections), rails (rail points (switches, crossovers, etc.)), tracks between rail points, cities (stations), vehicles (automobiles and trains), signal equipment, passengers, etc.

PG.:1013

  - ⋆ *Some Functions:* insertion of new roads (or rail tracks), removal of old roads (or rail tracks), setting of street intersection signals (rail switches), insertion [removal] of vehicles into [from] the traffic, etc.

PG.:1014

  - ⋆ *Some Events:* Road [rail track] breakdown (tantamount to, hopefully temporary, road [rail track] removal), vehicle-to-vehicle crash (whether automobile/automobile, automobile/train, or train/train), vehicle [automobile or train] entering or leaving the traffic, vehicle [automobile or train] exceeding the speed limit, etc.

PG.:1015

  - ⋆ *Some Behaviours:* A train ride: from entering the traffic, starting at train station of origin, travelling down tracks, visiting other stations, finally arriving at destination and being "removed" from the traffic. Similarly to the above, but for automobiles. A passenger behaviour: planning a trip, buying tickets, starting the trip, possibly changing reservations, finally ending the trip. The life-time of a road segment: being inserted into the transportation net, being trafficked, being serviced (maintained, repaired), being temporarily — or finally — taken out of service, etc.

PG.:1016

- **Project Partners and Places:** A summary is to be given here of the project and partners of Sect. E.1.2 on page 318.
- **Project Economy:** A summary is to be given here of the budget estimate of Sect. E.1.12 on page 328.

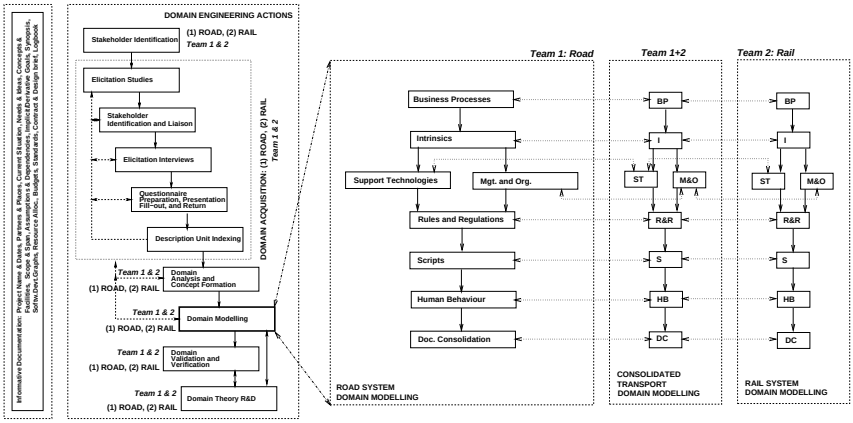
"SLIDE 1017"

E.1.10 Software Development Graphs

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.10, Page 13.

Software Development Graph

The basis for a software development graph for the TransDOM project is given below.



"slide 1018"

The above graph has been developed from the domain engineering process graph of Fig. 2.3 on page 103. One way of understanding the task of developing a description of the transport domain is to describe the concrete domains of road transport and of rail transport separately. Then "lift" these descriptions into a a description of the abstract domain of transport. With this interpretation of the task at hand we show three "concurrent", *Team 1 & 2* developments. "slide 1019"

Some, perhaps most, of the horisontal boxes (denoting development activities) of the above figure may be "blown up" to show details of precedence-related sub-activities to be carried out by various project sub-groups and between them. Such an (intermediate) expansion has been shown for the domain acquisition and the domain modelling actions, but they each need further elaboration. "slide 1020"

E.1.11 Resource Allocation

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.11, Page 15.

Resource Allocation

The possibly expanded action boxes of the above software development graph need now be attributed (cf. Defn. 14 on page 15). The vertical 'informative documentation' box denotes a project-duration management activity and its resource consumption must be estimated. We give examples of resource estimates

for a few activity attributes and for a few of the horisontal domain engineering actions:

PG.:1021

- **Stage Action:** Stakeholder identification
  - ⋆ **Total Time Period:** 2 months
  - ⋆ **Staff Requirements:**
    - · **Staff #1 and #2:** domain engineers, full time
    - · **Staff #3:** admin. secr. part time, 1.5 hour per day, corresponding to 1/5 full time
  - ⋆ **Equipment:** two office modules (one , two laptop PCs

PG.:1022

- **Stage Action:** Domain acquisition
  - ⋆ **Total Time Period:** 4 months
  - ⋆ **Staff Requirements:**
    - · **Staff #1, #2 and #3:** domain engineers, full time
    - · **Staff #4:** admin. secr. part time, 1.5 hour per day, corresponding to 1/5 full time
  - ⋆ **Equipment:** three office modules, four laptop PCs, weekly full day access to 8 person meeting room

PG.:1023

- **Stage Action:** Domain Analysis
  - ⋆ **Total Time Period:** 4 months concurrent with domain acquisition, Item E.1.11
  - ⋆ **Staff Requirements:**
    - · **Staff #1, #2 and #3:** shared with staff #1, #2 and #3 of Item E.1.11
    - · **Staff #4:** shared with staff #4 of Item E.1.11
  - ⋆ **Equipment:** — multiple use of that of Item E.1.11
- Etcetera

E.1.12 Budget Estimate "SLIDE 1024"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.12, Page 16.

Budget Estimate

We can only sketch, due to page limitations, how one might arrive at a budget estimate:

- **Stage Action:** Stakeholder identification
  - ⋆ **Total Time Period:** 2 months
  - ⋆ **Staff Costs:**
    - · **Staff #1 and #2:** $2\times$ ($1\times$ Snr.R&D Eng. + $1\times$ Jnr.R&D Eng.) monthly cost

- **Staff #3:** $\frac{1}{5}\times$ Admin.Secr. monthly cost
- ⋆ **Equipment:**
  - **Office Space:** $1\frac{1}{5}\times$ office module monthly cost
  - **Computers & Communication:** $4\times1+\frac{1}{5}$ laptop + shared Internet monthly cost
  - **Overhead:** — corresponding to three staff members

PG.:1025

- **Stage Action:** Domain acquisition
  - ⋆ **Total Time Period:** 4 months
  - ⋆ **Staff Costs:**
    - **Staff #1, #2 and #3:** $4\times$ (1 × Snr.R&D Eng. + 2× Jnr.R&D Eng.) monthly cost
    - **Staff #4:** $4\times\frac{1}{5}$ Admin.Secr. monthly cost
  - ⋆ **Equipment:**
    - **Office Space:** $4\times2\frac{1}{5}$ office module monthly cost
    - **Meeting Room:** $4\times\frac{1}{5}$ meeting room monthly cost
    - **Computers & Communication:** $4\times2\frac{1}{5}$ lap top + shared Internet monthly cost
    - **Overhead:** — corresponding to four staff members

PG.:1026

- **Stage Action:** Domain Analysis
  - ⋆ **Total Time Period:** 4 months concurrent with domain acquisition, Item E.1.12
  - ⋆ **Staff Costs:** shared with Item E.1.12
  - ⋆ **Equipment:** shared with Item E.1.12
  - ⋆ **Overhead:** shared with Item E.1.12
- Etcetera

"SLIDE 1027"

E.1.13 Standards Compliance

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.13, Page 16.

Standards Compliance

There are no standards, recommendations or guidelines issued by ISO or IEEE for the scientific and technical aspects of domain engineering. Existing ISO and IEEE standards that appears to apply, more generally, and to be followed by this project, are listed below. One should, however, keep in mind two things: These standards, recommendations or guidelines are formulated in a context which was unaware of the concept of domain engineering. They must therefor be reinterpreted in this new light of domain engineering. This book sets it own standards and they must be followed !

PG.:1028

- ISO 9001: Quality Systems Model for quality assurance in design, development, production, installation and servicing
- ISO 9000-3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software
- ISO 12207: Software Life Cycle Processes `http://www.12207.com`
- IEEE Std 1058.101987, Standard for Software Project Management Plans
- IEEE Std 1074.1-1995, Guide for Developing Software Life Cycle Processes
- IEEE Std 730.1-1995, Guide for Software Quality Assurance Plans
- IEEE Std 1063-1987 (reaffirmed 1993), Standard for Software User Documentation
- Software Process Improvement Models and Standards, including SEI's various Capability Maturity Models

The TRANSDOM project is to first follow the principles and techniques laid down in this book, then accommodate the above standards, recommendations or guidelines wherever applicable.

"SLIDE 1029"

E.1.14 Contract and Design Brief

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.14, Page 19.

Contract

We omit an example contract. Any subset example, however large, but still being a subset fitting into this book's page limitations, would not be a true example, but only a "joke" !

"slide 1030"

Design Brief

The following documents can shall be developed, delivered on dates shown and according to the principles and techniques of this book:

- **Stakeholder Identification:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft
- **Road, Rail & Transport Domain Acquisition:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft
- **Road, Rail & Transport Domain Analysis:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft

PG.:1031

- **Domain Terminology:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, initial draft

- **Domain Business Process Rough Sketches:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Initial draft
- **Road, Rail & Transport Intrinsics:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft

PG.:1032

- **Road, Rail & Transport Support Technologies:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft
- **Road, Rail & Transport Mgt. & Org.:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft
- **Road, Rail & Transport Rules & Regs.:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft

PG.:1033

- **Road, Rail & Transport Human Behaviours:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft
- **Domain Validation:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final draft
- **Domain Terminology:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final version
- **Consolidated Domain Description:**
  - ⋆ **Delivered:**
  - ⋆ **Status:** Complete, final version

"SLIDE 1034"

E.1.15 Logbook

We omit an example logbook. Example 1 on page 23 should suffice.

E.2 Stakeholder Identification

"SLIDE 1035"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.4 (Page 56).

Stakeholder Identification

Railways		**Roads**	
State Rail Authority		Directorate of Roads	
	Exec.Mgt.		Exec.Mgt.
	Planners		Planners
Rail Infrastructure Co.		Regional Road Authority	
	Exec.Mgt.		Exec.Mgt.
	Planners		Planners
	Rail Engs.		Road Engs.
	Rail Wrkrs.		Road Wrkrs.
	Etcetera		Etcetera
Rail Operators	DSB, SJ, DB	City Road Authority	
	Mgt.		Exec.Mgt.
	Planners		Planners
	Loco Drivers		Road Engs.
	Train Staff		Road Wrkrs.
	Ticket. Staff	Trucking Companies	
	Train Maint.		Mgt.
	Etcetera		Dispatchers
Passengers			Drivers
Freight Companies		Bus Companies	PG.:1036
Travel Agencies			Mgt.
	Mgt.		Dispatchers
	Service Staff		Drivers
		Gasoline Companies	
		Taxi Companies	
		Private Drivers	
Rail Regulatory Agency		Road Regulatory Agency	
Ministry of Transport		Ministry of Transport	
Rail Equipment Providers		Road Equipment Providers	
Politicians		Politicians	
Insurance Companies		Insurance Companies	
Etcetera		Etcetera	

E.3 Domain Acquisition

"SLIDE 1037"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.5 (Page 57).

We will present two examples: one for road transport, another for rail transport. The examples are just very short. We do not exemplify all the introductory aspects of solicitation and elicitation. Just resulting, slightly edited domain acquisition units.

E.3.1 Road Transport

The first example is that of road transport: entities, functions, events and behaviours. "slide 1038"

Domain Acquisition Units: Road Transport

1 *"A road net consists of roads."*
- **Attributes:**
  - ⋆ **Names:** Road Net, Road
  - ⋆ **Kinds:** Entities
  - ⋆ **Source:** Mr. NN_1
  - ⋆ **Date:** $DD_1MM_1YY_1$

2 *"Roads consists of road segments and intersections."*
- **Attributes:**
  - ⋆ **Names:** Road Segment, Intersection
  - ⋆ **Kinds:** Entities
  - ⋆ **Source:** Mrs. NN_2
  - ⋆ **Date:** $DD_2MM_2YY_2$

3 *"A road segment is connected to two intersections"*
- **Attributes:**
  - ⋆ **Name:** Road Segment, Intersection
  - ⋆ **Kinds:** Entities
  - ⋆ **Source:** Ms. NN_3
  - ⋆ **Date:** $DD_3MM_3YY_3$

4 *"Every intersection is connected to at least one segment"*
- **Attributes:** Entities
  - ⋆ **Names:** Intersection, Road Segment
  - ⋆ **Kinds:** Entities
  - ⋆ **Source:** Miss NN_4
  - ⋆ **Date:** $DD_4MM_4YY_4$

PG.:1039

5 *"One can enter an automobile onto a road."*
- **Attributes:**
  - ⋆ **Names:** Enter, Automobile, Road
  - ⋆ **Kinds:** Function (Enter), Entities (Automobile, Road)
  - ⋆ **Source:** Ms. NN_5
  - ⋆ **Date:** $DD_5MM_5YY_5$

6 *"Two automobile may crash on a road."*
- **Attributes:**
  - ⋆ **Names:** Automobile, Crash
  - ⋆ **Kinds:** Entity, Event
  - ⋆ **Source:** Mrs. NN_6
  - ⋆ **Date:** $DD_6MM_6YY_6$

7 *"One can Insert a new Road (Segment) between two new or existing Intersections."*

- **Attributes:**
  - ⋆ **Names:** Insert, Road Segment, Intersection
  - ⋆ **Kinds:** Function (Insert), Entities (Road Segment, Intersection)
  - ⋆ **Source:** Mr. NN_7
  - ⋆ **Date:** $DD_7MM_7YY_7$

8 *"One can Remove an existing Road (Segment) between Intersections"*

- **Attributes:**
  - ⋆ **Names:** Remove, Road Segment, Intersection
  - ⋆ **Kinds:** Function (Remove), Entities (Road Segment, Intersection)
  - ⋆ **Source:** Miss NN_8
  - ⋆ **Date:** $DD_8MM_8YY_8$

PG.:1040

9 *"Intersections that become isolated when a Road Segment is Removed are no longer part of the Road Net"*

- **Attributes:**
  - ⋆ **Names:** Intersection, Road Segment, Remove, Road Net
  - ⋆ **Kinds:** Entities (Intersection, Road Segment, Road Net), Function (Remove)
  - ⋆ **Source:** Mr. NN_9
  - ⋆ **Date:** $DD_9MM_9YY_9$

10 *"A Road Segment may "Break Down", that is, appear as having been Removed"*

- **Attributes:**
  - ⋆ **Names:** Road Segment, "Break Down", Remove
  - ⋆ **Kinds:** Entity (Road Segment), Event (Road Segment Break Down). Function (Remove)
  - ⋆ **Source:** Mr. NN_{10}
  - ⋆ **Date:** $DD_{10}MM_{10}YY_{10}$

11 *"Road Traffic is the Movement of Automobiles along Roads."*

- **Attributes:**
  - ⋆ **Names:** Road Traffic, Movement, Automobile, Road
  - ⋆ **Kinds:** Behaviour
  - ⋆ **Source:** Mr. NN_{11}
  - ⋆ **Date:** $DD_{11}MM_{11}YY_{11}$

"SLIDE 1041"

E.3.2 Rail Transport

The second example is that of rail transport: entities, functions, events and behaviours.

"slide 1042"

Domain Acquisition Units, Rail Transport

1 *"A Rail Net consists of Rail Units."*

- **Attributes:**

   - ⋆ **Names:** Rail Net, Rail Unit
   - ⋆ **Kinds:** Entities
   - ⋆ **Source:** Mr. NN_a
   - ⋆ **Date:** $DD_aMM_aYY_a$

2 *"Rail Units are either Linear, or Simple Switches, or Switchable Crossovers or Simple Crossovers."*
   - **Attributes:**
     - ⋆ **Names:** Rail Unit, Linear Rail Unit, Simple Switch Rail Unit, Switchable Crossover Rail Unit, Simple Crossover Rail Unit
     - ⋆ **Kinds:** Entities
     - ⋆ **Source:** Mrs. NN_b
     - ⋆ **Date:** $DD_bMM_bYY_b$

3 *"Linear Rail Units have two Connectors and a pair of Rails."*
   - **Attributes:**
     - ⋆ **Name:** Linear Rail Unit, Connector, Rail
     - ⋆ **Kinds:** Entities
     - ⋆ **Source:** Ms. NN_c
     - ⋆ **Date:** $DD_cMM_cYY_c$

4 *"Simple Switch Rail Units have three Connectors and a (... further defined ... structure) of Rails."*
   - **Attributes:** Entities
     - ⋆ **Names:** Simple Switch Rail Unit, Connector, ..., Rail
     - ⋆ **Kinds:** Entities
     - ⋆ **Source:** Miss NN_d
     - ⋆ **Date:** $DD_dMM_dYY_d$

PG.:1043

5 *"A Rail Line is a sequence of Connected Rail Units."*
   - **Attributes:**
     - ⋆ **Names:** Rail Line, Rail Unit, Connected
     - ⋆ **Kinds:** Entities (Rail Line, Rail Unit), Function [Predicate] (Connected)
     - ⋆ **Source:** Ms. NN_e
     - ⋆ **Date:** $DD_eMM_eYY_e$

6 *"Train Traffic is the Movement of Trains along Rails."*
   - **Attributes:**
     - ⋆ **Names:** Train Traffic, Movement, Train, Rail
     - ⋆ **Kinds:** Behaviour
     - ⋆ **Source:** Mr. NN_f
     - ⋆ **Date:** $DD_fMM_fYY_f$

7 *"Two Trains may Crash on a Rail Line."*
   - **Attributes:**
     - ⋆ **Names:** Train, Train Crash, Rail Line
     - ⋆ **Kinds:** Entities (Train, Rail Line), Event (Train Crash)

⋆ **Source:** Mrs. NN_g
⋆ **Date:** $\mathsf{DD}_g\mathsf{MM}_f\mathsf{YY}_g$

"slide 1044"

E.3.3 Review

The above two examples illustrate only a fragment of the work going into domain acquisition. You have to imagine that literally thousands of domain description units have to be collected and ascribed attributes. Clearly an activity that need be supported by an appropriate (relational) database system.

E.4 Domain Analysis and Concept Formation

"SLIDE 1045"

E.4.1 Inconsistencies

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.6 (Page 60).

Domain Analysis, Inconsistencies

We present two examples of pairs of inconsistent domain description units:

1 • *"A road segment is delimited by exactly two intersections."*
  • *"A road segment is delimited by exactly two distinct intersections."*
  The first statement allows for non-distinct intersections.
2 • *"An automobile can only enter the road net at any intersection."*
  • *"An automobile can only enter the road net at any intersection and along any road segment."*
  The first statement does not allow automobiles to enter along road segments.

"SLIDE 1046"

E.4.2 Incompleteness

Domain Analysis, Incompleteness

1 • *"An automobile can enter the road net at any intersection."*
  If the above is the only statement about where automobiles may enter the net, then it is not clear whether automobiles may also enter the net elsewhere.

"SLIDE 1047"

E.4.3 Concept Formation

Domain Concept Formation

1 Links
   - We merge the phenomena of road segments and linear rail units
   - into the concept of links.

2 Hubs
   - We merge the phenomena of (road) intersections and switch and crossover rail units
   - into the concept of hubs.

E.5 P

rocesses]Domain [i.e., Business] Processes "SLIDE 1048"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.7 (Page 62).

Business Processes

1 **An Automobile Journey:**Concerning the automobile journey route: it enters the road net at a position p_{origin} along a road segment, s_j, moves from there towards hub h_k, and, in sequence traverses a number of pairs of hubs and road segment, $(h_k, s_k), (h_\ell, s_\ell), \ldots, (h_m, s_m)$, to leave the road net at a position $p_{\text{destination}}$ along segment s_m. Concerning the journey dynamics: Along segment s_i, for $j \leq i \leq m$, the automobile travels at an average velocity of v_{s_i} kms/hour. In hub h_i, for $j \leq i \leq m$, the automobile travels at an average velocity of v_{h_i} kms/hour. The automobile may have to temporarily stop at some hubs due to red light signals. changes in velocity, from v_i to v_j, are made at a constant acceleration (or deceleration) of a_{ij} kms/second^2.

PG.:1049

2 **A Simple Road Net Topology "History":**Initially there is no road net. A first action is to insert a segment and two connecting hubs. Following actions are now a sequence of insertions and sometimes (usually only temporary) removals of segments: Sometimes an inserted segment is between two existing hubs, sometimes between one existing and one new (inserted) hub, and sometimes between two new inserted hubs. Similarly for removals. In-between these actions there may be occasional events: a road segment may be closed as the result of some accident: a bridge falling down, or a mud slide covering the road, or otherwise.

PG.:1050

3 **A Train Journey:**A train journey normally begins and ends at a station. And a train journey normally is then a sequence of station visits between,

but not including the begin and end stations. A station visit can be thought of as a triple:the train journey along a rail line leading into a station, the arrival, possible halt, and departure from the station, and the train journey along a rail line leading out from the station. The train journey is normally constrained by some timetable. A train timetable consists of a sequence of two or more timetable-station-visits. A timetable-station-visit is a quintuple:the arrival time at a designated station along a designated rail line, followed by a departure time (from the station) along a designated rail line.

E.6 Domain Terminology

"slide 1051"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.8 (Page 63).

Domain Terminology

1 **[Rail Unit] Connector:** A [rail unit] connector is a further undefined atomic (entity) concept. Its purpose is to connect two rail units, i.e., provide an interface between them.
2 **Rail Unit:** The purpose of a rail unit is to provide for train passage. A rail unit is either a linear, or a switch, or a simple crossover, or a switchable crossover unit.
3 **Linear Rail Unit:** A linear rail unit has two connectors $c1, c2$. Think of a linear rail unit as shown in Fig. E.1 [upper left quadrant]. Its purpose is to allow trains to pass through the unit from one end to the other ($c1 \rightarrow c2$), or vice versa ($c2 \rightarrow c1$).

"slide 1052"
"slide 1053"

Domain Terminology (**Continued**)

4 **Simple Switch Rail Unit:** A simple switch rail unit has three connectors: $c, c/, c$ |. Think of a simple switch rail unit as shown in Fig. E.1 [upper right quadrant]. Its purpose is to allow trains to pass through the unit in directions $c \rightarrow c$ | or opposite, or $c1 \rightarrow c/$ or opposite, as shown in Fig. E.2.

"slide 1054"

Domain Terminology (**Continued**)

5 **Simple Crossover Rail Unit:** A simple crossover rail unit has four connectors: $c1, c2, c3, c4$. See Fig. E.1 [lower left quadrant]. Its purpose is to allow trains to pass through the unit in directions $c1 \rightarrow c2$ or opposite, or $c3 \rightarrow 4$.
6 **Switchable Crossover Rail Unit:** A switchable crossover rail unit has four connectors: $c1, c2, c3, c4$. See Fig. E.1 [lower right quadrant]. Its purpose is

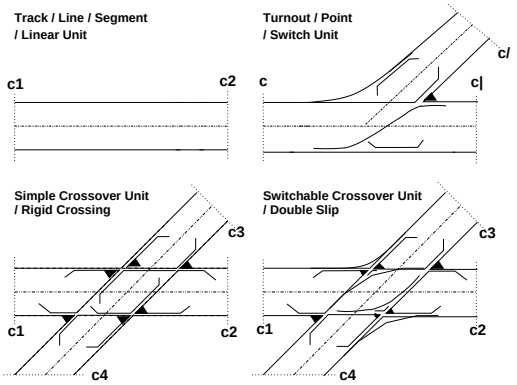


Fig. E.1. Rail Units

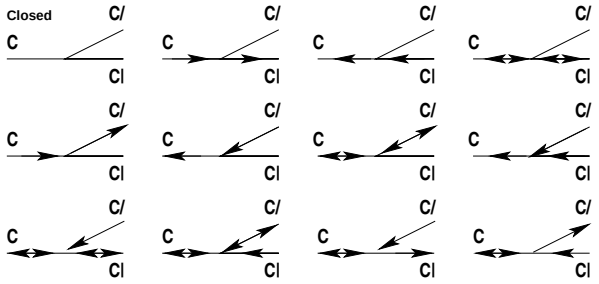


Fig. E.2. 12 possible states of a simple rail switch

to allow trains to pass through the unit in directions $c1 \rightarrow c2$ or opposite, or $c3 \rightarrow 4$ or opposite, or $c1 \rightarrow 3$ or opposite, or $c2 \rightarrow 4$ or opposite.

PG.:1055

7 **Rail Unit State:** A rail unit state is a set of pairs of distinct connectors of the rail units, namely those for which passage is possible. See Fig E.2 for the maximal 12 states of a simple switch unit.

E.7 Review

"SLIDE 1056"

E.8 Exercises

Exercise 35. Domain Prelude 1:

Solution 35 Vol. II, Page 536, suggests a way of answering this exercise.

Exercise 36. Domain Prelude 2:
Solution 36 Vol. II, Page 536, suggests a way of answering this exercise.

"SLIDE 1057"

F

Intrinsics

"SLIDE 1059"

F.1 An Essence of 'Transport'

We exemplify a transportation domain. By *transport* we shall mean *the* **movement** *of* **vehicle***s from* **hub***s to hubs along the* **link***s of a* **net**.

F.2 Business Processes

"SLIDE 1060"

We re-sketch the Business processes of Appendix Sect. E.5.

Rough Sketching of Some Transport Processes

The basic **entities** of the transportation "business" are the (i) **net**s with their (ii) **hub**s and (iii) **link**s, the (iv) **vehicle**s, and the (v) **traffic** (of vehicles on the net). PG.:1061 The basic **functions** are those of (vi) vehicles **entering** and **leaving** the net (here simplified to entering and leaving at hubs), (vii) for vehicles to **make movement** transitions along the net, and (viii) for **inserting** and **removing links** (and associated hubs) into and from the net. PG.:1062 The basic **events** are those of (ix) the **appearance** and **disappearance** of vehicles, and (x) the **breakdown** of links. PG.:1063 And, finally, the basic **behaviours** of the transportation business are those of (xi) **vehicle journey** through the net and (xii) **net development and maintenance** including insertion into and removal from the net of links (and hubs).

F.3 Simple Entities

"SLIDE 1064"

F.3.1 Basic Entities

Nets, Hubs and Links

Narrative

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

Formalisation

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

RSL Explanation

- 5: 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.
  (The fact that the type clause (5) is "spread" over two lines is immaterial.)
- 6: 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,
  - ⋆ × 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
  - ⋆ -**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.
  - ⋆ Similarly for L-**set**.

  (The fact that type clause (6), 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 6.)
- 6: The axiom **axiom** ∀ (hs,ls):N • **card** hs≥2 ∧ **card** ks≥1
- 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).
- 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

Hub and Link Identifiers

Narrative

S.1.2 Pg.506 S.3.3 Pg.504[9] Pg.504[7] S.3.2 S.2,S.3.6 Pg.512[30] S.8

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

Formalisation

type
7 HI, LI
value
8a obs_HI: H → HI, obs_LI: L → LI
axiom
8b ∀ h,h′:H, l,l′:L •
h≠h′⇒obs_HI(h)≠obs_HI(h′) ∧ l≠l′⇒obs_LI(l)≠obs_LI(l′)

RSL Explanation

- 7: introduces two new sorts;
- 8a: introduces two new observer functions:
  - ⋆ → 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
- 8b: expresses the uniqueness of identifiers.

S.4.5
Pg.504[13]

End of RSL Explanation

"slide 1066"

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:

Mutual Hub and Link Referencing

Narrative

9 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.
10 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.
11 Extending Item 9: the observed hub identifiers must be identifiers of hubs of the net to which the link belongs.
12 Extending Item 10: 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, functions (iohs, iols) which calculate these sets.

PG.:1067

Formalisation

```
value
  9a  obs_HIs: L → HI-set,
  10a  obs_LIs: H → LI-set,
axiom
  9b  ∀ l:L • card obs_HIs(l)=2 ∧
  10b  ∀ h:H • card obs_LIs(h)≥1 ∧
  ∀ (hs,ls):N •
  9(a)      ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ obs_LIs(h) ⇒
              ∃ l′:L • l′ ∈ ls ∧ li=obs_LI(l′) ∧ obs_HI(h) ∈ obs_HIs(l′) ∧
  10(a)     ∀ l:L • l ∈ ls ⇒
              ∃ h′,h″:H • {h′,h″}⊆hs ∧ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}
  11  ∀ h:H • h ∈ hs ⇒ obs_LIs(h) ⊆ iols(ls)
  12  ∀ l:L • l ∈ ls ⇒ obs_HIs(h) ⊆ iohs(hs)
value
  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}
```

RSL Explanation

- 9a,10a: Two observer functions are introduced.
- 9b,10b: Universal quantification secure that all hubs and links have prerequisite number of unique (reference) identifiers.
  - ⋆ 9(a): We read ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ obs_LIs(h) ⇒ ∃ l′:L • l′ ∈ ls ∧ li=obs_LI(l′) ∧ obs_HI(h) ∈ obs_HIs(l′): For all hubs (h) of the net (∀h:H•h∈hs) it is the case (⇒) that for all link identifiers (li) of that hub (∀li:LI•li∈obs_LIs(h)) it is the case that there exists a link of the net (∃l′:L•l′∈ls) where that link's (l′'s) identifier is li and the identifier of h is observed in the link l′.
  - ⋆ 10(a): We read ∀ l:L • l ∈ ls ⇒ ∃ h′,h″:H • {h′,h″} ⊆hs ∧ obs_HIs(l) = {obs_HI(h′), obs_HI(h″)}: for all ... further reading is left as exercise to the reader.
- 11: Reading is left as exercise to the reader.
- 12: Reading is left as exercise to the reader.
- iohs,iols: These two lines define the signature: name and type of two functions.
- 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.
- 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 ≡: "the set of all obs_LI(l) for which (|) l is of type L and such that (•) l is in ls".

End of RSL Explanation

S.2.3
S.4.5
S.3.2
Pg.508
S.3.2
Pg.508

"SLIDE 1068"

F.3.2 Further Entity Properties

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. In Exercise 6.1 (Page 364) we shall introduce further attributes and other properties of hubs and links.[1]

"SLIDE 1069"

F.3.3 Entity Projections

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. "slide 1070"

Projection of Unique Identifiers

Narrative

13 Assume conceptual types of links and hubs such that such "pseudo" links and hubs can be compared for equality where the comparison does not include their own or their reference identifiers.
14 By a 'link (hub) identifier reset'
   (a) we understand a function, reset_I which applies to links or hubs, and results in a pseudo-link, respectively a pseudo-hub.
   (b) For any pseudo-link (pseudo-hub), reset_I applied to the result of applying restore_I to that pseudo-link (pseudo-hub) results in that pseudo-link (pseudo-hub).
15 By a 'link (hub) identifier restore'
   (a) we understand a function, restore_I which applies to pseudo-links or pseudo-hubs, and results in a link, respectively a hub.
   (b) For any link (hub), restore_I applied to the result of applying reset_I to that link (hub) results in that link (hub).
16 By an "other than identifier hub", respectively "... link", comparison', non_I_eq, we understand a predicate function which applies either to a pair of hubs or pseudo-hubs or to a pair of links or pseudo-links and yields truth

[1] Link attributes include link length, link name, link location, link units (where links can be seen as sequences of units, from one hub to another, or vice versa), etc. Hub attributes include hub name, hub location, etc. Link properties include that they are composite, consisting of units (See Exercise 6.2 Page 364).

value **true**, if the hubs or pseudo-hubs (links or pseudo-links) are "equal" except for their identifiers.
17 That is, a hub (link) is non_l_equal to its reset version.

PG.:1071

Formalisation

```
type
  13 pseudo_H, pseudo_L
value
  14(a)   reset_l: (H → pseudo_H) | (L → pseudo_L)
  15(a)   restore_l: (pseudo_H → H) | (pseudo_L → L)
axiom
  14(b),15(b)  ∀ h:H,l:L,ph:Pseudo_H,pl:Pseudo_H •
                 restore_l(reset_l(h))=h∧restore_l(reset_l(l))=l ∧
                 reset_l(restore_l(ph))=ph∧reset_l(restore_l(pl))=pl
value
  16  non_l_eq: ((H|pseudo_H)×(H|pseudo_H)→Bool)
            | ((L|pseudo_L)×(L|pseudo_L)→Bool)
axiom
  17  ∀ h:H,l:L • non_l_eq(h,reset_l(h)) ∧ non_l_eq(l,reset_l(l)) etc.
```

RSL Explanation

- 13: pseudo_H and pseudo_L are further undefined types.
- 14(a): The type union (|) expression (H → pseudo_H) | (L → pseudo_L) expresses that reset_l either applies to hubs or to links.
- 15(a): Similar to reset_l.
- 14(b),15(b),17: The axioms governing the pseudo_H and pseudo_L types and the reset_l and restore_l functions
- 16: As for predicates reset_l and restore_l (line 14(a), respectively line 15(a)), the type of the postulated non_l_eq predicate function is of union type.

End of RSL Explanation

F.4 Operations

"SLIDE 1072"

To illustrate the concept of operations[2] 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

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

(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 !

F.4.1 Syntax

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

Link Insertion and Removal

Narrative

18 To a net one can insert a new link in either of three ways:
   (a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;
   (b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;
   (c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.
   (d) From the inserted link one must be able to observe identifier of respective hubs.
19 From a net one can remove a link. The removal command specifies a link identifier.

PG.:1073

Formalisation

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

RSL Explanation

- 18: 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.
  - ⋆ 18(a): 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.
  - ⋆ 18(b): Reading of this type clause is left as exercise to the reader.
  - ⋆ 18(c): Reading of this type clause is left as exercise to the reader.
  - ⋆ 18(d): 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).
- 19; The type clause **type** Remove == Rmv(s_li:LI)
  - ⋆ (as for Items 18(b) and 18(c))
  - ⋆ 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

"SLIDE 1074"

F.4.2 Semantics

Then we consider the meaning of the Insert operation designators.

Semantic Well-formed of Insert Operations

Narrative

20 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.

21 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.

PG.:1075

1 If the command is of the form 2oldH(hi′,l,hi′) then
  ⋆1 hi′ must be the identifier of a hub in hs,
  ⋆s2 l must not be in ls and its identifier must (also) not be observable in ls, and
  ⋆3 hi″ must be the identifier of a(nother) hub in hs.
2 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.
3 If the command is of the form 2newH(h′,l,h″) then
  ⋆1 h′ — left to the reader as an exercise (see formalisation !),
  ⋆2 l — left to the reader as an exercise (see formalisation !), and
  ⋆3 h″ — left to the reader as an exercise (see formalisation !).

PG.:1076

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

```
value
  20  int_Insert: Insert → N ~→ N
  21′  pre_int_Insert: Ins → N → Bool
  21″  pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2     s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
       case op of
  1)       2oldH(hi′,l,hi″) → {hi′,hi″}∈ iohs(hs),
  2)       1oldH1newH(hi,l,h) →
                 hi ∈ iohs(hs) ∧ h∉ hs ∧ obs_HI(h)∉ iohs(hs),
```

3) 2newH(h′,l,h″) →
{h′,h″}∩ hs={} ∧ {obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
end

RSL Explanation

- 20: The value clause **value** int_Insert: Insert → N $\xrightarrow{\sim}$ N names a value, int_Insert, and defines its type to be Insert → N $\xrightarrow{\sim}$ N, that is, a partial function ($\xrightarrow{\sim}$) from Insert commands and nets (N) to nets.
(int_Insert is thus a function. What that function calculates will be defined later.)
- 21′: The predicate pre_int_Insert: Insert → N → **Bool** function (which is used in connection with int_Insert to assert semantic well-formedness) applies to Insert commands and nets and yield truth value **true** if the command can be meaningfully performed on the net state.
- 21″: 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:
- ⋆2: s_l(op)∉ls∧obs_LI(s_l(op))∉iols(ls), namely that the new link is not an existing net link and that its identifier is not already known.
  - ⋆ 1): If the Insert command is of kind 2oldH(hi′,l,hi″) then {hi′,hi″}∈ iohs(hs), that is, then the two distinct argument hub identifiers must not be in the set of known hub identifiers, i.e., of the existing hubs hs.
  - ⋆ 2): If the Insert command is of kind 1oldH1newH(hi,l,h) then ... exercise left as an exercises to the reader.
  - ⋆ 3): 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. S.3.6 on page 510 Item 23 on page 511.

End of RSL Explanation

"slide 1077"

Some Auxiliary Functions: Hub and Link "Extraction"

Narrative

22 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.
23 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.

Formalisation

value
22: xtr_H: HI → N $\xrightarrow{\sim}$ H
22: xtr_H(hi)(hs,_) ≡ **let** h:H•h ∈ hs ∧ obs_HI(h)=hi **in** h **end**

```
                    pre hi ∈ iohs(hs)
23: xtr_L: HI → N ~→ H
23: xtr_L(li)(_,ls) ≡ let l:L•l ∈ ls ∧ obs_LI(l)=li in l end
                    pre li ∈ iols(ls)
```

RSL Explanation

- 22: 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. Pg.520
- 22: The wild-card, _, expresses that the extraction (xtr_H) function does not need the **L-set** argument.
- 23: Left as an exercise for the reader.

End of RSL Explanation

"slide 1078"

Auxiliary Functions: Hub and Link Identifier "Updates"

Narrative

24 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.

25 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).

Formalisation

```
value
  aLI: H × LI → H, rLI: H × LI ~→ H
  24: aLI(h,li) as h′
        pre li ∉ obs_LIs(h)
        post obs_LIs(h′) = {li} ∪ obs_LIs(h) ∧ non_I_eq(h,h′)
  25: rLI(h′,li) as h
        pre li ∈ obs_LIs(h′) ∧ card obs_LIs(h′)≥2
        post obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_I_eq(h,h′)
```

RSL Explanation

- 24: The add link identifier function aLI:
  - ⋆ The function definition clause aLI(h,li) **as** h′ defines the application of aLI to a pair (h,li) to yield an update, h′ of h.
  - ⋆ The pre-condition **pre** li ∉ obs_LIs(h) expresses that the link identifier li must not be observable h.
  - ⋆ The post-condition **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_I_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.512[25]

- 25: 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

Semantics of the Insert Operation

Narrative

26 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}.

27 If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has
- 27.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.
- 27.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.
- 27.2 : The set of links augmented by the new link.

28 If the Insert command is of kind 2oldH(hi',l,hi") then
- 28.1–.2: the two connecting hubs are updated to reflect the new link,
- 28.3 : and the resulting sets of hubs and links updated.

PG.:1080

Formalisation

```
int_Insert(op)(hs,ls) ≡
⋆i  case op of
26      2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
27      1oldH1newH(hi,l,h) →
27.1        let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
27.2        (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
28      2oldH(hi′,l,hi″) →
28.1        let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
28.2                   aLI(xtr_H(hi″,hs),obs_LI(l))} in
28.3        (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** $p_1 \rightarrow c_1$, $p_2 \rightarrow c_2$, ... $p_n \rightarrow c_n$ **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.
- $\star_i$ + 26: If op is of the form 2newH(h′,l,h″ then — the narrative explains the rest;
  else
- $\star_i$ + 27: If op is of the form 1oldH1newH(hi,l,h) then
  - ⋆ 27.1: h′ is the known hub (identified by hi) updated to reflect the new link being connected to that hub,
  - ⋆ 27.2: 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
- $\star_i$ + 28: 28: If op is of the form 2oldH(hi′,l,hi″) then
  - ⋆ 28.1: the first element of the set of two hubs (hsδ) reflect one of the updated hubs,
  - ⋆ 28.2: the second element of the set of two hubs (hsδ) reflect the other of the updated hubs,
  - ⋆ 28.3: 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

"slide 1081"

Semantics of the Remove Operation

Narrative

29 The remove command is of the form Rmv(li) for some li.
30 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.

PG.:1082

Formalisation

```
value
  29  int_Remove: Rmv → N ~→ N
  30  int_Remove(Rmv(li))(hs,ls) ≡
  30(a)  let l = xtr_L(li)(ls), {hi′,hi″} = obs_Hls(l) in
  30(b)  let {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} in
  30(c)  let hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) in
  30(d)  (hs\{h′,h″} ∪ hs′,ls\{l}) end end end
  30(a)  pre li ∈ iols(ls)

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

RSL Explanation

- 29: 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.
- 30: To Remove a link identifier by li from the net (hs,ls) can be formalised as follows:
  - ⋆ 30(a): obtain the link l from its identifier li and the set of links ls, and
  - ⋆ 30(a): obtain the identifiers, {hi′,hi″}, of the two distinct hubs to which link l is connected;
  - ⋆ 30(b): then obtain the hubs {h′,h″} with these identifiers;
  - ⋆ 30(c): now examine cond_rmv each of these hubs (see Lines 30((c)i)–30((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).
    - ◦ 30(c) 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
  - ⋆ 30((c)i): either yields the empty set (of no hubs) if li is the only link identifier inh,
  - ⋆ 30((c)ii): or yields a modification of h in which the link identifier li is no longer observable.

End of RSL Explanation

F.5 Events

"SLIDE 1083"

F.5.1 Some General Comments

First we remind the reader of our definition of what constitutes an event (Sect. 1.12.3 on page 39).

This section shall be very brief. The reason is this: The concept of events and their description is very important. But examples of event descriptions are closely intertwined with examples of behaviour descriptions. We shall therefore postpone the illustration of serious event descriptions till Sect. F.7. After "slide 1084" some tiny examples of events and before example of behaviours we insert a section, Sect. F.6, a section which introduces some concepts, like time and time intervals, which are necessary to properly describe events and behaviours. "slide 1085"

But first we informally illustrate a number of event scenarios.

F.5.2 Transport Event Examples

(i) A link, for some reason "ceases to exist"; for example: a bridge link falls down, or a level road link is covered by a mud slide, or a road tunnel is afire, or a link is blocked by some vehicle accident. (ii) A vehicle enters or leaves the net. (iii) A hub is saturated with vehicles. "slide 1086"

Relating the above three sets of examples of events to out "formal" definition of an event we have these remarks: (i) The state is the transport net of hubs and links at the times observed. We can think of a "link ceasing to exist" as an instantaneous event, i.e., $t_a = t_b$, or as an event that occurs over some time, i.e., $t_a \geq t_b$ — in all cases $\sigma_a \neq \sigma_b$ (see the int_Remove function definition Pages 355–357 and our 'Net Behaviour' example Pages 362–364). "slide 1087"

(ii) The state is the traffic at the times, t_a, t_b, observed. We would say that $t_a = t_b$ but that $\sigma_a \neq \sigma_b$ (state σ_a is state σ_b "plus" or less the vehicle — provided we consider just one vehicle). (iii) The state is the traffic at the times observed. Times are different by a fraction: $t_a \geq t_b$. States are different. "slide 1088"

F.5.3 Banking Event Examples

(i) Withdrawal of funds from an account (i.e., a certain action) leads to either of two events: (i.A) either the remaining balance is above or equal to the credit limit, (i.B) or it is not. In the latter case that event may trigger a corrective action. (ii) A national (or federal) bank interest rate change is an action by the the national (or federal) bank, but is seen as an event by any local bank, and may cause such a bank to change (i.e., an action) its own interest rate. (iii) A local bank goes bankrupt.

We leave it to the reader to comment on the time and state relations for the above banking examples.

F.6 Some Fundamental Modelling Concepts "SLIDE 1089"

Before we illustrate formal examples of traffic **events** (Pages 364–364) we must formalise concepts of vehicle (Pages 361–362) and net (Pages 362–364) **behaviours**. But first we need introduce (and describe: narrate and formalise) some further entities: time (Pages 358–359), time intervals (Pages 359–359), link and hub positions (Pages 359–360), traffic (Pages 360–360) and various notions of traffic well-formedness (meaningful net positions: Pages 361–361, monotonic vehicle movements: Pages 361–362 and no erratic vehicle movements: Pages ??–??).

F.6.1 Time and Time Intervals

Time

Narrative

31 Time is here considered an ordered, infinite set of points
  (a) such that for each time there is a unique next time.
32 A proper subset of Time
  (a) is an ordered, possibly infinite set of Time points
  (b) such that there is a minimum, i.e., a smallest, or begin time point and a maximum, i.e., a largest, or an end time point and
  (c) such that for each time in the proper subset, other than the end point time, there is a unique next time.
33 Traffics are here considered to be discrete functions from a proper subset of Time to pairs of nets and positions of vehicles.
34 Positions of vehicles are discrete functions from vehicles to positions.

PG.:1091

Formalisation

```
type
  31 Time
value
  31(a) next_T: Time ~→ Time, pre pre_next_T(t)
  31(a) pre_next_T: T → Bool
  31(a) pre_next_T(t) ≡ ~is_end_T(t)
type
  32 PSoTime = {|tset:Time-set•wf_PSoTime(tset)|}
value
  32 wf_PSoTime: Time-set → Bool
  32 wf_PSoTime(ts) ≡
  32(a)  is_ordered(ts) ∧
  32(b)  ∃ t_begin,t_end:Time •
                {t_begin,t_end}⊆ts ∧
                ∀ t:Time • t ∈ ts ⇒ t_begin≤t≤t_end ∧
```

32(c) ∀ t:Time • t ∈ ts\{t_end} • next_T(t)∈ ts

type

33 V, P [to be defined later, see Items 42–42]
[and Items 43(a)–43(b), Page 359]

33 TF = Time $\xrightarrow[m]{}$ N × VP

34 VP = V $\xrightarrow[m]{}$ P

"slide 1092"

Time Intervals

Narrative

35 A time interval is a finite passage of time.
36 One cannot add two times, but one can subtract an earlier time from a later time and obtain a time interval.
37 One can add (and subtract) two time intervals and obtain a time interval.
38 One can multiply a real with a time intervals and obtain a time interval.
39 One can divide a time interval by another time interval and obtain a real.
40 One can divide a time interval by a real and obtain a time interval.
41 One can compare pairs of times and pairs of time intervals for smaller than or equal, smaller than, equality, inequality, larger than, or larger than or equal.

We do not specify these operations.

Formalisation

type

35 TI

value

36 −: Time × Time → TI
37 ∗: **Real** × TI → TI
38 +,−: TI × TI → TI
39 /: TI × TI → **Real**
40 /: TI × **Real** → TI
41 ≤, <, =, ≠, >, ≥: (Time×Time)|(TI×TI) → **Bool**

We thus use the overloaded operators −, ∗, /, ≤, <, =, ≠, >, ≥ also for time related functions.

"slide 1093"

F.6.2 Vehicles and Hub and Link Positions

Vehicles and Hub and Link Positions

Narrative

42 There are vehicles, and vehicles are further undefined.
43 There are positions, and a position is either on a link or in a hub.

(a) A hub position is indicated just by a triple: the identifier of the hub in question, and a pair of (from and to) link identifiers, namely of links connected to the identified hub.
(b) A link position is identified by a quadruplet: The identifier of the link, a pair of hub identifiers (of the link connected hubs), designating a direction, and a real number, properly between 0 and 1, denoting the relative offset from the from hub to the to hub.

PG.:1094

Formalisation

type
42 V
43 P = HP | LP
43(a) HP == hpos(s_hi:HI,s_fli:LI,s_tli:LI)
43(b) LP == lpos(s_li:HI,s_fhi:LI,s_tli:LI,s_offset:Frac)
43(b) Frac = {|r:**Real**•0<r<1|}

F.7 Behaviours

"SLIDE 1095"

F.7.1 Traffic as a Behaviour

Traffic

Narrative

44 Traffic is a discrete function from a 'Proper subset of Time' to pairs of nets and vehicle positions.
45 Vehicles positions is a discrete function from vehicles to vehicle positions.

We shall have much to say, later, on the well-formedness of traffics.

Formalisation

type
44 TF = PSoTime $\overrightarrow{m}$ (N × VehPos′)
45 VehPos′ = V $\overrightarrow{m}$ P
axiom

RSL Explanation

Pg.504[12]

- 44 $\overrightarrow{m}$ is an infix type operator. Applied to types PSoTime and VehPos′ it constructs the traffic type of all discrete maps (i.e., function) from values of type PSoTime to values of type VehPos′.
- 45 As for 44.

End of RSL Explanation

"slide 1096"

Traffic: Well-formedness, I, Positions

Narrative

46 All positions recorded in traffics must be positions of the net of the traffic.

Formalisation

```
value
  46 wf_TFc: TF → Bool
     wf_TFc(tf) ≡
        ∀ ((hs,ls),vp):(N × VehPos′) • ((hs,ls),vp) ∈ rng tf ⇒
           ∀ p:P • p ∈ rng vp ⇒
              case p of
                 hpos(li′,hi,li″) →
                    hi ∈ iohs(hs) ∧
                    let h=xtr_H(hi,hs) in {li′,li″}⊆obs_HIs(h) end,
                 lpos(hi′,li,hi″,f) →
                    li ∈ iols(ls) ∧
                    let l=xtr_L(li,ls) in {hi′,hi″}=obs_LIs(l) end end
```

"slide 1097"

Traffic: Well-formedness, II, Monotonicity

Narrative

47 A traffic must satisfy the following well-formedness properties:
   (a) If a vehicle is in the traffic at times t' and t'' then it is also in the traffic at all times, t, between t' and t''.
   (b) If a vehicle is in the traffic at time t and at position p and at time $next\_T(t)$, then its position at time $next\_T(t)$ is $next\_P(p)$ where $next\_P(p)$ is
      i. either p (the vehicle has either not moved along a link, or is still at a hub),
      ii. or if p is a link position, lpos(l_i, fh_j, th_k, f), then $next\_P(p)$
         A. is either a hub position (hpos(l_i, h_j, l_k)) provided f is infinitisimally [or just very, very] close to 1,
         B. or is a link position lpos(l_i, fh_j, th_k, f') where f' is $f + \delta_f, f + \delta_f < 1$ where δ_f is a positive, very small real between 0 and 1. (That is: the vehicle has moved, but just a little bit.)
      iii. or if p is a hub position, hpos(l_i, h_j, l_k, f), then $next\_P(p)$
         A. is either the same position p,
         B. or is a link position lpos$(h_j, l_k, h_\ell, \delta_f)$ for the other hub identifier, h_ℓ, of the link identified by l_k.
   (c) A vehicle behaviour, during some time interval, can be seen
      i. either as a "degenerated" traffic of only one vehicle,
      ii. or a a sequence of that vehicle's positions.

It follows from the above that vehicles cannot change direction of movement. We can relax this constraint, but will not do so here.

PG.:1098

Formalisation

```
value
        v_in_tf_at_time: V × TF × Time → Bool
        v_in_tf_at_time(v,(_,tvp),t) ≡ t ∈ dom tf ∧ v ∈ dom tf(t)
  47  wf_TFa: (PSoTime →m VehPos) → Bool
      wf_TFa(tvp) ≡
  47(a)  ∀ t,t′:Time • {t,t′} ⊆ dom tvp ⇒
           ∀ v:V • v_in_tf_at_time(v,tvp,t) ∧ v_in_tf_at_time(v,tvp,t) ⇒
             ∀ t″:Time • t<t″<t′ ⇒ v_in_tf_at_time(v,tvp,t″)
```

PG.:1099

```
value
  44  tf:TF, v:V, ft,tt:Time
axiom
  44  is_in_TF(v)(ft,tt)(tf)
value
  ⋆1  is_in_TF: V → (Time×Time) → TF → Bool
  ⋆1  is_in_TF(v)(ft,tt)(tf) ≡
  ⋆2    {ft,tt}⊆dom tf ∧
  ⋆3    assert {ft,tt}⊆dom tf ⇒ ∀ t:Time•ft<t<tt⇒t ∈ dom tf
  ⋆4    ∀ t:Time•ft<t<tt ⇒ v ∈ dom tf(t)

  47((c)i)  VehBeh_tf: V → (Time×Time) → TF
        VehBeh_tf(v)(ft,tt)(tf) ≡ [ t↦[ v↦((tf)(t))(v) ]|t:Time•ft<t<tt ]
          pre is_in_TF(v)(ft,tt)(tf)

  47((c)ii)  VehBeh_seq: V → (Time×Time) → P*
        VehBeh_seq(v)(ft,tt)(tf) ≡ ⟨((tf)(t))(v)|t:Time•t in {ft..tt}⟩
          pre is_in_TF(v)(ft,tt)(tf)
```

"slide 1100"

F.7.2 A Net Behaviour

A Net Behaviour

Narrative

48 Nets constantly undergo changes:
   (a) New links are properly inserted.
   (b) Old links are properly removed.

(c) Links "suddenly" ceases to "function", i.e., appears as having been (improperly) removed.

PG.:1101

Formalisation

```
value
  n:N
variable
  net:N := n
type
  Road_Event == L_Ev(s_li:LI)|...
channel
  rch:(Insert|Remove)
  ech:Road_Event
value
  system: Unit → Unit
  dept_of_publ_works: Unit → out ch  Unit
  road_net: Unit → in rch  Unit
  net_events: Unit → out ech  Unit

  system() ≡ dept_of_publ_works() ‖ road_net() ‖ net_events()
```

PG.:1102

```
  dept_of_publ_works() ≡
    (.. ⫿
     let ins:Insert • pre_int_Insert(ins)(c net) in
     ins!ch end
     ⫿
     let rem:Remove • pre_int_Remove(rem)(c net) in
     ins!ch end
     ⫿ ...); dept_of_publ_works()

  net_events() ≡
    (skip ⫿
     let Link_Event(li):Road_Event • li ∈ iols(c net) in
     Link_Event(li)!evc end); net_events()
```

PG.:1103

```
  road_net() ≡
    (... []
     let cmd = (rch?‖ech?) in
     case cmd of
48(a)     Ins(ins) → net := int_Insert(cmd)(c net),
48(b)     Rmv(li) → net := int_Remove(Rmv(li))(c net),
```

```
48(c)   L_Ev(li) → net := int_Remove(Rmv(li))(c net),
        ... → ...
    end end); road_net()
```

We model the net behaviour in terms of a system of concurrent behaviours: that of a public works department which non-deterministically (⌈⌉) issues orders to insert or remove links; that of a net events behaviour which non-deterministically either does nothing or "signals" an appropriate breakdown of a link; and that of the road net behaviour which (external) deterministically reacts to the insert or remove orders or to the link breakdown.

Channels connect these subsidiary, recursive behaviours. A global variable represents the net. It is initialised to some net.

F.8 Traffic Events

"slide 1104"

"slide 1105"

F.9 Review

"slide 1106"

F.10 Exercises

Exercise 37. Link and Hub Attributes: We extend the attribites of links and hubs.

37.1 Links have names, locations and lengths. We do not further define a notion of 'location'.
37.2 At most two links connected to the same hub may have the same link name.
37.3 If two links of a net have the same link name then there is a sequence of links of which these two links are the first and the last and such that intermediate links connect via hubs to these. ("Figure that one out !")
37.4 A sequence of identically named links of a net is said to form a corridor (thus of one or more links). Thus define a predicate which tests whether a sequence of links are [hub] connected and are identically named.
37.5 Define a function which yields all corridors of a net.
37.5 Define a function which calculates the length of a corridor — where "lengths" of connecting hubs are assumed 0 !
37.6 A corridor which is not a proper sub-corridor of another is called a line. Define a function which yields all lines of a net.

Solution 37 Vol. II, Page 536, suggests a way of answering this exercise.

Exercise 38. Link Units: We now consider links to be composite entities.

38.1 There are units and units have unique unit identfiers, lengths and locations.

38.2 Links consists of units.
38.3 No two links of a net share units, i.e., all units of a net are distinct.
38.4 Units also "record" the link of which they are units.
38.5 The units of links form sequences, one sequence when the link is observed from one of its connecting hub identifiers to the other, the reverse sequence when seen "the other way around" !
38.6 We leave it to the reader to define 'reverse'.

Solution 38 Vol. II, Page 536, suggests a way of answering this exercise.

"SLIDE 1107"

"slide 1108"

G

Support Technologies

"slide 1109"

In this Appendix we shall exemplify three sets of support technologies: hub signalling (like semaphores), road-rail gates, sensors, signals, and rail switching.

G.1 Net Signalling

"slide 1110"

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. 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 sempahore lamps. But, eventually, one has to, all of it, as part of domain modelling. "slide 1111"

G.1.1 Intrinsic Concepts of States

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

Narrative

Link and Hub States

We claim that the concept of hub and link states is an intrinsics facet of transport nets. We now introduce the notions of hub and link states and state spaces and hub and link state changing operations. A hub (link) state is the "slide 1112" 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. "slide 1113"

Link and Hub State Spaces and State-change Designators

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
"slide 1114" a possibly new hub state (the link and a possibly new link state).

Formalisation

States

Syntactic Well-formedness Functions:

type
  $L\Sigma' = L\_Trav$-**set**
  $L\_Trav = (HI \times LI \times HI)$
  $L\Sigma = \{|\ lnk\sigma{:}L\Sigma' \bullet syn\_wf\_L\Sigma\{lnk\sigma\}\ |\}$
value
  $syn\_wf\_L\Sigma: L\Sigma' \to$ **Bool**
  $syn\_wf\_L\Sigma(lnk\sigma) \equiv$
    $\forall\ (hi',li,hi''),(hi''',li',hi''''){:}L\_Trav \bullet \Rightarrow$
      $(\{(hi',li,hi''),(hi''',li',hi'''')\} \in lnk\sigma \Rightarrow li = li' \wedge$
      $hi' \neq hi'' \wedge hi''' \neq hi'''' \wedge \{hi',hi''\} = \{hi''',hi''''\})$

"slide 1115"

type
  $H\Sigma' = H\_Trav$-**set**
  $H\_Trav = (LI \times HI \times LI)$
  $H\Sigma = \{|\ hub\sigma{:}H\Sigma' \bullet wf\_H\Sigma\{hub\sigma\}\ |\}$
value
  $syn\_wf\_H\Sigma: H\Sigma' \to$ **Bool**
  $syn\_wf\_H\Sigma(hub\sigma) \equiv$
    $\forall\ (li',hi,li''),(li''',hi',li''''){:}H\_Trav \bullet$
      $\{(li',hi,li''),(li''',hi',li'''')\} \subseteq hub\sigma \Rightarrow hi = hi'$

"slide 1116"

Syntactic and Semantic Well-formedness Functions: 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
"slide 1117" existing links and that hub identifiers designate existing hub.

Semantic Well-formedness Functions:

value
  $sem\_wf\_L\Sigma: L\Sigma \to N \to$ **Bool**
  $sem\_wf\_H\Sigma: H\Sigma \to N \to$ **Bool**

  $sem\_wf\_L\Sigma(lnk\sigma)(ls,hs) \equiv lnk\sigma \neq \{\} \Rightarrow$

```
    ∀ (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)
```

"slide 1118"

Auxiliary Functions:

```
value
  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≠{}
```

"slide 1119"

```
  xtr_LI: LΣ → LI
  xtr_LIs(lnkσ) ≡
    let (hi,li,hi′):L_Trav • (hi,li,hi′) ∈ hubσ in li end
  pre: lnkσ≠{}

  xtr_HI: LΣ → HI-set
  xtr_HI(lnkσ) as his
  pre: lnkσ≠{}
  post his={hi,hi′|(hi,li,hi′):L_Trav • (hi,li,hi′) ∈ hubσ}∧ card his=2
```

"slide 1120"

State Spaces

```
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

"slide 1121"

G.1.2 A Support Technology Concept of States

Narrative (I)

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.

"slide 1122"

Formalisation (I)

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$'$

"slide 1123"

Narrative (II)

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,[1] before the final hub state ($h\sigma_f$) is set. "slide 1124"

Formalisation (II)

```
type
  TI [ time interval ]
  Signalling = (HΣ × TI)*
  Sema = (HΣ × HΣ) 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
  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
```

"slide 1125"

G.1.3 Discussion

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 of when to start and end semaphore switching, etcetera, must all be detailed. "slide 1126"

For this one will undoubtedly need use other formalisms than the ones mainly used in this book, for example: Message and Live Sequence Charts, MSCs and LSCs [118, 119, 120] and [63, 102, 126], Petri nets [124, 173, 185, 184, 186], Statecharts [98, 99, 101, 103, 100], SCs, Duration Calculus [228, 229], DC, Temporal Logic of Actions [131, 132, 155, 156], TLA+, Temporal Logic of Reactive Systems [74, 149, 150, 165, 180], STeP, etcetera

In the next example, a road-rail level crossing, Sect. G.2, we shall illustrate the use of a temporal logic, DC. The last example, a rail switch, Sect. G.3, we shall, however briefly, hint at probabilistic behaviours of support technologies.

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

G.2 Road-Rail Level Crossing

"SLIDE 1127"

The presentation of this section (i.e., Sect. G.2) follows that of [200, Skakkebæk et al., 1992].

We have chosen a rather large example, but we will present it in parts. In this way the reader can read the first example, or the first two, and so on. The aim of bringing in the examples is, besides the main one of showing aspects of the supporting technologies facet, also to illustrate some aspects of the Duration Calculus [228, 229] specification language.[2] We refer to Fig. G.1 for a "picture" of a road-rail level crossing.

"slide 1128"

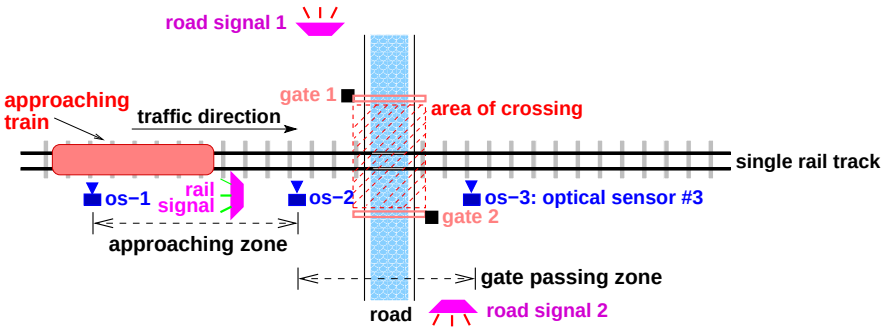


Fig. G.1. A road-rail level hub (with adjoining links)

"slide 1129"

G.2.1 An Intrinsic Concept of Road-Rail Level State

We have already introduced an intrinsic notion of a hub state, namely as that of a set of triples

$$\{(li_a, hi_k, li_b), (li_c, hi_k, li_d), \ldots, (li_e, hi_k, li_f)\}$$

of link, hub and link identifiers — where the hub identifiers in that set are the same. We refer to Sect. G.1.1. We take that model of a hub state to be interpreted as follows: The hub is the road-rail crossing area of Fig. G.1; the four links are the rail tracks and the road segments on either side of the hub.

"slide 1130"

We identify the hub by hi, and the four links by li1, li2, li3 and li4. Cf. Fig. G.2 on the facing page.

The intended intrinsic hub state space is:

"slide 1131"

- {{(li1,hi,li2),(li2,hi,li1)},{(li3,hi,li4)}}

[2] We are grateful to Dr. Jens Ulrik Skakkebæk and to Profs. Anders Peter Ravn and Hans Rischel (and the publisher, the IEEE Computer Science Press) for permission to bring in the extensive, albeit substantially edited examples.

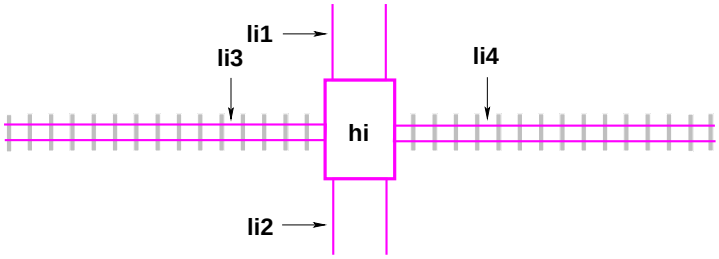


Fig. G.2. Road-rail level hub and link identifiers

G.2.2 A Concrete Concept of Road-Rail Level State

We refer to Fig. G.1 on the preceding page. The concrete state we have in mind for this road-rail level crossing consists of the following state components: train position ($\mathsf{os}_i, i : 1, 2, 3$), rail signal (signal), two road signals (traffic) and two gates (seen as one, gates). We shall later formalise these. "slide 1132"

G.2.3 Overview

We shall "chop" our presentation of the road-rail crossing example into five parts: Sect. G.2.4 on the following page deals with the function and safety expectations that one should have from a properly functioning road-rail level crossing system. The next three examples deal with various assumptions about the domain of road traffic, rail traffic, i.e., trains, and the optical/mechanical [support technology] devices that are to assist the road-rail level crossing system in achieving required safety and functionality. Section G.2.5 on page 378 deals with assumptions about road traffic. Section G.2.6 on page 379 deals deals with assumptions about train traffic, and Sect. G.2.7 deals with assumptions about the devices. Finally Sect. G.2.8 outlines the monitoring and control strategy for the computing system, i.e., the machine design.

"SLIDE 1133"

The set of five examples, Sects. G.2.4–G.2.8, (i) function and safety, (ii) road traffic, (iii) train traffic, (iv) device characteristics, (v) and software design, may be interpreted as also exemplifying phases and stages of development: (i) requirements engineering; (ii–iv) domain engineering: (ii) road traffic, (iii) train traffic and (iv) device characteristics; (v) and software design. Be that as it may, we can interpret all the five examples as reflecting a domain which already has all the specified aspects of the road-rail level crossing, including the software. "slide 1134"

G.2.4 Function and Safety

Narrative

The problem is to describe the *function* and the *safety* of an optical-mechanical traffic system. The problem, in this example, is not to specify how to achieve *function* and *safety*, but only to specify what we mean by *function* and *safety*. Thus the problem is more a domain and a requirements specification than a computing systems design problem.

Consider a road-rail level crossing (Fig. G.1). All dimensions are rather "out of scale". The road-rail level crossing is for a single track rail with all trains passing only in one direction (left to right on the figure). Many factors
"slide 1135" determine the monitoring and control of road and rail traffic.

(i) Road traffic is controlled by gates, one on either side of the track.

(ii) The gates close only when road traffic is not "stuck" in the crossing area (shown dashed).

(iii) Road traffic is advised of approaching and crossing trains by road signals, one on either side of the track. When the gates are to be lowered these road signals are set to red (STOP). When the gates have been fully raised the road signals are set to green GO.

(iv) Train traffic is controlled (i.e., advised) by a rail signal on the right side of the track of approaching trains, well before the crossing area.

(v) The rail signal indicates either STOP (red) or GO (green) for oncoming
"slide 1136" (i.e., approaching) trains.

(vi) Optical sensors (os) monitor trains in the vicinity of the crossing area.

(vii) A sensor, os_1, is placed at a reasonable distance from the rail signal such that a train will reach this, the first sensor before it reaches the rail signal.

(viii) A train enters the system whenever it is so determined by sensor os_1.

(ix) A train has left the crossing whenever sensor os_3 determines that the rearend of the train has passed the crossing.

(x) When a train approaches the gates are to be closed — provided there
"slide 1137" is no traffic "stuck" in the crossing area.

(xi) The rail signal is (to be) set to GO after the gates have closed.

(xii) When no trains are approaching or passing, the rail signal must be set to STOP and the gates are to be opened.

The main goal of the combination of optics and mechanics with a computing system monitoring the traffic and controlling the gates and the signal is to ensure *safety:*

(xiii) The complete system (optics, mechanics, computing) must never allow road and train traffic to pass the crossing area at the same time.

(xiv) Furthermore, the system must ensure that both road and rail traffic are able to pass the crossing area within some reasonable time.

"slide 1138" (xv) A train is passing whenever it is between sensors os_2 and os_3.

Formalisation

Let us refer to the required system as the Road-Rail Level Crossing System: $\text{R}^2\ell\text{CS}$.

The $\text{R}^2\ell\text{CS}$ accepts inputs from the optical and the gate sensors, and offers output (commands) to the signals and the gates.

State Variables

The *state* consists of a number of variables: (a) one for the *(rail) signal*, (b) one for the two *gates*, (c) one for the road traffic and (d) one for the rail traffic. "slide 1139"

```
type
  Rail_Signal == stop | go
  Gates == opening | open | closing | closed
  Road_Traffic == stopped | stuck_in_cross | free_to_cross
variable
  signal:Rail_Signal
  gates:Gates
  traffic:Road_Traffic
```

Trains are either *approach*ing or *pass*ing.

```
variable
  approach: Nat-set
  pass: Nat-set
```

"slide 1140"

That is, a train is identified by a unique, natural number, i. If some part of train i is between the first two sensors (os_1-os_2), then train i is *approaching*, i.e.,

```
  approach := {i} ∪ approach ;
```

And, if some part of train i is between the last two sensors (os_2-os_3), then train i is *passing*, i.e.:

```
  pass := {i} ∪ pass ;
```

"slide 1141"

Trains are *active* (wrt. crossing) if either approaching or passing (or both). One can define three *state assertions* concerning the state of trains:

```
value
  passing: Unit → Bool
  passing() ≡ pass ≠ {}

  approaching: Unit → Bool
  approaching() ≡ approach ≠ {}
```

active: **Unit** → **Bool**
active() ≡ (approach ∪ pass) ≠ {}

"slide 1142"

Properties

Now we are ready to express possible domain properties:

Prop ≡ □(SafeProp∧FunProp$_1$∧FunProp$_2$∧FunProp$_3$)

DC Explanation

- The Duration Calculus expression □ P, where P is an assertion over states, expresses:
  - ⋆ that P **always** holds,
  - ⋆ that is, at any time from now on.

End of DC Explanation

It turns out that we can express the functional requirements in terms of three state assertions.

Safety Properties: If the gates are not closed or road traffic is "stuck" in the crossing, then the train must not pass:

SafeProp ≡ ⌈((gates≠closed)∨(traffic=stuck))⌉ ⇒ ⌈∼passing()⌉

DC Explanation

- The Duration Calculus expression ⌈P⌉ expresses:
  - ⋆ the state assertion P holds during some time interval.
  - ⋆ ⌈(Q∨R)⌉⇒⌈S⌉ reads:
    - · during the interval (or duration) in which Q or R holds
    - · S also holds.

End of DC Explanation

"slide 1143"

Function Properties: There are three function requirements:

1 FunProp$_1$: The road traffic should maximally be held back for a predefined period of time t_{stop}:

FunProp$_1$ ≡ ⌈traffic=stopped⌉ ⇒ $\ell \leq t_{\text{stop}}$

DC Explanation
- The Duration Calculus expression ℓ stands for a time interval.
- The Duration Calculus expression ⌈P⌉⇒$\ell \leq t$ thus reads:
  - ⋆ ℓ refers to the duration during which ⌈P⌉ holds;
  - ⋆ that duration must, in this case, be less than or equal to t.

End of DC Explanation

2 $\mathsf{FunProp}_2$: When all trains have left the crossing, the gates must be open for at least time t_{open}:

$$\text{FunProp}_2 \equiv \lceil \text{active}() \rceil ; \lceil \sim \text{active}() \rceil ; \lceil \text{active}() \rceil \Rightarrow \int(\text{gates}{=}\text{open}) > t_{\text{open}}$$

DC Explanation

- The Duration Calculus expression $\lceil \sim \mathsf{P} \rceil$; $\lceil \mathsf{P} \rceil$; $\lceil \mathsf{P} \rceil$ reads:
  - ⋆ A duration during which P holds is followed (";") by
  - ⋆ a duration during which P dos not hold; which is then followed by
  - ⋆ a duration during which P again holds.
- The Duration Calculus expression $\lceil \mathsf{Q} \rceil \Rightarrow \int(\mathsf{R}) > t_{\mathsf{given}}$ reads:
  - ⋆ Q holding during a time interval implies
  - ⋆ that the summing of the time, during that time interval, of when R holds
  - ⋆ is larger than some given time.

End of DC Explanation

3 $\mathsf{FunProp}_3$: Provided the road traffic is not stuck, a single train must be able to pass within time t_{active}:

$$\text{FunProp}_3 \equiv \lceil \text{i} \in \text{approach} \cup \text{pass} \wedge (\text{traffic} \neq \text{stuck}) \rceil \Rightarrow \ell \leq t_{\text{active}}$$

"slide 1144"

What is Next ?

Section G.2.4 illustrated principles and techniques of prescribing requirements, as they were decomposed into those of safety and those of functionality.

In the next three examples, Example G.2.5 on the next page to Example G.2.7 on page 379. respectively, we "go backwards", as it were, to record the assumptions that any (later) design must (usually) make. That is, we describe (some facets of) the (application) domain. Normally, according to our "dogma", we first establish a domain description, before we, as we have just done, produce a requirements prescription, and, certainly long before we develop a software design specification. The design for the present problem domain of railway level crossings is recorded in Example G.2.8. "slide 1145"

We somewhat arbitrarily, it may seem, but pragmatically this is very sound, decompose the domain description into three parts: Describing the *road traffic*, i.e., Domain_1, describing the *train traffic*, i.e., Domain_2, and describing the supporting technology, i.e., the *device technology*, i.e., Domain_3. The relevant domain "theory" is the conjunction of these:

$$\text{Domain} \equiv \Box \bigwedge_{i=1}^{3} \text{Domain}_i$$

DC Explanation

- The conjunction predicate (and Duration Calculus) expression $\bigwedge_{i=1}^{n} P_i$ expresses the same as the conjunction of the P_i expressions: $P_1 \wedge P_2 \wedge \ldots \wedge P_n$.

End of DC Explanation

"slide 1146"

G.2.5 The Road Traffic Domain

We continue the railway level crossing example based on [200].

When running freely, i.e., without control, that is, without proper road signaling and gate control, the road traffic may eventually either stop properly in front of the gates, or get stuck in the crossing. Such stopped or stuck road traffic may subsequently become free, i.e., neither stopped nor stuck:

RoadTrafficAssump$_1$ ≡
a. (⌈traffic=stopped⌉→⌈traffic=free_to_cross⌉)
b. ∧ (⌈traffic=free_to_cross⌉→(⌈traffic=stopped⌉∨⌈traffic=stuck_in_cross⌉))
c. ∧ (⌈traffic=stuck_in_cross⌉→⌈traffic=free_to_cross⌉)

DC Explanation

- The Duration Calculus expression ⌈P⌉→⌈Q⌉ (or ⌈P⌉→(⌈Q⌉ ∨ ⌈R⌉)) expresses:
  - ⋆ after a duration during which P holds
  - ⋆ follows a duration during which Q holds
  - ⋆ (or a duration during which Q or a duration during which R holds).

a. (⌈traffic=stopped⌉→⌈traffic=free_to_cross⌉) reads: a time duration during which road traffic was stopped leads to a time duration during which road traffic is free to cross.
b. (⌈traffic=free_to_cross⌉→(⌈traffic=stopped⌉∨⌈traffic=stuck_in_cross⌉)) reads: a time duration during which road traffic was free to cross leads to a time duration during which the road traffic was either stopped or was stuck in the crossing area.
c. (⌈traffic=stuck_in_cross⌉→⌈traffic=free_to_cross⌉) reads: a time duration during which road traffic was stuck in the crossing area leads to a time duration during which the road traffic is free to cross.

End of DC Explanation

Road traffic is stopped iff the gates are not open:

RoadTrafficAssump$_2$ ≡ ⌈traffic=stopped⌉≡⌈gates≠open⌉

In closing, we record:

$$\text{Domain}_1 \equiv \Box \bigwedge_{i=1}^{2} \text{RoadTrafficAssump}_i$$

"slide 1147"

G.2.6 The Train Traffic Domain

We continue the railway level crossing example based on [200].

Trains must only pass if the rail signal is set to GO:

$$\text{TrainTrafficAssump}_1 \equiv \lceil \text{passing}() \rceil \Rightarrow \lceil \text{signal} \rceil$$

An active train travels in one direction only, i.e., initially approaches and finally passes:

$$\begin{aligned}&\text{TrainTrafficAssump}_2 \equiv \\ &\quad \lceil \text{i} \notin \text{approach} \cup \text{pass} \rceil \rightarrow \lceil \text{i} \in \text{approach} \wedge \text{i} \notin \text{pass} \rceil \\ &\quad \wedge \lceil \text{i} \in \text{approach} \wedge \text{i} \notin \text{pass} \rceil \rightarrow \lceil \text{i} \in \text{pass} \rceil \\ &\quad \wedge \lceil \text{i} \in \text{pass} \rceil \rightarrow \lceil \text{i} \notin \text{active} \rceil\end{aligned}$$

"slide 1148"

The last train in a series of trains passes the crossing before leaving the crossing:

$$\begin{aligned}&\text{TrainTrafficAssump}_3 \equiv \\ &\quad (\lceil \sim\text{active}() \rceil \rightarrow \lceil \text{approaching}() \wedge \sim\text{passing}() \rceil) \\ &\quad \wedge (\lceil \text{approaching}() \wedge \sim\text{passing}() \rceil \rightarrow \lceil \text{passing}() \rceil) \\ &\quad \wedge (\lceil \text{passing}() \rceil \rightarrow (\lceil \sim\text{active}() \rceil \vee \lceil \text{active}() \rceil)) \\ &\quad \wedge (\lceil \text{active}() \rceil \rightarrow \lceil \text{passing}() \rceil)\end{aligned}$$

The trains do not hesitate when the rail signal is GO:

$$\text{TrainTrafficAssump}_4 \equiv \lceil \text{signal=go} \wedge \text{active}() \rceil \Rightarrow \ell \leq \text{T}_{\text{sched}}$$

"slide 1149"

The railway lines are not overloaded with trains:

$$\begin{aligned}&\text{TrainTrafficAssump}_5 \equiv \\ &\quad \lceil \text{active}() \rceil ; \lceil \sim\text{active}() \rceil ; \lceil \text{active}() \rceil \\ &\qquad \Rightarrow \ell > \text{T}_{\text{inactive}} + \text{T}_{\text{wait}} + \text{T}_{\text{gate\_open}} + \text{T}_{\text{open}}\end{aligned}$$

Assumptions 1, 2 and 4 are really just obvious domain facts.

In closing, we record:

$$\text{Domain}_2 \equiv \Box \bigwedge_{i=1}^{5} \text{TrainTrafficAssump}_i$$

"slide 1150"

G.2.7 The Device Domain

We continue the railway level crossing example based on [200].

It takes, at most, time $T_{\text{gate\_close}}$ for the gates to close if the road traffic is not stuck in the crossing:

$$\begin{aligned}&\text{DeviceAssump}_1 \equiv \\ &\quad \lceil \text{gates=closing} \wedge \text{traffic} \neq \text{stuck\_in\_cross} \rceil \Rightarrow \ell \leq T_{\text{gate\_close}}\end{aligned}$$

It takes, at most, time $T_{\text{gate\_open}}$ for the gates to open:

$$\text{DeviceAssump}_2 \equiv \lceil\text{gates=opening}\rceil \Rightarrow \ell \leq T_{\text{gate\_open}}$$

"slide 1151"

The physical properties of *Gates* constrain the value of *gates* to cycle: open, closing, closed, opening, open, ... (in that order):

$$\begin{aligned}\text{DeviceAssump}_3 \equiv\ & (\lceil\text{gates=open}\rceil \rightarrow \lceil\text{gates=closing}\rceil) \\ & \wedge (\lceil\text{gates=closing}\rceil \rightarrow \lceil\text{gates=closed}\rceil) \\ & \wedge (\lceil\text{gates=closed}\rceil \rightarrow \lceil\text{gates=opening}\rceil) \\ & \wedge (\lceil\text{gates=opening}\rceil \rightarrow \lceil\text{gates=open}\rceil)\end{aligned}$$

The rail signal switches between STOP and GO:

$$\text{DeviceAssump}_4 \equiv (\lceil\text{signal=go}\rceil \rightarrow \lceil\text{signal=}\mathbf{\text{stop}}\rceil) \wedge (\lceil\text{signal=}\mathbf{\text{stop}}\rceil \rightarrow \lceil\text{signal=go}\rceil)$$

In closing, we record:

$$\text{Domain}_3 \equiv \Box \bigwedge_{i=1}^{4} \text{DeviceAssump}_i$$

"slide 1152"

Finally we are ready to record the design decisions.

G.2.8 The Software Design

We continue the railway level crossing example based on [200].

The software design was chosen by the system designers (Skakkebæk, Ravn and Rischel) to facilitate a proof of correctness with respect to the requirements and the assumptions (i.e., the domain).

The design decisions now presented are a formalisation of a finite state control, one that cycles through phases with inactive, approaching and passing trains. The overall design specification predicate is:

$$\text{Design} \equiv \Box(\bigwedge_{i=1}^{3} \text{ApproachTrains}_i \wedge \bigwedge_{j=1}^{4} \text{PassingTrains}_j)$$

"slide 1153"

Approaching Trains

The gates will remain open when no trains are present:

$$\text{ApproachTrains}_1 \equiv (\lceil\sim\text{active}()\rceil \wedge (\lceil\text{gates=open}\rceil \ ; \mathbf{\text{true}})) \Rightarrow \lceil\text{gates=open}\rceil$$

If trains are present, then the gates are open for at most T_{react}:

$$\text{ApproachTrains}_2 \equiv \lceil \text{gates=open} \wedge \text{active}() \rceil \Rightarrow \ell \leq T_{\text{react}}$$

It takes, at most, T_{nts} before the rail signal is GO when the gates have closed:

$$\text{ApproachTrains}_3 \equiv \lceil \text{signal=stop} \wedge \text{active}() \rceil \Rightarrow \ell \leq T_{\text{nts}}$$

"slide 1154"

Passing Trains

The gates remain closed as long as the rail signal is GO:

$$\text{PassingTrains}_1 \equiv \lceil \text{signal=go} \rceil \Rightarrow \lceil \text{gates=closed} \rceil$$

The rail signal remains GO while trains are present:

$$\text{PassingTrains}_2 \equiv \lceil \text{active}() \rceil \wedge (\lceil \text{signal=go} \rceil \ ; \ \textbf{true}) \Rightarrow \lceil \text{signal=go} \rceil$$

The rail signal will only indicate GO for at most T_{inactive} after the trains have left:

$$\text{PassingTrains}_3 \equiv \lceil \sim\text{active}() \wedge \text{signal=go} \rceil \Rightarrow \ell \leq T_{\text{inactive}}$$

The gates will remain closed for at most T_{wait} after all trains have left:

$$\text{PassingTrains}_4 \equiv \lceil (\text{gates=closed}) \wedge \sim\text{active}() \wedge \text{signal=stop} \rceil \Rightarrow \ell \leq T_{\text{wait}}$$

"slide 1155"

G.2.9 Some Observations

Some observations — after a long series of detailed examples — may now be in order:

(1) For the first time, perhaps, in this text books, we have sketched one part of an entire, albeit small, development, reordering a bit: from *domain descriptions* (in the form of assumptions about the environment in which a software design is to serve), via *requirements prescriptions*, to *software design*. "slide 1156"

(2) The examples all focused, initially, on requirements. That is to be expected, as real-time applications are typically those related to safety-critical issues.

(3) And those examples have then shown requirements to be expressible in two parts: *safety-critical requirements* issues, and *functional requirements* issues. We have, in Chap. 3, called functional requirements for domain requirements.

G.3 A Rail Switch "SLIDE 1157"

Our third example is that of a rail switch. To illustrate that concept, let us first exemplify the concept of rail units. Cf. Sect. E.3.2 on page 334.

G.3.1 A Diagrammatic Rendering of Rail Units

We refer to Fig. E.1 on page 339.

G.3.2 Intrinsic Rail Switch States

We refer to Item 1 on page 338, Item 2 on page 338 and Item 4 on page 338.

"slide 1158" With a rail unit we associate a concept of connectors. A rail unit connector is defined when two rail units are connected and is then the "point" at which they are connected. Rail units can therefore be said to have two, three or four distinct connectors. Rail switch units thus have three distinct connectors. A path of a rail unit is a pair of connectors such that a train may pass in the direction from one to the other connector. We abstract a rail unit state by a set of such paths. Fig. E.2 on page 339 shows the 12 potentially possible states of a switch unit. Many of these states are usually not provided for by actual switch units. States that include paths along the straight direction of the switch, that is, from C| to C, are usually not allowed in conjunction with other paths in a state. This excludes the "last" five states of Fig. E.2 on page 339, that is, the rightmost of the second row of states and all the four of "slide 1159" the last row of states.

G.3.3 Rail Switching Support Technologies

Some examples of rail state changing technologies are given below. (i) In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.

(ii) With the advent of reasonably reliable mechanics, pulleys and levers[3] (and steel wires), switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

(iii) This partial mechanical technology then emerged into electromechanics, and cabin tower staff was "reduced" to pushing buttons.

(iv) Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another). In any one rail net it is often seen that several such supports "slide 1160" exists simultaneously.

"slide 1161" It must be stressed that the above is just a rough sketch. In a proper narrative description the domain engineer must describe, in detail, the subsystem of electronics, electromechanics and the human operator interface (buttons, lights, sounds, etc.) of rail switches. An aspect of supporting technology includes recording the state-behaviour in response to external state switching "slide 1162" stimuli. We next give an example.

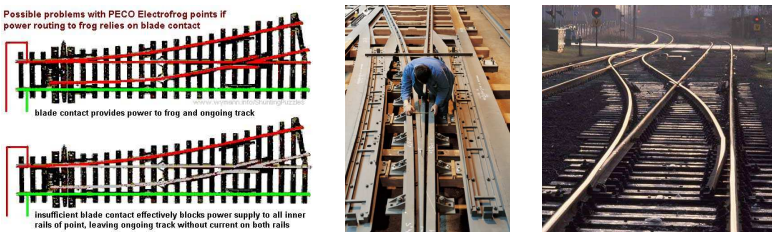


Fig. G.3. Rail Switches

G.3.4 Switches With Probabilistic Behaviour and Error States

Figure G.4 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to "switched" (s) state (C↔C/)) and re-settings (to "direct" (d) state (C↔C|)). A switch may go to the 'switched' state from the 'direct' state when subjected to a switch setting s with probability psd. Etcetera. "slide 1163"

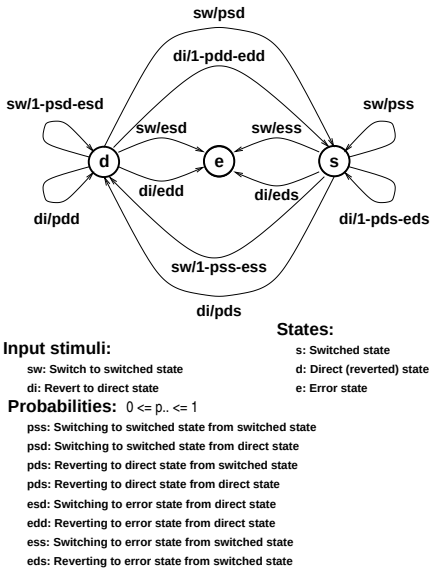


Fig. G.4. Probabilistic state switching

"slide 1164"

A rail switch equipment provider must thus furnish precise descriptions of (i) the electronic and electro-mechanical means of switching the rails, that is, of actuators and their timed behaviour; (ii) the sensing of their current

positions; (iii) possible error states; (iv) of the probabilities of correct and erroneous switching; (v) etcetera.

Using this information the domain engineer can now describe actual switches using tools like `RSL`, `MSC, LSC, Petri nets, SC, DC, TLA+` and `STeP` cf. Sect. G.1.3 Page 371.

G.4 Discussion

"slide 1165"

We have illustrated a few kinds of support technologies.

They were all characterised by notions of intrinsic versus concrete states and state switching.

Our formalisations have been rather superficial. Enough, we believe, to justify that the reader learns to see the need for first describing the intrinsics, then describing the support technologies that help 'implement' the intrinsics. This appendix has also shown the insufficiency of just using one formal description language: by showing the use of at least two other formal description languages: the Duration Calculus and probabilistic finite state machines.

G.5 Exercises

"slide 1166"

Exercise 39. Semaphore Technology:
Solution 39 Vol. II, Page 537, suggests a way of answering this exercise.

Exercise 40. Optical Gate Technology:
Solution 40 Vol. II, Page 537, suggests a way of answering this exercise.

Exercise 41. :
Solution 41 Vol. II, Page 537, suggests a way of answering this exercise.

"slide 1167"

H

Management and Organisation

"SLIDE 1169"

In this appendix we illustrate three aspects of the management and organisation facet. Two models suggest how "layers" of management collaborate and in two different styles and at two different levels of detail: as a simple functional model, Sect. H.1, and as a not quite so simple model based on communicating sequential processes Sect. H.2. The two formal models share the same basic narrative. That narrative is given in Sect. H.1.1. One model illustrates organisational aspects (Sect. H.4).

H.1 A Simple, Functional Description of Management

"SLIDE 1170"

By a functional description we mean a description which focuses on functions, that is, which explains things in terms of functions. By a simple description we mean a description which is is short. "slide 1171"

H.1.1 A Base Narrative

We think of (i) strategic, (ii) tactic, and (iii) operational managers as well as (iv) supervisors, (v) team leaders and the rest of the (vi) staff (i.e., workers) of a domain enterprise as functions. To make the description simple we think of each of the six categories (i–vi) of personnel and staff as functions, that is, there are six major domain functions related to management. "slide 1172"

Each category of staff, i.e., each function, works in state and updates that state according to schedules and resource allocations — which are considered part of the state.

To make the description simple we do not detail the state other than saying that each category works on an "instantaneous copy" of "the" state. "slide 1173"

Now think of six staff category activities, strategic managers, tactical managers, operational managers, supervisors, team leaders and workers as

six simultaneous sets of actions. Each function defines a step of collective (i.e., group) (strategic, tactical, operational) management, supervisor, team leader and worker work. Each step is considered "atomic".

"slide 1174"

Now think of an enterprise as the "repeated" step-wise simultaneous performance of these category activities. Six "next" states arise. These are, in the reality of the domain, ameliorated, that is reconciled into one state. however with the next iteration, i.e., step, of work having each category apply its work to a reconciled version of the state resulting from that category's previously yielded state and the mediated "global" state. We refer to Sect. 2.9.6, Page 79 for a caveat: The doubly[1] recursive definition of the enterprise function is a pseudo-definition. It is not a mathematically proper definition.

"slide 1175"

H.1.2 A Formalisation

type
0. Σ

value
1. str, tac, opr, sup, tea, wrk: $\Sigma \to \Sigma$
2. ame_s, ame_t, ame_o, ame_u, ame_e, ame_w: $\Sigma \to \Sigma^5 \to \Sigma$
3. objective: $\Sigma^6 \to$ **Bool**
4. enterprise, ameliorate: $\Sigma^6 \to \Sigma$
5. enterprise($\langle\sigma_s,\sigma_t,\sigma_o,\sigma_u,\sigma_e,\sigma_w\rangle$) $\equiv$
6. **let** σ'_s = ame_s(str(σ_s))($\langle\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w\rangle$),
7. σ'_t = ame_t(tac(σ_t))($\langle\sigma'_s,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w\rangle$),
8. σ'_o = ame_o(opr(σ_o))($\langle\sigma'_s,\sigma'_t,\sigma'_u,\sigma'_e,\sigma'_w\rangle$),
9. σ'_u = ame_u(sup(σ_u))($\langle\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_e,\sigma'_w\rangle$),
10. σ'_e = ame_e(tea(σ_e))($\langle\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_w\rangle$),
11. σ'_w = ame_w(wrk(σ_w))($\langle\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e\rangle$) **in**
12. **if** objective($\langle\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w\rangle$)
13. **then** ameliorate($\langle\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w\rangle$)
14. **else** enterprise($\langle\sigma'_s,\sigma'_t,\sigma'_o,\sigma'_u,\sigma'_e,\sigma'_w\rangle$)

end end

"slide 1176"

H.1.3 A Discussion of The Formal Model

A Re-Narration

0. Σ is a further undefined and unexplained enterprise state space. The various enterprise players view this state in their own way.

[1] By doubly recursive we mean (i) that the enterprise function recurses and (ii) that the definition of its next state transpires from a recursive set of local (**let**) definitions.

1. Six staff group operations, str, tac, opr, sup, tea and wrk, each act in the enterprise state such as conceived by respective groups to effect a resulting enterprise state such as achieved by respective groups.
2. Six staff group state amelioration functions, ame_s,ame _t, ame_o, ame_u, ame_e and ame_w, each apply to the resulting enterprise states such as achieved by respective groups to yield a result state such as achieved by that group.
3. An overall objective function tests whether a state summary reflects that the objectives of the enterprise has been achieved or not. "slide 1177"
4. The enterprise function applies to the tuple of six group-biased (i.e., ameliorated) states. Initially these may all be the same state. The result is an ameliorated state.
5. An iteration, that is, a step of enterprise activities, lines 5.–13. proceeds as follows:
6. strategic management operates
   - in its state space, $\sigma_s : \Sigma$;
   - effects a next (un-ameliorated strategic management) state σ'_s;
   - and ameliorates this latter state in the context of all the other player's ameliorated result states. "slide 1178"

7.–11. The same actions take place, simultaneously for the other players: tac, opr, sup, tea and wrk.

12. A test, *has* objectives *been met*, is made on the six ameliorated states.
13. If test is successful, then the enterprise terminates in an ameliorated state.
14. Otherwise the enterprise recurses, that is, "repeats" itself in new states. "slide 1179"

On The Environment *&c.*

The model does not explicitly cover interaction with the enterprise customers, suppliers, etc., the enterprise board and other such external domain "players".

Either we can include those "players" as an additional number of actions like those of str, tac, ..., wrk, each with their states, or we can think of their states and hence their state changes and interaction (communication — see below) with the enterprise being integrated into the enterprise state. "slide 1180"

Thus the omission of the environment is not serious: its modelling is just a simple extension to the given model. "slide 1181"

On Intra-communication

The model does not explicitly cover communication between different enterprise staff group members or between these and the environment. We claim now that these forms of communication are modelled by the enterprise state: in each atomic action step such intended communications are reflected in "messages" of the resulting state where these messages are, or are not handled by appropriate other enterprise staff groups in some next atomic step. "slide 1182"

On Recursive Next-state Definitions

Above, in Items 1.–14., we gave an intuition of the enterprise operating modes. But we have left un-explained the non-traditional recursive definition and use of mediated states of formula lines 6.–11. We now explain this unconventional recursion.

Let us consider just two such group activities:

$$
\begin{array}{l}
\ldots \\
\sigma'_\alpha = \mathsf{ame}_\alpha(\mathsf{alpha}(\sigma_\alpha))(< \sigma'_\beta, \sigma'_\gamma, \sigma'_\delta, \sigma'_\epsilon, \sigma'_\zeta >) \\
\sigma'_\beta = \mathsf{ame}_\beta(\mathsf{beta}(\sigma_\beta))(< \sigma'_\alpha, \sigma'_\gamma, \sigma'_\delta, \sigma'_\epsilon, \sigma'_\zeta >) \\
\ldots
\end{array}
$$

"slide 1183"

We observe that the values σ'_α and σ'_β depend on each other. Thus formula lines 6.–11. recursively defines six values. Mathematically such recursive definitions may have solutions. If so, then such solutions are said to be fix points of the equations. In conventional computer science one normally seeks what is called least fixed point solutions. Such demands are not necessary in the domain. Mathematically one can explain a process that converges towards a solution to the set of recursive equations as an iterative process. If some solution exists then the process converges and terminates in one atomic step. If it does not exist then the process does not terminate — the enterprise is badly managed and goes bankrupt !

"slide 1184"

Summary

We have sketched a formal model. It captures some aspects of enterprise management and work. It abstracts most of this management and work: there is no hint at the nature of and differences between strategic, tactic, etc., work. That is, we have neither narrate-described such work to a sufficiently concrete level nor, obviously formalised it.

H.2 A Simple, Process Description of Management "slide 1185"

H.2.1 An Enterprise System

In this model we view the six "kinds" of manager and worker behaviours as six "kinds" of processes centered around a shared state process.

There are any number,

- $\mathcal{CARD}$ StrIdx, of strategic,
- $\mathcal{CARD}$ TacIdx, of tactic and
- $\mathcal{CARD}$ OpeIdx, of operations managers,
- $\mathcal{CARD}$ SupIdx, of supervisors,
- $\mathcal{CARD}$ TeaIdx, of team leaders and
- $\mathcal{CARD}$ WrkIdx, of workers

$\mathcal{CARD}$ NameIdx expresses the **card**inality of the set of further undefined indexes in NameIdx. The staff index sets, StrIdx, TacIdx, OpeIdx, SupIdx, TeaIdx and WrkIdx are pairwise disjoint. The single state process operates concurrently with all the concurrently operating manager, supervisor, team leader and worker behaviours. "slide 1186"

H.2.2 States and The System Composition

```
type
  Ω, Idxℐ = Idx →m Ω, value idxω:Idxℐ,
  Σ, value σ:Σ
value
  enterprise: Unit → Unit
  enterprise() ≡ shared_state(σ) ‖
    ‖ {strateg_process(i)(idxω(i))|i:StrIdx} ‖ {tactic_process(i)(idxω(i))|i:TacIdx}
    ‖ {operat_process(i)(idxω(i))|i:OpeIdx} ‖ {superv_process(i)(idxω(i))|i:SupIdx}
    ‖ {teamld_process(i)(idxω(i))|i:TeaIdx} ‖ {worker_process(i)(idxω(i))|i:WrkIdx}
```

The signature of these seven functions will be given shortly.

H.2.3 Channels and Messages

Staff interaction with one another is modelled by messages sent over channels. Staff obtains current state from and "delivers" updated states to the state process, also via channels, one for each staff process. "slide 1187"

We postulate a linear ordering, <, on indexes. Channels are bidirectional — so there is only a need for $n \times (n-1)$ channels to serve n staff behaviours.

```
type
  Idx = StrIdx | TacIdx | OpeIdx | SupIdx | TeaIdx | WrkIdx
    [ axiom : pairwise disjoint ]
    [   StrIdx ∩ (TacIdx ∪ OpeIdx ∪ SupIdx ∪ TeaIdx ∪ WrkIdx) = {} ]
    [   TacIdx ∩ (OpeIdx ∪ SupIdx ∪ TeaIdx ∪ WrkIdx) = {} ]
    [   OpeIdx ∩ (SupIdx ∪ TeaIdx ∪ WrkIdx) = {} ]
    [   SupIdx ∩ (TeaIdx ∪ WrkIdx) = {} ]
    [   TeaIdx ∩ WrkIdx  = {} ]
channel
  σ_ch[ i|i:ChIdx ]: (get_Σ|Σ),
  staff_ch[ i,j|i,j:ChIdx•j<i ]: Msg
type
  Msg, get_Σ
```

"slide 1188"

H.2.4 Process Signatures

```
value
  shared_state: Σ → in, out σ_ch[ j:Idx ]  Unit
  strateg_process:
    j:StrIdx → Ω → in,out σ_ch[ j ], staff_ch[ j,i|i:Idx•i<j ] Unit
  tactic_process:
    j:TacIdx → Ω → in,out σ_ch[ j ], staff_ch[ j,i|i:Idx•i<j ] Unit
  operat_process:
    j:OpeIdx → Ω → in,out σ_ch[ j ], staff_ch[ j,i|i:Idx•i<j ] Unit
  superv_process:
    j:SupIdx → Ω → in,out σ_ch[ j ], staff_ch[ j,i|i:Idx•i<j ] Unit
  teamld_process:
    j:TeaIdx → Ω → in,out σ_ch[ j ], staff_ch[ j,i|i:Idx•i<j ] Unit
  worker_process:
    j:WrkIdx → Ω → in,out σ_ch[ j ], staff_ch[ j,i|i:Idx•i<j ] Unit
```

"slide 1189"

H.2.5 The Shared State Process

The shared_state process recurses around the following triplet of actions: waiting for a get state (get_Σ) message from any staff process, j; forwarding the state, σ, to that staff process; waiting for an updated state to be returned from staff process j.

```
value
  shared_state: Σ → in, out σ_ch[ j:Idx ]  Unit
  shared_state(σ) ≡
    ⫿ {let msg = σ_ch[ j ]? in
       case msg of
         req_Σ → (σ_ch[ j ]!σ ; shared_state(σ_ch[ j ]?)),
         _ → shared_state(σ_ch[ j ]?) end end | j:Idx}
```

From the definition of the enterprise and the staff processes one can prove that the message, msg, is either the req_Σ token or a state.

"slide 1190"

H.2.6 Staff Processes

There are six different kinds of staff processes:

- strateg_process,
- tactic_process,
- operat_process,
- superv_process,
- teamld_process and
- worker_process.

We define a process, staff_process, to generically model any of these six processes.

H.2.7 A Generic Staff Behaviour

We narrate the staff behaviour: (0.) We can model staff members as having three "alternative" behaviours; (2.–4.) doing their own work; (5.–9.) taking an initiative to act with other staff (i); and (10.–13.) being prompted by other staff (i) to react. "slide 1191"

(0.) Each staff behaviour selects whether to "do own work", to act, or to react; select is assumed to internally non-deterministically ($\sqcap$) choose "what to do".

(2.) Doing own work means to work on an "own state" (ω'). (3.) The result, ω'', is "merged" into the global state which is request-obtained from the shared state. "slide 1192"

(5.) Acting means to act on the global state (σ); (6.) by performing some "local" operations (staff_act$_j(\omega,\sigma)$) (7.) which result in local and global state changes (ω',σ') and the identification of another staff member from whom to request some action (req) which then result in some new global state (σ'') and a "local" result (res). (8.) The new global state is updated "locally" (9.) as is the local state. "slide 1193"

(10.) Reacting means to accept a request (req) from some other staff member (i); (11.) to then perform some "local" operation (staff_react$_j(req,\omega,\sigma)$) which result in local and global state changes (ω',σ'') and some result (res) (12.) The new global state is updated "locally" (13.) as is the local state.

(4., 9., 13.) The staff process iterates (by "tail-recursion"). "slide 1194"

```
value
0. staff_process(j)(ω) ≡ let (ω',wtd) = select_wtd(ω) in
1.   case wtd of
2.     own →  let ω'' = own_work(ω') in
3.            (σ_ch[j]!σ_update_j(ω'',(σ_ch[j]!get_Σ ; σ_ch[j]?)) ‖
4.             staff_process(j)(ω'')) end
5.     act  →  let σ = (σ_ch[j]!get_Σ ; σ_ch[j]?) in
6.             let (i,req,ω'',σ') = staff_act_j(ω',σ) in
7.             let (res,σ'') = (staff_ch[j,i]!(req,σ') ; staff_ch[j,i]?) in
8.             (σ_ch[j]!σ_update_j(req,res,ω'',σ'') ‖
9.              staff_process(j)(ω_update_j(req,res,ω''))) end end end
10.    react → let (i,req,σ') = ⌈⌉{staff_ch[j,i]?|i:Idx} in
11.            let (res,ω'',σ'') = staff_react_j(req,ω',σ') in
12.            (staff_ch[j,i]!(res,σ_update_j(req,res,ω'',σ'')) ‖
13.             staff_process(j)(ω_update_j(req,res,ω''))) end end end end
```

"slide 1195"

A Diagrammatic Rendition

Let us consider Fig. H.1 on the next page: The digits of Fig. H.1 on the following page refer to the line numbers of the staff_process definition (Page 393).

The figure intends to show the trace of three processes: the shared state process, staff process j serving in the own work work mode as well as in the active work mode, and staff process i (serving in the reactive work mode).

"slide 1196"

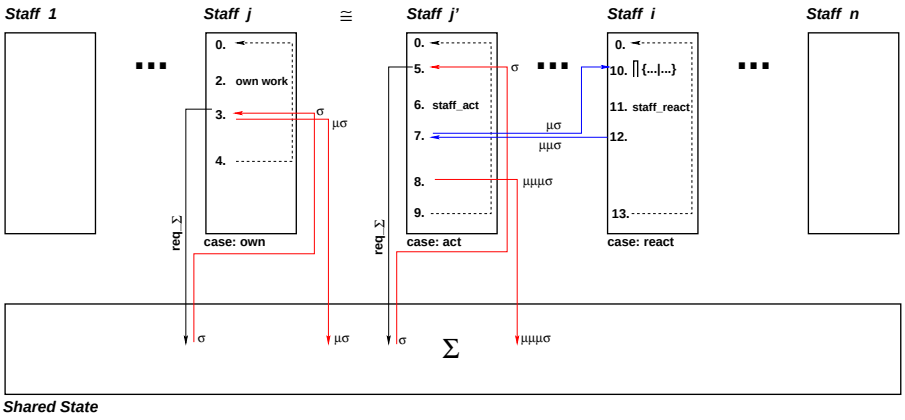


Fig. H.1. Own and 'action'-'reaction' process traces: $\mu\sigma{=}\sigma'$, $\mu\mu\sigma{=}\sigma''$, $\mu\mu\mu\sigma{=}\sigma'''$

"slide 1197"

Auxiliary Functions

A number of auxiliary functions have been used.

The select_wtd ("select what to do") clause

type
  WhatToDo = own | act | react
value
  select_wtd: $\Omega \to \Omega \times$ WhatToDo

internally, non-deterministically chooses one of the three WhatToDo alternatives as also shown in Lines 2., 6. and 11 Page 393. The choice is based on the local state (ω). The outcome of the choice, whether own, act or react, reflects, we could claim, a priority need for either of these alternatives, or just reflects human vagary! The choice is recorded in an update local state (ω').

"slide 1198"

The own_work function

value own_work: $\Omega \to \Omega$

is "own" as it applies only to the local state (ω). The modelling idea is the following: The 'own work', by any staff member, is modelled as taking place, not on the global state, but the result is eventually (see Line 3. Page 393 and,

next, the σ_update$_j$) into the global state. This models, we claim, that staff works locally on "copies" of the global state. "slide 1199"

The σ_update$_j$ function

value σ_update$_j$: $\Omega \times \Sigma \to \Sigma$

models the "merging" of a local state (ω) into the global state (σ). We do not describe this 'merging'. But such a description should be made, if need be, separately, for each case of the own_work (Line 3.), staff_act (Line 8.) and staff_react (Line 12., Page 393) alternatives. "slide 1200"

The staff_act function

type
  Request = Req_1 | Req_2 | ... | Req_m
value
  staff_act: $\Omega \times \Sigma \to$ Idx $\times$ Request $\times \Omega \times \Sigma$

applies to both the local and the global state. The purpose of the staff_act query is (line 6., Page 393) to determine with which other staff (i:Idx) the current staff (j) need interact and for what purposes (req), The staff_act query updates both the local and the global states (ω'', σ'). The staff_act query is assumed to take very little time. "slide 1201"

The ω_update function (Lines 9. and 13., Page 393)

value ω_update: Request $\times$ Result $\times \Omega \to \Omega$

updates the local state with the action request and result (req,res) being made to and yielded by staff i. We do not describe this 'merging'. But such a description should be made, if need be, separately, for each case of the staff_act (Line 9.) and staff_react (Line 13., Page 393) alternatives. "slide 1202"

The staff_react$_j$ function is, for each $_j$, a (usually large) set of functions:

type
  Result = Res_1 | Res_2 | ... | Res_m
value
  staff_react$_j$: Request $\times \Omega \times \Sigma \to$ Result $\times \Omega \times \Sigma$
  staff_react$_j$(req,ω,σ) $\equiv$
    (**case** req **of** $\text{r}_1 \to \text{op}_{j_1}(\text{r}_1)$,$\text{r}_2 \to \text{op}_{j_2}(\text{r}_2)$,...,$\text{r}_m \to \text{op}_{j_m}(\text{r}_m)$ **end**)(ω,σ)

  op_{j_i}: $\text{Req}_i \to \Omega \times \Sigma \to \text{Res}_i \times \Omega \times \Sigma$

Each of these operations are assumed to be of the kind: prepare for this operation to be carried out when doing 'own work'. We shall comment on these operations below (Sect. H.2.8 [Pages 396–398]). "slide 1203"

Assumptions

A number of assumptions have been made in expressing the staff process: The time-duration of the inter-process communications and the ω and σ updates are zero; and the time-durations of $\mathsf{staff\_act}_j(\omega,\sigma)$ and $\mathsf{staff\_react}_j(req,\omega,\sigma))$ operations are "near"-zero. These two functions do not cause any interaction with neither the shared state nor other staff processes.

"slide 1204"

The time-duration of the $\mathsf{own\_work}(\omega)$ operation may be any length of time.

(The real work is done during the local state change $\mathsf{own\_work}(\omega)$ operation and the result of this work is eventually "fed back into the global state"!)

"slide 1205"

When offering to act or react the designated partner staff behaviour will accept the offer and within a reasonably realistic time interval.

With these assumptions fulfilled it is acceptable to model global state changes as non-interleaved.

"slide 1206"

H.2.8 Management Operations

So, which are the functions $\mathsf{op}_{j_i}(\omega,\sigma)$? As is obvious from Sect. 2.9.6 there are zillions of management operations. They are loosely suggested in Sect. 2.9.6, Pages 73–79. Thus we shall not further define these here. But some comments are in order.

"slide 1207"

Focus on Management

We focus on management rather than on "workers". Operations by workers, say in a railway transport system, deal with: selling and cancelling tickets, starting, driving and stopping trains, setting signals, laying down new and maintaining rails, etcetera. Management operations are of two kinds: Own, preparatory 'work' — that may take hours and days; and dispensing or receiving order — that may take "down to" fractions of minutes. This view of 'management operations' partly justifies our staff model.

"slide 1208"

Own and Global States

Workers spend most, and managers some of their time on 'own work' — and then apply the operations of that 'own work' to local states. Worker local states are usually very clearly delineated "copies" of the global states somehow made inaccessible to other staff while subject to 'own work'. Manager local states are usually not so clearly delineated. 'Own work' is "reflected back into", that is, updates, the global state (cf. Line 3. Page 393).

"slide 1209"

State Classification

We have presented a notion of local ($\omega : \Omega$) global states ($\sigma : \Sigma$) without really saying much about these. We may also have given the impression that

these states were inert, that is, changed only when operated upon by the staff. We now redress this impression, that is, we now make it clear that states may have several components, and that some individual state components may be (i) inert dynamic[2], (ii) active dynamic[3] comprising: (ii.1) autonomous active dynamic[4], (ii.2) biddable active dynamic[5] and (ii.3) programmable active dynamic[6] and (iii) reactive dynamic[7]. "slide 1210"

Transport System States

In a transport system these are some of the state components.

Transport Net State Changes: (i) the transport net which changes state due to (i.1) wear and tear of the net, (i.2) setting and resetting of signals, (i.3) insertion and removal of links, etcetera. "slide 1211"

Net Traffic State Changes: (ii) the net traffic which changes state due to (ii.1) vehicles entering, moving around, and leaving the net, (ii.2) vehicles accidents, road/bridge/tunnel breakdowns, (ii.3) the state changes of the underlying net, etcetera. "slide 1212"

Managed State Changes: (iii) management state changes due to (iii.1) changed transport vehicle timetables being inserted, (iii.2) changed toll road fee schedules being enacted, (iii.3) changed speed limits, (iii.4) changed signalling rules, (iii.5) changed resource (incl. public vehicle) allocation, etcetera. "slide 1213"

H.2.9 The Overall Managed System

One can speak of an *overall managed system* which we consider as consisting of all staff, all explicitly shared state components, and the more implicitly observable state components of the environment. Figure H.2 on the next page tries to conceptually "picture" such an overall managed system "slide 1214"

In a domain model we try, to our best, to describe the n staff behaviours: executive, strategic, tactic and operations managers, supervisors, team leader and workers, and the various explicitly known state components of the domain, whether only observable (that is: monitorable) or also controllable (that is: generable). In a domain model we may have, through the modelling of the state components, to thus implicitly model environment state concepts. "slide 1215" "slide 1216"

[2] An entity is inert dynamic if it never changes value of it own volition.
[3] An entity is active dynamic if it changes value of it own volition.
[4] An entity is autonomous active dynamic if it changes value only of it own volition.
[5] An entity is biddable active dynamic if it can be advised to change state — but it does not have to follow that advice, i.e., that control.
[6] An entity is programmable active dynamic if its state changes, over some future time interval, can be fully controlled.
[7] An entity is reactive dynamic if it performs not necessarily fully predictable state changes in response to external stimuli (i.e., control).

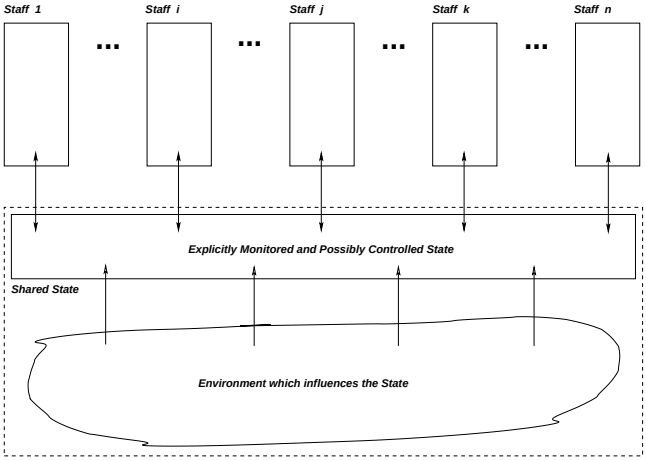


Fig. H.2. Staff and explicit (shared) and implicit states

H.2.10 Discussion

Management Operations

We leave it to the reader to draw the necessary conclusions: (i) only some state components are of concern to management, (ii) not all such state components can be controlled by management, (iii) to model the all the system state changes is thus not a concern of modelling management (and organisation), and (iv) to model all management operations is not feasible.

We have, in other sections of this domain model development, i.e., Part V, Appendices E–L, Pages 317–457, while covering other domain facets, illustrated many management operations.

"slide 1217"

Managed States

H.3 Discussion of First Two Management Models

"SLIDE 1218"

H.3.1 Generic Management Models

The first two management models, Sects. H.1–H.2, Pages 387–398, the functional, Sect. H.1, Pages 387–390, and the process descriptions, Sect. H.2, Pages 390–398, really did not show any management aspects of transportation systems.

The two models can be claimed to be generic. As such they apply to a wide variety of domain management.

"slide 1219"

The two models can also be claimed to be one anothers' "inverse". The process description, Sect. H.2, can be claimed to "implement" the functional description, Sect. H.1; each "step" of the staff processes can be claimed to correspond to an "iteration" of the "solving" of the recursive equations of Lines 6.–11., Page 388. We leave it as a research challenge for the reader "clean-up" the two formal definitions, that is, express them in a formalism, such that a theorem expressing that the process model "implements" the functional, doubly recursive model. Such a 'clean-up' might possibly involve rewriting the functional, doubly recursive model into an imperative tail-recursive model. "slide 1220"

H.3.2 Management as Scripts

It was said, above, that management is manifested in "zillions" of actions, some occurring concurrently, some occurring in strict sequence, etcetera. But nothing was then said, above, of the order, if any, of these actions. Some actions cannot be meaningfully applied before others have been applied. (i) The management decision to remove a specific link, ℓ, must occur some time after a (thus previous) management decision to insert that specific link, ℓ. (ii) "slide 1221" The management decision to construct a (new) train timetable must not occur before reasonable completion dates for the construction of the underlying rail net, the purchase of required rolling stock, and the hiring of required net and train operation staff have all been established. (iii) the management go-ahead for the start of train traffic according to a new timetable must not occur before the completion of the construction of a train (staff) rostering plan, the construction of a train maintenance plan, and the rail net construction, etcetera. "slide 1222"

One — not so extreme — interpretation of the above is that we cannot meaningfully describe specific concurrent and sequential sets of management actions but must basically, in most cases of systems, accept any such patterns of actions.

Another — slightly less extreme — interpretation of the above is that we can in some cases describe what we shall later define as a family of management scripts Let that suffice for the time being.

H.4 Transport Enterprise Organisation "SLIDE 1223"

Transportation is "home" to many different kinds of enterprises. Each of these enterprises is concerned with relatively distinct and reasonably non-overlapping issues: road (bridge, etc.) building, $\mathcal{E}_{rd_b}$, road (etc.) maintenance, $\mathcal{E}_{rd_m}$, road & car traffic signaling, $\mathcal{E}_{rd_s}$, bus sercies (i), $\mathcal{E}_{bs_i}$, fire brigade, $\mathcal{E}_{fi}$, police, $\mathcal{E}_{po}$, rail building, $\mathcal{E}_{rl_b}$, rail maintenance, $\mathcal{E}_{rl_m}$, rail & train signaling, $\mathcal{E}_{rl_s}$, train services (j), $\mathcal{E}_{tp_j}$, map making (k) $\mathcal{E}_{mm_k}$, etcetera. But they "share" the transportation net. "slide 1224"

H.4.1 Transport Organisations

Each kind of transportation enterprise, say $\mathcal{E}_i$, covers a subset, say $\mathcal{NET}_{\mathcal{E}_i}$, of the net, that is, not necessarily the entire net. For two or more $i, j, i \neq j$, it may be that $\mathcal{NET}_{\mathcal{E}_i} \cap \mathcal{NET}_{\mathcal{E}_j} \neq \{\}$. Each transportation enterprise has its own distinct staff, that is, sets of strategics managers, $\mathcal{STR}_{\mathcal{E}_i}$, tactics managers, $\mathcal{TAC}_{\mathcal{E}_i}$, operations managers, $\mathcal{OPS}_{\mathcal{E}_i}$, supervisors, $\mathcal{SUP}_{\mathcal{E}_i}$, team leaders, $\mathcal{TLD}_{\mathcal{E}_i}$, and workers, $\mathcal{WRK}_{\mathcal{E}_i}$, For some managers, supervisors, team leaders and workers the areas of the net for which they are responsible are proper, disjoint subsets of the net.

"slide 1225"

H.4.2 Analysis

To describe the transportation domain one has to model for each transportation enterprise, $\mathcal{E}_i$, separate subsets of the net $\mathcal{NET}_{\mathcal{E}_{rd_b}}$, $\mathcal{NET}_{\mathcal{E}_{rd_m}}$, $\mathcal{NET}_{\mathcal{E}_{rd_s}}$, $\mathcal{NET}_{\mathcal{E}_{bp_i}}$, $\mathcal{NET}_{\mathcal{E}_{rl_b}}$, $\mathcal{NET}_{\mathcal{E}_{rl_m}}$, $\mathcal{NET}_{\mathcal{E}_{rl_s}}$, $\mathcal{NET}_{\mathcal{E}_{tp_t}}$, $\mathcal{NET}_{\mathcal{E}_{fi}}$, $\mathcal{NET}_{\mathcal{E}_{po}}$, and $\mathcal{NET}_{\mathcal{E}_{mm_k}}$; and, for each such transportation enterprise, one has to model a number of separate enterprise structures: $\mathcal{E}_{rd_b}$, $\mathcal{E}_{rd_m}$, $\mathcal{E}_{rd_s}$, $\mathcal{E}_{bp_i}$, $\mathcal{E}_{rl_b}$, $\mathcal{E}_{rl_m}$, $\mathcal{E}_{rl_s}$, $\mathcal{E}_{tp_t}$, $\mathcal{E}_{fi}$, $\mathcal{E}_{po}$, and $\mathcal{E}_{mm_k}$.

"slide 1226"

H.4.3 Modelling Concepts

Net Kinds

In order to model the various nets, $\mathcal{NET}_{\mathcal{E}_i}$, given that we have a base model N, we introduce a notion of 'net kind': one for each of the nets $\mathcal{NET}_{\mathcal{E}_{rd_b}}$, $\mathcal{NET}_{\mathcal{E}_{rd_m}}$, $\mathcal{NET}_{\mathcal{E}_{rd_s}}$, $\mathcal{NET}_{\mathcal{E}_{bp_i}}$, $\mathcal{NET}_{\mathcal{E}_{rl_b}}$, $\mathcal{NET}_{\mathcal{E}_{rl_m}}$, $\mathcal{NET}_{\mathcal{E}_{rl_s}}$, $\mathcal{NET}_{\mathcal{E}_{tp_t}}$, $\mathcal{NET}_{\mathcal{E}_{fi}}$, $\mathcal{NET}_{\mathcal{E}_{po}}$, $\mathcal{NET}_{\mathcal{E}_{mm_k}}$, etcetera. A 'net kind', k:K, is like a type designator

```
type
  K = SimP|BusK|TrainK|MapMK
  SimK == road_b|road_m|road_s|rail_b|rail_m|rail_s|fireb|police
  BusK == bus_p1|bus_p2|...|bus_pm
  TrainK == train_p1|train_p2|...|trai_pn
  MapMK == map_m1|map_m2|...|map_mo
```

"slide 1227"

To each hub and link in a net we then associate zero, one or more net kinds. We then postulate an observer function:

```
value
  obs_K: (H|L) → K-set
```

Given a net kind we can then "extract" the net of hubs and links "carrying" that net kind:

```
value
  xtr_N: N × K → N
```

To guarantee that the extracted net is indeed a (well-formed) net we must make sure that any assignment of net kinds to hubs and links results in well-formed net kind nets:

```
value
  wf_NK: N → Bool
```

To express wf_NK we define a function

```
value
  xtr_Ks: N → K-set
```

which collects all net kinds from all hubs and links of the net. "slide 1228"

```
value
  xtr_Ks(hs,ls) ≡
    ∪{obs_Ks(h)|h:H•h ∈ hs} ∪ ∪{obs_Ks(l)|l:L•l ∈ ls}

  wf_N′(hs,ls) ≡
    ∀ k:K • k ∈ xtr_Ks(hs,ls) ⇒
      wf_N({h|h:H•h ∈ hs∧k ∈ obs_Ks(h)},{l|l:L•l ∈ ls∧k ∈ obs_Ks(l)})
```

The predicate wf_N′ extends the predicate wf_N which corresponds to the satisfaction of all the axioms given in Sect. F.3, Pages 343–348.

```
  xtr_N((hs,ls),k) ≡
    ({h|h:H•h ∈ hs∧k ∈ obs_Ks(h)},{l|l:L•l ∈ ls∧k ∈ obs_Ks(l)})
    pre k ∈ xtr_Ks(hs,ls)
```

"slide 1229"

Enterprise Kinds

In order to model the various enterprises, $\mathcal{E}_i$, given that we have a base model for business staff, SIdx, we introduce a notion of 'enterprise kind': one for each of the enterprise kind: $\mathcal{E}_{rd_b}$, $\mathcal{E}_{rd_m}$, $\mathcal{E}_{rd_s}$, $\mathcal{E}_{bp_i}$, $\mathcal{E}_{rl_b}$, $\mathcal{E}_{rl_m}$, $\mathcal{E}_{rl_s}$, $\mathcal{E}_{tp_t}$, $\mathcal{E}_{fi}$, $\mathcal{E}_{po}$, $\mathcal{E}_{mm_k}$, etcetera. A 'enterprise kind', b:B, is like a type designator

```
type
  E = SimE|BusE|TrainE|MapMakE
  SimE == road_b|road_m|road_s|rail_b|rail_m|rail_s|fireb|police
  BusE == bus_p1|bus_p2|...|bus_pm
  TrainE == train_p1|train_p2|...|train_pn
  MapMakE == map_m1|map_m2|...|map_mo
```

"slide 1230"

Staff Kinds

To each staff we then associate an enterprise kind.

value
obs_B: SIdx → B

We may further have to impose that interaction between staff, viz.:

staff_ch[i,j]

be subject to an "internal business constraint":

obs_B(i) = obs_B(j)

"slide 1231"

Staff Kind Constraints

Narrative

The staff, $\mathcal{S}taff_{\mathcal{E}_i}$, of each transportation enterprise, $\mathcal{E}_i$, can be roughly categorised into: strategic managers, $\mathcal{STR}_{\mathcal{E}_i}$, tactics managers, $\mathcal{TAC}_{\mathcal{E}_i}$, operations managers, $\mathcal{OPS}_{\mathcal{E}_i}$, supervisors, $\mathcal{SUP}_{\mathcal{E}_i}$, team leaders, $\mathcal{TLD}_{\mathcal{E}_i}$, and workers $\mathcal{WRK}_{\mathcal{E}_i}$, as already mentioned (Page 400). And each of these can be further sub-categorised.

"slide 1232"

Formalisation

We suggest the "beginnings" of a formalisation.

type
StaffK == stra|tac|ope|sup|tld|wrk
value
obs_StaffK: SIdx → StaffK

A more realistic model, for a given enterprise, would provide a far more detailed categorisation. Typically there might be several "layers" of strategic and of tactic and of operations management. Similarly a model might detail different kinds of supervisor, team leader and worker ("blue collar") staff.

"slide 1233"

Hierarchical Staff Structures

"slide 1234"

We refer to Fig. 2.1, Page 80.

Matrix Staff Structures

"slide 1235"

Net and Enterprise Kind Constraints

Narrative

A link (or a hub) is either a road or a rail link (hub). If a link is a road link then the two hubs that it connects are road hubs. If a hub is a road hub then all the links emanating from the hub are road links. If a link is a rail link then the two hubs that it connects are rail hubs. If a hub is a rail hub then all the links emanating from the hub are rail links. In consequence we may speak of disjoint road nets and rail nets. The enterprises associated with a road [rail] net must be road [rail] related (building, maintenance, signaling, bus [train] services, police, etc.). "slide 1236"

Formalisation

```
type
  NetK: road|rail|...
value
  obs_NetK: (H|L) → NetK

  xtr_L: LI → L-set → L,xtr_H: HI → H-set → H
  xtr_L(li)(ls) ≡ ∃!l:L • l ∈ ls∧obs_LI(l)=li
  xtr_H(hi)(hs) ≡ ∃!h:H • h ∈ hs∧obs_HI(h)=hi

  wf_N″: N → Bool
  wf_N″(hs,ls) ≡
    ∀ h:H • h ∈ hs ⇒
      ∀ li:LI • li ∈ obs_LIs(h) ⇒ obs_NetK(xtr_L(li)(ls))=obs_NetK(h) ∧
    ∀ l:L • l ∈ ls ⇒
      ∀ hi:HI • hi ∈ obs_HIs(l) ⇒ obs_NetK(xtr_H(hi)(hs))=obs_NetK(l)
```

The predicate wf_N″ extends the predicate wf_N′ given earlier (Page 401). "slide 1237"

```
value
  netk: N → NetK-set
  netk(hs,ls) ≡ {obs_NetK(h)|h:H•h ∈ hs}∪{obs_NetK(l)|l:L•l ∈ ls}

  road_k:NetK-set={road_b,road_m,road_s,bus_p1,bus_p2,...,bus_pm,fireb,police},
  rail_k:NetK-set={rail_b,rail_m,rail_s,train_p1,train_p2,...,train_pn,fireb,police}

  wf_N‴: N → Bool
  wf_N‴(hs,ls) ≡
    let nks = netk(hs,ls) in
    case nks of
      {road} → xtr_Ks(hs,ls)⊆road_k,{rail} → xtr_Ks(hs,ls)⊆rail_k,_ → false
    end end
```

"slide 1238" The predicate wf_N‴ extends wf_N″ given earlier (Page 403).

H.4.4 Net Signaling

In the previous section on support technology we did not describe who or which "ordered" the change of hub states. We could claim that this might very well be a task for management.

(We here look aside from such possibilities that the domain being modelled has some further support technology which advises individual hub controllers as when to change signals and then into which states. We are interested in finding an example of a management & organisation facet — and the upcoming "slide 1239" one might do!)

Narrative

So we think of a 'net hub state management' for a given net. That management is divided into a number of 'sub-net hub state managements' where the sub-nets form a partitioning of the whole net. For each sub-net management there are two kinds management interfaces: one to the overall hub state management, and one for each of interfacing sub-nets. What these managements do, what traffic state information they monitor, etcetera, you can yourself "dream" up. Our point is this: We have identified a management organisation. "slide 1240"

Formalisation

```
type
  HIsLIs = HI-set × LI-set
  MgtNet′ = HIsLIs × N
  MgtNet = {| mgtnet:MgtNet′ • wf_MgtNet(mgtnet)|}
  Partitioning′ = HIsLIs-set × N
  Partitioning = {| partitioning:Partitioning′ • wf_Partitioning(partitioning)|}
value
  wf_MgtNet: MgtNet′ → Bool
  wf_MgtNet((his,lis),n) ≡
    [ The his component contains all the hub ids. of links identified in lis ]
  wf_Partitioning: Partitioning′ → Bool
  wf_Partitioning(hisliss,n) ≡
    ∀ (his,lis):HIsLIs • (his,lis) ∈ hisliss ⇒ wf_MgtNet((his,lis),n)∧
    [ no sub−net overlap and together they ″span″ n ]
```

H.5 Discussion

"slide 1241"

"slide 1242"
"slide 1243"

H.6 Exercises

"slide 1244"

Exercise 42. 41:
Solution 42 Vol. II, Page 537, suggests a way of answering this exercise.

Exercise 43. 42:
Solution 43 Vol. II, Page 537, suggests a way of answering this exercise.

Exercise 44. 43:
Solution 44 Vol. II, Page 537, suggests a way of answering this exercise.

"slide 1245"

I

Rules and Regulations

"SLIDE 1247"

For the motivation and the principles and techniques for carrying out this stage of development of domain rules and regulations description we refer to Sect. 2.9.7 (starting Page 80).

I.1 Two Informal Examples

Example. 28 – **Trains at Stations: The "Available Station" Rule and Regulation:**

- Rule: *In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:*

  In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.*

•

"slide 1248"

Example. 29 – **Trains Along Lines: The "Free Sector" Rule and Regulation:**

- Rule: *In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:*

  There must be at least one free sector (i.e., without a train) between any two trains along a line.
- Regulation: *If it is discovered that the above rule is not obeyed, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.*

•

The above incomplete regulation will be completed in Sect. I.2.2 on page 411.

I.2 Two Formal Examples

"slide 1249"

We shall develop Example 29 on the previous page into a partial, formal specification. That is, not complete, but "complete enough" for the reader to see what goes on.

"slide 1250"

I.2.1 The "Free Sector" Rule

Analysis of Informal "Free Sector" Rule Text

We start by analysing the text of the rule and regulation. The rule text: *There must be at least one free sector (i.e., without a train) between any two trains along a line.* contains the following terms: free (a predicate), sector (an entity), train (an entity) and line (an entity). We shall therefore augment our formal model to reflect these terms. We start by modelling sectors and sector descriptors, lines and train position descriptors, we assume what a train is,,

"slide 1251"

and then we model the predicate free.

Formalised Concepts of Sectors, Lines, and Free Sectors

```
type
  Sect′ = H × L × H,
  SectDescr = HI × LI × HI
  Sect = {|(h,l,h′):Sect′ • obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
  SectDescr = {|(hi,li,hi′):SectDescr′ •
                  ∃ (h,l,j′):Sect•obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
  Line′ = Sect*,
  Line = {|line:Line′•wf_Line(line)|}
  TrnPos′ = SectDescr*
  TrnPos = {|trnpos′:TrnPos′•∃ line:Line•conv_Line_to_TrnPos(line)=trnpos′|}
value
  wf_Line: Line′ → Bool
  wf_Line(line) ≡
    ∀ i:Nat • {i,i+1}⊆inds(line) ⇒
      let (_,l,h)=line(i),(h′,l′,_)=line(i+1) in h=h′ end
  conv_Line_to_TrnPos: Line → TrnPos
  conv_Line_to_TrnPos(line) ≡
    ⟨(obs_HI(h),obs_LI(l),obs_HI(h′))|1≤i≤len line∧line(i)=(h,l,h′)⟩
```

"slide 1252"

The function lines yield all lines of a net.

```
value
  lines: N → Line-set
  lines(hs,ls) ≡
    let lns = {⟨(h,l,h′)⟩|h,h′:H,l:L•proper_line((h,l,h′),(hs,ls))}
              ∪ {ln^ln′|ln,l′:Line•{ln,ln′}⊆lns∧adjacent(ln,ln′)} in
    lns end
```

The function lines makes use of an auxiliary function:

```
  adjacent: Line × Line → Bool
  adjacent((_,l,h),(h′,l′,_)) ≡ h=h′
    pre {obs_LI(l),obs_LI(l′)}⊆ obs_LIs(h)
```

"slide 1253"

We reformulate traffic in terms of train positions.

```
type
  TF = T ~m~> (N × (TN ~m~> TrnPos))
```

We formulate a necessary property of traffic, namely that its train positions correspond to actual lines of the net.

```
value
  wf_TF: TF → Bool
  wf_TF(tf) ≡
    ∀ t:T•t ∈ dom tf ⇒
      let ((hs,ls),trnposs) = tf(t) in
      ∀ trn:TN • trn ∈ dom trnposs ⇒
        ∃ line:Line • line ∈ lines(hs,ls) ∧
          trnposs(trn) = conv_Line_to_TrnPos(line) end
```

Nothing prevents two or more trains from occupying overlapping train positions. They have "merely" – and regrettably – crashed. But such is the domain. So wf_TF(tf) is not part of an axiom of traffic, merely a desirable property.

"slide 1254"

```
value
  has_free_Sector: TN × T → TF → Bool
  has_free_Sector(trn,(hs,ls),t)(tf) ≡
    let ((hs,ls),trnposs) = tf(t) in
    (trn ∉ dom trnposs ∨ (tn ∈ dom trnposs(t) ∧
     ∃ ln:Line • ln ∈ lines(hs,ls) ∧
       is_prefix(trnposs(trn),ln))(hs,ls)) ∧
       ~∃ trn′:TN • trn′ ∈ dom trnposs ∧ trn′≠trn ∧
         trnposs(trn′)=conv_Line_to_TrnPos(⟨follow_Sect(ln)(hs,ls)⟩)
    end
    pre exists_follow_Sect(ln)(hs,ls)
```

```
is_prefix: Line × Line → N → Bool
is_prefix(ln,ln′)(hs,ls) ≡ ∃ ln″:Line • ln″ ∈ lines(hs,ls) ∧ ln⌢ln″=ln′
```

The test ln″ ∈ lines(hs,ls) in the definition of is_prefix is not needed for the cases where that function is invoked as only shown here.

The function follow_Sect yields the sector following the argument line, if such a sector exists.

```
exists_follow_Sect: Line → Net → Bool
exists_follow_Sect(ln)(hs,ls) ≡
    ∃ ln′:Line•ln′ ∈ lines(hs,ls)∧ln⌢ln′ ∈ lines(hs,ls)
    pre ln ∈ lines(hs,ls)
follow_Sect: Line → Net ~→ Sect
follow_Sect(ln)(hs,ls) ≡
    let ln′:Line•ln′ ∈ lines(hs,ls)∧ln⌢ln′ ∈ lines(hs,ls) in hd ln′ end
    pre line ∈ lines(hs,ls)∧exists_follow_Sect(ln)(hs,ls)
```

"slide 1255"

Formalisation of the "Free Sector" Rule

We doubly recursively define a function free_sector_rule(tf)(r). tf is that part of the traffic which has yet to be "searched" for non-free sectors. Thus tf is "counted" up from a first time t till the traffic tf is empty. That is, we assume a finite definition set tf . r is like a traffic but without the net. Initially r is the empty traffic. r is "counted" up from "earliest" cases of trains with no free sector ahead of them. The recursion stops, for a given time when there are no more train positions to be "searched" for that time; and when the "to-be-searched" traffic is empty.

"slide 1256"

```
type
    TNPoss = T ~m→ (TN → TrnPos)
value
    free_sector_rule: TF × TF → TNPoss
    free_sector_rule(tf)(r) ≡
        if tf=[ ] then r else
        let t:T•t ∈ dom tf∧smallest(t)(tf) in
        let ((hs,ls),trnposs)=tf(t) in
        if trnposs=[ ] then free_sector_rule(tf\{t})(r) else
        let tn:TN•tn ∈ dom trnposs in
        if exists_follow_Sect(trnposs(tn))(hs,ls)∧∼has_free_Sector(tn,(hs,ls),t)(tf)
            then
                let r′ = if t ∈ dom r then r else r ∪ [t↦[ ]] end in
                free_sector_rule(tf†[t↦((hs,ls),trnposs\{tn})])(r†[t↦r(t)∪[tn↦trnposs(tn)]]) end
            else
                free_sector_rule(tf†[t↦((hs,ls),trnposs\{trn})])(r)
        end end end end end end
```

smallest(t)(tf) ≡ ∼∃ t′:T• t′isin **dom** tf∧t′<t **pre** t ∈ **dom** tf

"slide 1257"

The rule is obeyed for a traffic tf if free_sector_rule(tf)([])=[].

Please observe that the rule is obeyed if two or more trains occupy the same sectors! If you do not like that, then you must state so. This is left as an exercise! "slide 1258"

I.2.2 The "Free Sector" Regulation

Completion of the "Free Sector" Regulation

The "free sector" regulation read: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic. That regulation text must be made more precise. Some precisions could be: (i) Administrative action: *The railway regulatory agency must establish an investigation seeking to uncover the reasons for the breach of the "free sector" rule.* (ii) Legal action: *If the railway regulatory agency finds that a potentially punishable staff conduct has occurred then the public prosecutor must be notified and given all investigation material.* (iii) Current traffic correction: *As soon as it has been established that a train has progressed into a non-free sector that train must be stopped and the train ahead of it must be informed of the situation.* (iv) Future traffic corrections: *If the railway regulatory agency finds that the 'business processes' could be improved then the rail and train operators are asked to improve their train traffic handling procedures.* Usually all of these parts are present in the regulation. "slide 1259" "slide 1260"

Analysis of the Completed "Free Sector" Regulation

The nature of part regulations (i, ii, iv) is such that they cannot be formalised. Part regulation (iii) can be formalised: (iii.A) the offending, the "rear", train must be stopped, (iii.B) possible "follower" trains must presumably be informed or stopped, and (iii.C) the "ahead" train must be informed. We omit any formalisation. The regulation statements (iii.A–.C) amount to management actions.

I.3 Review "slide 1261"

I.4 Exercises "slide 1262"

Exercise 45. 51:

Solution 45 Vol. II, Page 537, suggests a way of answering this exercise.

Exercise 46. 52:
Solution 46 Vol. II, Page 537, suggests a way of answering this exercise.

"SLIDE 1263"

J

Scripts

"SLIDE 1265"

For the motivation and the principles and techniques for carrying out this stage of development of domain script description we refer to Sect. 2.9.8 (starting Page 84).

J.1 Informal Examples

Example. 30 – **Bus [Train] Timetables (Schedules):** A bus [train] timetable usually covers more than one bus [train] line. All bus [train] lines operate within a named area. Each bus [train] line has a name and a list of list of bus [train] ride descriptions. Each bus [train] ride description lists two or more bus stop [station] visits. Each bus stop [station] visit lists the name of the bus stop [station], and the approximate time of visit [the precise arrival and departure times. The lists of bus stop [station] visits are ordered chronologically. "slide 1266"
The list of bus [train] ride descriptions are ordered chronologically. The bus stops [stations] must be bus stops [stations] of the named area. Train timetables lists further information per train ride description: train type/classes, train services: restaurant/bistro/..., wireless Internet, location of handicap compartments, whether mandatory seat reservation, etcetera. •

We shall further explore the concept of timetables in Sect. J.2. "slide 1267"

Example. 31 – **Aircraft Flight Simulator Script:** The following is part of a longer example of a script for an aircraft simulator:

1 Takeoff:
   (a) Record time
   (b) Release brakes and taxi onto runway 26L
   (c) Advance power to "FULL"
   (d) Maintain centerline of runway
   (e) At 50 knots airspeed lift nose wheel off runway

(f) At 70 knots ease back on the yoke to establish a 10 degree pitch up attitude
(g) Maintain a climb AIRSPEED of 80 knots
(h) Maintain a climb AIRSPEED of 80 knots
(i) Raise Gear when there is no more runway to land on
(j) At "500" feet above the ground raise the FLAPS to "0"
(k) Reduce power to about "2300" RPM at "1000" feet above the ground (AGL)

"slide 1268"

2 Climb out:
(a) Maintain runway heading and climb to "2400" feet
(b) At "2400" feet start a climbing LEFT turn
(c) Start to roll out when you see "140" in the DG window
(d) Maintain a heading of "130"
(e) Watch your NAV 1 CDI, when the needle is three dots LEFT of center, start your RIGHT turn to a heading of "164"
(f) Track outbound on the POMONA VOR 164 radial

3 Level off:
(a) Begin to level off when the altimeter reads "3900" feet
(b) Maintain "4000" feet
(c) Reduce power to about "2200" [2400] RPM

"slide 1269"

4 Course change *1:
(a) Watch your NAV 2 CDI, when the needle is one dot LEFT of center, start your RIGHT turn to a heading of "276"
(b) When your heading indicator reads "265" start to roll out
(c) After you have rolled out, press "P" to pause the simulation
(d) Record your: NAV, I, DME, DIST, ALTITUDE, AIRSPEED VSI, GEAR, FLAPS, MAGS, STROBE, LIGHTS
(e) Press "P" to continue the simulation
(f) Track outbound on the PARADISE VOR 276 radial that your NAV 2 OBI is displaying
(g) Track outbound on the PARADISE VOR 276 radial that your NAV 2 OBI is displaying
(h) Switch ¿DME to "NAV 2"

"slide 1270"

5 Altitude change:
(a) When the DME on NAV 2 reads "29.0", press "P" to pause the simulation
(b) Record your: ALTITUDE, AIRSPEED, VSI, HEADING
(c) Press "P" to continue the simulation
(d) Tune NAV 1 to "113.1" and set radial "276" in the upper window
(e) Track inbound on the VAN NUYS VOR "096" radial (course 276) that your NAV 1 OBI is displaying
(f) Switch DME to "NAV 1"

6 Etcetera.

•

"slide 1271"

Example. 32 – **Bill of Lading:**

- B/L No.:
- Shipper:
- Reference No.:
- Consignee:
- Notify address:
- Vessel:
- Port of loading:
- Port of discharge:
- Shipper's description of goods:
  - ⋆ Gross weight:
  - · of which ... is on deck at Shipper's risk; the Carrier not being responsible for loss or damage howsoever arising.
  - ⋆ Measure:
  - ⋆ Quality:
  - ⋆ Quantity:
  - ⋆ Condition:
  - ⋆ Contents value unknown.

- SHIPPED at the Port of Loading in apparent good order and condition on board the Vessel for carriage to the Port of Discharge or so near thereto as she may safely get the goods specified above

- Freight payable as per:
- Dated (by charter part:):

"slide 1272"

- Freight Advance:
- Time used for loading:
  - ⋆ Days:
  - ⋆ Hours:
- Freight payable at:
- Place and date of issue:
- Signature:
- Number of original Bs:

Conditions:

- (1) All terms and conditions, liberties and exceptions of the Charter Party, dated as overleaf, including the Law and Arbitration Clause, are herewith incorporated.
- (2) General Paramount Clause.
  - ⋆ (a) The Hague Rules contained in the International Convention for the Unification of certain rules relating to Bills of Lading, dated Brussels the 25th August 1924 as enacted in the country of shipment, shall apply to this Bill of Lading. When no such enactment is in force in the country of shipment, the corresponding legislation of the country of destination shall apply, but in respect of shipments to which no such enactments are compulsorily applicable, the terms of the said Convention shall apply.
  - ⋆ (b) Trades where Hague-Visby Rules apply. In trades where the International Brussels Convention 1924 as amended by the Protocol signed at Brussels on February 23rd 1968 - the Hague- Visby Rules - apply compulsorily, the provisions of the respective legislation shall apply to this Bill of Lading.

"slide 1273"

⋆ (c) The Carrier shall in no case be responsible for loss of or damage to the cargo, howsoever arising prior to loading into and after discharge from the Vessel or while the cargo is in the charge of another Carrier, nor in respect of deck cargo or live animals.

- (3) General Average.
  General Average shall be adjusted, stated and settled according to York-Antwerp Rules 1994, or any subsequent modification thereof, in London unless another place is agreed in the Charter Party. Cargo's contribution to General Average shall be paid to the Carrier even when such average is the result of a fault, neglect or error of the Master, Pilot or Crew. The Charterers, Shippers and Consignees expressly renounce the Belgian Commercial Code, Part II, Art. 148.
- (4) New Jason Clause.
  In the event of accident, danger, damage or disaster before or after the commencement of the voyage, resulting from any cause whatsoever, whether due to negligence or not, for which, or for the consequence of which, the Carrier is not responsible, by statute, contract or otherwise, the cargo, shippers, consignees or the owners of the cargo shall contribute with the Carrier in General Average to the payment of any sacrifices, losses or expenses of a General Average nature that may be made or incurred and shall pay salvage and special charges incurred in respect of the cargo. If a salving vessel is owned or operated by the Carrier, salvage shall be paid for as fully as if the said salving vessel or vessels belonged to strangers. Such deposit as the Carrier, or his agents, may deem sufficient to cover the estimated contribution of the goods and any salvage and special charges thereon shall, if required, be made by the cargo, shippers, consignees or owners of the goods to the Carrier before delivery.

"slide 1274"

- (5) Both-to-Blame Collision Clause.
  If the Vessel comes into collision with another vessel as a result of the negligence of the other vessel and any act, neglect or default of the Master, Mariner, Pilot or the servants of the Carrier in the navigation or in the management of the Vessel, the owners of the cargo carried hereunder will indemnify the Carrier against all loss or liability to the other or non-carrying vessel or her owners in so far as such loss or liability represents loss of, or damage to, or any claim whatsoever of the owners of said cargo, paid or payable by the other or non-carrying vessel or her owners to the owners of said cargo and set-off, recouped or recovered by the other or non-carrying vessel or her owners as part of their claim against the carrying Vessel or the Carrier.
  The foregoing provisions shall also apply where the owners, operators or those in charge of any vessel or vessels or objects other than, or in addition to, the colliding vessels or objects are at fault in respect of a collision or contact.

•

J.2 Timetable Scripts

"SLIDE 1275"

We shall view timetables as scripts.

In this section (that is, Pages 419–430) we shall first narrate and formalise the **syntax**, including the well-formedness of timetable scripts, then we consider the **pragmatics** of timetable scripts, including the bus routes prescribed by these journey descriptions and timetables marked with the status of its currently active routes, and finally we consider the **semantics** of timetable, that is, the traffic they denote.

In the next section, Sect. J.3, on licenses for bus traffic, we shall assume the timetable scripts of this section. "slide 1276"

We all have some image of how a timetable may manifest itself. Figure J.1 shows some such images.

Fig. J.1. Some bus timetables: Italy, India and Norway

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.[1] 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". "slide 1277"

[1] We do not distinguish between a time and a time description. That is, when we say December 17, 2008, 15: 56 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 !".

J.2.1 The Syntax of Timetable Scripts

49 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.
50 A TimeTable associates to Bus Line Identifiers a set of Journies.
51 Journies are designated by a pair of a BusRoute and a set of BusRides.
52 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.
53 A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
54 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.
55 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.
56 A Fraction is a **Real** properly between 0 and 1.
57 The Journies must be well_formed in the context of some net.

"slide 1278"

type
49. T, BLId, BId
50. TT = BLId $\overrightarrow{m}$ Journies
51. Journies′ = BusRoute × BusRides
52. BusRoute = BusStop × BusStop* × BusStop
53. BusRides = BId $\overrightarrow{m}$ BusSched
54. BusSched = T × T* × T
55. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
56. Frac = {|r:**Real**•0<r<1|}
57. Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}

"slide 1279"

The free *n* in ∃ n:N • wf_Journies(j)(n) is the net given in the license.

Well-formedness of Journies

58 A set of journies is well-formed
59 if the bus stops are all different[2],
60 if a defined notion of a bus line is embedded in some line of the net, and
61 if all defined bus trips (see below) of a bus line are commensurable.

value
58. wf_Journies: Journies → N → **Bool**
58. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡

[2] This restriction is, strictly speaking, not a necessary domain property. But it simplifies our subsequent formulations.

59. diff_bus_stops(bs1,bsl,bsn) ∧
60. is_net_embedded_bus_line(⟨bs1⟩⌢bsl⌢⟨bsn⟩)(hs,ls) ∧
61. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

"slide 1280"

62 The bus stops of a journey are all different
63 if the number of elements in the list of these equals the length of the list.

value
62. diff_bus_stops: BusStop × BusStop* × BusStop → **Bool**
62. diff_bus_stops(bs1,bsl,bsn) ≡
63. **card elems** ⟨bs1⟩⌢bsl⌢⟨bsn⟩ = **len** ⟨bs1⟩⌢bsl⌢⟨bsn⟩

We shall refer to the (concatenated) list (⟨bs1⟩⌢bsl⌢⟨bsn⟩ = **len** ⟨bs1⟩⌢bsl⌢⟨bsn⟩) of all bus stops as the bus line. "slide 1281"

64 To explain that a bus line is embedded in a line of the net
65 let us introduce the notion of all lines of the net, lns,
66 and the notion of projecting the bus line on link sector descriptors.
67 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.

value
64. is_net_embedded_bus_line: BusStop* → N → **Bool**
64. is_net_embedded_bus_line(bsl)(hs,ls)
65. **let** lns = lines(hs,ls),
66. cbln = compress(proj_on_links(bsl)(**elems** bsl)) **in**
67. ∃ ln:Line • ln ∈ lns ∧ cbln ∈ projs_on_links(ln) **end**

"slide 1282"

68 Projecting a list (*) of BusStop descriptors (mkBS(hi,li,f,hi′)) onto a list of Sector Descriptors ((hi,li,hi′))
69 we recursively unravel the list from the front:
70 if there is no front, that is, if the whole list is empty, then we get the empty list of sector descriptors,
71 else we obtain a first sector descriptor followed by those of the remaining bus stop descriptors.

value
68. proj_on_links: BusStop* → SectDescr*
68. proj_on_links(bsl) ≡
69. **case** bsl **of**
70. ⟨⟩ → ⟨⟩,
71. ⟨mkBS(hi,li,f,hi′)⟩⌢bsl′ → ⟨(hi,li,hi′)⟩⌢proj_on_links(bsl′)
71. **end**

"slide 1283"

72 By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.
73 The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.
74 We express the function recursively.
75 If the argument sector descriptor list an empty result sector descriptor list is yielded;
76 else
77 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"
78 in front of the compression of the rest of the argument sector descriptor list.

"slide 1284"

```
72. compress: SectDescr* → SectDescr-set → SectDescr*
73. compress(sdl)(sds) ≡
74.   case sdl of
75.     ⟨⟩ → ⟨⟩,
76.     ⟨sd⟩^sdl′ →
77.       (if sd ∈ sds then ⟨sd⟩ else ⟨⟩ end)
78.       ^compress(sdl′)(sds\{sd}) end
```

In the last recursion iteration (line 78.) the continuation argument sds\{sd} can be shown to be empty: {}.

"slide 1285"

79 We recapitulate the definition of lines as sequences of sector descriptions.
80 Projections of a line generate a set of lists of sector descriptors.
81 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 "orderedness" is expressed by making that arbitrary subset isx into an ordered list isl, isl=sort(isx).

```
type
79. Line′ = (HI×LI×HI)* axiom ... type Line = ... Page 408
value
80. projs_on_links: Line → Line′-set
80. projs_on_links(ln) ≡
81.   {⟨isl(i)|i:⟨1..len isl⟩⟩|isx:Nat-set•isx⊆inds ln∧isl=sort(isx)}
```

"slide 1286"

82 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.
83 The result list of (arbitrary) indices must contain all the members of the argument set;

84 and "earlier"elements of the list must precede, in value, those of "later" elements of the list.

```
value
82. sort: Nat-set → Nat*
82. sort(isx) as isl
83.   post card isx = lsn isl ∧ isx = elems isl ∧
84.       ∀ i:Nat • {i,i+1}⊆inds isl ⇒ isl(i)<isl(i+1)
```

"slide 1287"

85 The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:
86 All the intermediate bus stop times must equal in number that of the bus stop list.
87 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.
88 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 timewise.
89 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.
90 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.
91 As for Item 90 but now with respect to last, resp. destination bus stop.
92 And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops
93 they must exist and fit time-wise.

"slide 1288"

```
value
85. commensurable_bus_trips: Journies → N → Bool
85. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
86.   ∀ (t1,til,tn):BusSched•(t1,til,tn)∈ rng js∧len til=len bsl∧
87.     case len til of
88.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
89.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧fit((til(1),t2),(bsl(1),bsn))(hs,ls),
90.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
91.           real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
92.           ∀ i:Nat•{i,i+1}⊆inds til ⇒
93.             real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

"slide 1289"

94 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:

95 All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).
96 There exists links, l, l′, for the identified bus stop links, li, li′,
97 such that these links connect the identified bus stop hubs.
98 Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops
99 The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.
100 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.
101 The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.
102 So we leave also this as an exercise to the reader to complete.

94. real_and_fit: (T×T)×(BusStop×BusStop) → N → **Bool**
94. real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
95. {hi,hi′,hi″,hi‴}⊆his(hs)∧
96. ∃ l,l′:L•{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
97. obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
98. afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

99. distance: BusStop × BusStop → N → Distance
100. distance(bs1,bs2)(n) ≡ ... [left as an exercise !] ...

101. afit: TI → Distance → **Bool**
102. [time interval fits distance between bus stops]

J.2.2 The Pragmatics of Timetable Scripts

A main purpose of a timetable is to bring an order into the traffic, as seen from the side of net operators (signalling etc.), train operators and passengers. With a net which is owned by one enterprise, many different train operators on that one net, and with cross-train passengers a consolidated timetable offers a common, fixed interface.

Let us illustrate this point by two examples.

Subset Timetables

The pragmatics of a timetable may include its decomposition into a number of sub-timetables. When speaking of two timetables it is often convenient to make sure that bus line identifiers occuring in both designate identical bus routes.

103 A bus line identifier occurring in two timetables is said to define compatible bus rides in those two timetables provided the corresponding two bus routes are identical.

```
103 have_compatible_BLIds: TT × TT → Bool
103 have_compatible_BLIds(tti,ttj) ≡
103   ∀ blid:BLId • blid ∈ dom tti ∩ dom ttj
103     ⇒ let (bri,_)=tti(blid),(brj,_)=ttj(blid) in bri=brj end
```

"slide 1293"

104 Two journies are similar if they have identical bus line identified bus routes. Thus a bus line identified journey in one timetable can be similar to a bus line identified journey in another or the same timetable if the bus line identifiers are the same and the journies are the same.

105 A timetable, stt, is said to be a sub-timetable of a timetable, tt, if every bus line identified bus ride of similar journies is also an identical bus line identified bus ride of tt.

```
value
104 are_similar_Js: Journies × Journies → Bool
104 are_similar_Js((bri,_),(brj,_)) ≡ bri=brj

105 is_sub_TT: TT × TT → Bool
105 is_sub_TT(stt,tt) ≡
105   ∀ sblid,blid:BLId•sblid=blid∧sblid ∈ dom stt∧blid ∈ dom tt
105     ⇒ ∀ (sbr,sbrs),(br,brs):Journies•(sbr,sbrs)=stt(sblid)∧(br,brs)=tt(blid)
105       ⇒ sbr=br ∧ ∀ bid:BId•bid ∈ dom sbrs ∩ dom br
105         ⇒ sbrs(bid)=brs(bid)
105   pre have_compatible_BLIds(stt,tt)
```

"slide 1294"

106 We can thus generate all sub-timetables of a timetable.

```
106 all_sub_TTs: TT → TT-set
106 all_sub_TTs(tt) ≡ {stt|stt:TT•is_sub_TT(stt,tt)}
```

107 Two timetables, stt_i and stt_j, are said to be disjoint if they share no same bus line identifier bus rides.

```
107 are_disjoint_TTs: TT × TT → Bool
107 are_disjoint_TTs(tti,ttj) ≡
107   ∀ blidi,blidj:BLId•blidj=blidj∧blidi ∈ dom tti∧blidj ∈ dom ttj
107     ⇒ dom tti(blidi) ∩ dom ttj(blidj) = {}
107   pre have_compatible_BLIds(tti,ttj)
```

So disjointness is purely a matter of whether two bus rides (of the same bus route and bus line identifier) have different bus ride identifiers. The time schedule is not considered.

"slide 1295"

108 Two timetables can be merged into one timetable provided they are disjoint.
109 Merging two disjoint timetables result in a timetable which has exactly the bus line identified journies of either of the timetables.

```
108 can_be_merged_TTs: TT × TT → Bool
108 can_be_merged_TTs(tti,ttj) ≡ are_disjoint_TTs(tti,ttj)

109 merge_TTs: TT × TT → TT
109 merge_TTs(tti,ttj) as tt
109   pre are_disjoint_TTs(tti,ttj) [ i.e., have_compatible_BLIds(tti,ttj) ]
109   post is_sub_TT(tti,tt′)∧is_sub_TT(ttj,tt′)
109     ∧ dom tt = dom tti ∪ dom ttj
109     ∧ ∀ blid:BLId•blid ∈ dom tt ∧ blid ∈ dom tti ∪ dom ttj
109       ⇒ let ((br,brs),(bri,brsi),(brj,brsj)) = (tt(blid),tti(blid),ttj(blid)) in
109         dom brsi ∩ dom brsj = {} ∧ dom brsi ∪ dom brsj = dom brs
109         ∧ brs = brsi ∪ brsj end
```

"slide 1296"

From a timetable one can construct any number of sub-timetables.

110 Given a timetable, tt, and given a mapping of bus line identifiers, ex., blid, of tt into the set, bids, of bus ride identifiers of the bus rides of tt(blid), construct, cons_STT(tt,blid_to_bids_map), the sub-timetable, stt, of tt where stt exactly lists the so identified bus rides of tt.

```
value
110 cons_STT: TT × (BLId →m BId-set) → TT
110 cons_STT(tt,id_map) ≡
110   [ blid ↦ (tt(blid))(bid)
110     | blid:BLId,bid:BId • blid ∈ dom id_map ∧ bid ∈ id_map(blid) ]
110   pre dom id_map ≠ {} ∧ dom id_map ⊆ dom tt ∧
110     ∧ ∀ blid:BLId•blid ∈ dom(tt)
110       ⇒ id_map(blid)≠{}∧id_map(blid)⊆rng tt(blid)
```

"slide 1297"

111 Given a timetable, tt, and given a mapping of bus line identifiers, ex., blid, of tt into the set, bids, of bus ride identifiers of the bus rides of tt(blid), construct, cons_compl_STT(tt,blid_to_bids_map), the sub-timetable, stt, of tt where stt exactly lists the **other** identified bus rides of tt.

```
111 cons_compl_STT: TT × (BLId →m BId-set) → TT
111 cons_compl_STT(tt,id_map)
111   let idmap = [ blid ↦ bids | blid:BLId,bids:BId-set
111         • (blid ∈ dom tt \ dom id_map ∧ bids=dom tt(blid))
111           ∨ blid ∈ dom tt ∩ dom id_map ∧ bids=id_map(blid) ]
111   construct_STT(tt,idmap) end
```

The following should be proven:

theorem:
∀ tt:TT, id_map • **pre** construct_STT(tt,id_map) ⇒
merge_TTs(cons_STT(tt,id_map),cons_compl_STT(tt,id_map))
= tt =
merge_TTs(cons_compl_STT(tt,id_map),cons_STT(tt,id_map))

"slide 1298"

Some auxiliary functions might come in handy at a later stage.

112 Given a bus line identifier to inquire whether it is the bus line identifier of a proper, non empty sets of bus rides in a given timetable.
113 Given a bus line identifier and a bus ride identifier to inquire whether they together identify a proper bus ride of a given timetable.
114 Given a bus ride identifier and a time table to to inquire whether there is a bus line identifier of that timetable for which the bus ride identifier is defined.
115 Given a bus line identifier and a bus ride identifier to find, if it exists, the bus route and ride schedule of that identification.

"slide 1299"

value
112. is_def: BLId×TT→**Bool**,
112. is_def(blid,tt) ≡ blid ∈ **dom** tt ∧ tt(blid)≠[ ]

113. is_def: BLId×BId×TT→**Bool**,
113. is_def(blid,bid,tt) ≡ **dom** tt ∧ bid ∈ **dom** tt(blid)

114. is_def: BId×TT→**Bool**,
114. is_def(bid,tt) ≡ ∃ blid:BLId • is_def(blid,bid,tt)

115. inquire: BLId×BId×TT$\xrightarrow{\sim}$(BusRoute×BusSched),
115. inquire(blid,bid,tt) ≡
115. **let** (br,brs)=tt(blid) **in** (br,brs(bid)) **end**
115. **pre** is_def(blid,bid,tt)

"slide 1300"

Marked Timetables

By a marked timetable, MrkdTT, we mean a table whose journey and bus stop entries may be annotated by a set of notes (JN respectively BSN). The table is very similar to a bus timetable.

116 A MrkdTT associates to bus line identifiers, BLId, marked bus rides (MrkdBusRides)
117 A MrkdBusRides associates to bus identifiers, BId, marked journies (MrkdJourney).

118 A MrkdJourney consists of a pair: a set of zero one or more Journey Notes (JN) and a mapping from BusStops to zero, one or more BusvStop Notes (BSN).
119 A JN is either one of `"commenced", "cancelled", "inserted"` or other.
120 A BSN is either a quadruplet (or something else) where the quadruplet records the arrival and departure times at and from the bus stop as well as how many passengers alighted and board the bus at that bus stop.

"slide 1301"

type
116. MrkdTT = BLId $\overrightarrow{m}$ MrkdBusRides
117. MrkdBusRides = BId $\overrightarrow{m}$ MrkdJourney
118. MrkdJourney = JN-**set** × (BusStop $\overrightarrow{m}$ BSN-**set**)
119. JN = ″`commenced`″ | ″`cancelled`″ | ″`inserted`″ | ...
120. BSN = mkBSN(s_at:T,s_dt:T,s_pa:**Nat**,s_pb:**Nat**) | ...

One can define a function which reconstruct time tables from marked timetables.

value
construct_TT: MrkdTT $\xrightarrow{\sim}$ TT

"slide 1302"

The Marking of Timetables

The marked timetables are updated whenever a ride is committed, cancelled or a new inserted, or and whenever a bus "passes" a bus stop.

1 The JNmarkTT(jn,(bln,bn))(mtt) command results in a marked timetable mtt′ updated with a journey note jn.
2 The updated line bln and bus ride bn must be in mtt.
3 All but the bln,bn entries must be unchanged in mtt′ (wrt. mtt).
4 The specific bln,bn entry must have only its journey note part updated.

value
1. JNmarkTT: JN × (BLId×BId) → MarkdTT → MrkdTT
1. JNmarkTT(jn,(bln,bn))(mtt) **as** mtt′
2. **pre** bln ∈ **dom** mtt ∧ bn ∈ **dom** mtt(bln)
3. **post dom** mtt = **dom** mtt′
3. ∧ ∀ blid:BLId•blid ∈ **dom** mtt•**dom** mtt(blid)=**dom** mtt′(blid)
3. ∧ ∀ blid:BLId•blid ∈ **dom** mtt\{bln}•mtt(blid)=mtt′(blid)
4. ∧ **let** (js,mbr)=(mtt(bln))(bn), (js′,mbr′)=(mtt′(bln))(bn) **in**
4. js′=js ∪{jn} ∧ mbr=mbr′ **end**

"slide 1303"

1 The BSNmarkTT((bs,bsn),(bln,bn))(mtt) command results in a marked timetable mtt′ updated with a bus stop bs note bsn.
2 The updated line bln and bus ride bn must be in mtt.

3 The updates bus stop bs must be in the bus ride and that entry must not already have been updated.
4 All but the bln,bn entries must be unchanged in mtt′ (wrt. mtt).
5 The specific bln,bn entry must have only its bus ride note part updated and only for the bs entry.

value
1. BSNmarkTT: (BusStop × BSN) × (BLId × BId) → MarkdTT → MrkdTT
1. BSNmarkTT((bs,bsn),(bln,bn))(mtt) **as** mtt′
2. **pre** bln ∈ **dom** mtt∧bn ∈ **dom** mtt(bln)
3. ∧ **let** (_,mbr)=(mtt(bln))(bn) **in** bs ∈ **dom** mbr ∧ mbr(bn)={} **end**
4. **post dom** mtt = **dom** mtt′
4. ∧ ∀ blid:BLId•blid ∈ **dom** mtt•**dom** mtt(blid)=**dom** mtt′(blid)
4. ∧ ∀ blid:BLId•blid ∈ **dom** mtt\{bln}•mtt(blid)=mtt′(blid)
5. ∧ **let** (js,mbr) = (mtt(bln))(bn),(js′,mbr′) = (mtt′(bln))(bn) **in**
5. js′=js ∧ mbr′=mbr†[bs↦{bsn}] **end**

"slide 1304"

J.2.3 The Semantics of Timetable Scripts

One form of timetable denotations is the bus traffic implied by a timetable.

Bus Traffic

121 We postulate a type of Buses.
122 From a bus one can observe the value of a number of attributes: current number of passengers, identity of driver, number of passengers who alighted and boarded at the most recent bus stop, etc. (We let X stand for any one of these attributes.)
123 Bus traffic maps discrete times into the pair of a bus net and the positions of buses. "slide 1305"
124 A bus positions is either at a hub, on a link or at a bus stop.
125 When a bus is at a hub we can also observe from which link it came and to which link it proceeds.
126 When a bus is on a link we can observe how far it has progressed down the link from one of the two hubs it connects.
127 When a bus is at a bus stop — which is like "on a link" — we can observe that bus stop accordingly.
128 Fractions have also be described earlier.

"slide 1306"

type
121. Bus
value
122. obs_X: Bus → X
type

123. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos)))
124. BPos = atHub | onLnk | atBS
125. atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
126. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
127. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
128. Frac = {|r:**Real**•0<r<1|}

We omit detailing necessary well-formedness constraints – such as (i) all bus positions being on the designated net, (ii) traffic moving monotonically, (iii) no two buses of the same pair of bus line and bus identification at the same time (or otherwise conflicting), (iv) no "ghost" busses, etcetera. .

"slide 1307"

From a bus timetable we can generate the set of all bus traffics that satisfy the bus timetable. (We have covered this notion earlier.)

```
value
  gen_BusTraffic: TT → BusTraffic-infset
  gen_BusTraffic(tt) as btrfs
    post ∀ btrf:BusTraffic • btrf ∈ btrfs ⇒ on_time(btrf)(tt)
```

"slide 1308"

We leave it to the reader to define the on_time predicate.

J.2.4 Discussion

We have built the foundations for a theory of timetables. We have not yet formulated theorems let alone proven any such.

J.3 A Contract Language

"SLIDE 1309"

J.3.1 Narrative

Preparations

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.)

"slide 1310"

A Synopsis

Contracts obligate transport companies to deliver bus traffic according to a timetable. The timetable is part of the contract. A contractor may subcontract (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[3]. The cancellation rights are spelled out in the contract[4]. A sub-contractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor[5]. Etcetera. "slide 1311"

A Pragmatics and Semantics Analysis

The "works" of the bus transport contracts are two: the timetables and, implicitly, the designated (and obligated) bus traffic. A bus timetable appears to define one or more bus lines, with each bus line giving rise to one or more bus rides. We assume a timetable description along the lines of Sect. J.2. 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. "slide 1312"

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[6] We assume that such insertions must also be reported back to the contractor. "slide 1313"

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.) "slide 1314"

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:*

[3] We do not treat this aspect further in this book.
[4] See Footnote 3.
[5] See Footnote 3.
[6] Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

to cancel a bus ride (allowed, with restrictions); (iii) *insert:* to insert a bus
"slide 1315" ride; and (iv) *subcontract:* to sub-contract part or all of a contract.

The Final Narrative

We leave, as an exercise, the expression of a complete narrative.
Instead we proceed directly to a formalisation.

J.3.2 A Formalisation

Syntax

We treat separately, the syntax of contracts (for a schematised example see Page 432) and the syntax of the actions implied by contracts (for schematised
"slide 1316" examples see Page 433).

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.

We assume a context (a global state) in which all contract actions (including
"slide 1317" contracting) takes place and in which the implicit net is defined.

129 contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.
130 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.
131 A contract body stipulates a timetable and the set of operations that are mandated or allowed by the contractor.
132 An Operation is either a "start" (i.e., start a bus ride), a bus ride "cancel"lation, a bus ride "insert", or a "subcontrat"ing operation.
"slide 1318"

type
129. CId, CNm
130. Contract = CId × CNm × CNm × Body
131. Body = Op-**set** × TT
132. Op == ″start″ | ″cancel″ | ″insert″ | ″subcontract″

An abstract example contract:
(cid,cnm$_i$,cnm$_j$,({″start″,″cancel″,″insert″,″sublicense″},tt))
"slide 1319"

Actions

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 432) 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′.

"slide 1320"

All actions are being performed by a sub-contractor in a context which defines that sub-contractor cnm, the relevant net, say n, the base contract, referred here to by cid (from which this is a sublicense), and a timetable tt of which tt′ is a subset. contract name cnm′ is new and is to be unique. The subcontracting action can (thus) be simply transformed into a contract as shown on Page 432. "slide 1321"

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 135 on the next page

"slide 1322"

We observe that the essential information given in the start, cancel and insert action prescriptions is the same; and that the RSL record-constructors (mkSta, mkCan, mkIns) make them distinct. "slide 1323"

Uniqueness and Traceability of Contract Identifications

133 There is a "root" contract name, rcid.
134 There is a "root" contractor name, rcnm.

value
133 rcid:CId
134 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.

"slide 1324"

135 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.
136 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
135 gen_CId: CId × CNm × Time → CId
136 obs_CId: CId ~→ CIdL [ pre obs_CId(cid):cid≠rcid ]
136 obs_CNm: CId ~→ CNm [ pre obs_CNm(cid):cid≠rcid ]
136 obs_Time: CId ~→ Time [ pre obs_Time(cid):cid≠rcid ]
```

"slide 1325"

137 All contract names are unique.

```
axiom
137 ∀ cid,cid′:CId•cid≠cid′⇒
137   obs_CId(cid)≠obs_CId(cid′) ∨ obs_CNm(cid)≠obs_CNm(cid′)
137   ∨ obs_LicNm(cid)=obs_CId(cid′)∧obs_CNm(cid)=obs_CNm(cid′)
137     ⇒ obs_Time(cid)≠obs_Time(cid′)
```

"slide 1326"

138 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
138 contract_trace: CId → LCIdCNmTTrace
138 contract_trace(cid) ≡
138   case cid of
138     rcid → ⟨⟩,
138     _ → contract_trace(obs_LicNm(cid))^⟨obs_TraceTriple(cid)⟩
138   end

138 obs_TraceTriple: CId → TraceTriple
138 obs_TraceTriple(cid) ≡
138   mkTrTr(obs_CId(cid),obs_CNm(cid),obs_Time(cid))
```

"slide 1327"

The trace is generated in the chronological order: most recent contract name generation times last.

Well, there is a theorem to be proven once we have outlined the full formal model of this contract language: namely that time entries in contract name traces increase with increasing indices.

theorem
∀ licn:LicNm •
∀ trace:LicNmLeeNmTimeTrace • trace ∈ license_trace(licn) ⇒
∀ i:**Nat** • {i,i+1}⊆**inds** trace ⇒ s_t(trace(i))<s_t(trace(i+1))

"slide 1328"

Semantics

Execution State

Local and Global States: Each sub-contractor has an own local state and has access to a global state. All sub-contractors access the same global state. The global state is the bus traffic on the net. There is, in addition, a notion of running-state. It is a meta-state notion. The running state "is made up" from the fact that there are n sub-contractors, each communicating, as contractors, over channels with other sub-contractors. The global state is distinct from sub-contractor to sub-contractor – no sharing of local states between sub-contractors. We now examine, in some detail, what the states consist of. "slide 1329"

Global State: The net is part of the global state (and of bus traffics). We consider just the bus traffic.

type
55. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI) 420

123. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos))) 429
124. BPos = atHub | onLnk | atBS
125. atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
126. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
127. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

We shall consider BusTraffic (with its Net) to reflect the global state. "slide 1330"

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 the second LicNm.

"slide 1331"

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.[7]

"slide 1332"

type
  BusNo
  BusΣ = FreeBusesΣ × ActvBusesΣ × BusHistsΣ
  FreeBusesΣ = BusStop $\overrightarrow{m}$ BusNo-**set**
  ActvBusesΣ = BusNo $\overrightarrow{m}$ BusInfo
  BusInfo = BLId×BId×LicNm×LeeNm×BusTrace
  BusHistsΣ = Bno $\overrightarrow{m}$ BusInfo*
  BusTrace = (Time×BusStop)*
  LorBusΣ = LeeNm $\overrightarrow{m}$ (LicNm $\overrightarrow{m}$ ((BLId×BId) $\overrightarrow{m}$ (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.

"slide 1333"

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

"slide 1334"

[7] 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.

```
value
  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
```

"slide 1335"

```
value
  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))
```

"slide 1336"

```
value
  update_ActΣ_FreeΣ:
    LeeNm×LicNm×BusStop×(BLId×BId)→BusΣ→BusΣ×BNo
  update_ActΣ_FreeΣ(leen,licn,bs,(blid,bid))(freeσ,actvσ) ≡
    let bno:Bno • bno ∈ freeσ(bs) in
    ((freeσ\{bno},actvσ ∪ [ bno↦(blid,bid,licnm,leenm,⟨⟩) ]),bno) end
    pre bs ∈ dom freeσ ∧ bno ∈ freeσ(bs) ∧ bno ∉ dom actvσ ∧ [ bs exists ... ]
```

"slide 1337"

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.

139 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.

140 We also assume a constant set, leens, of sub-contractors. In reality sub-contractors, that is, transport companies, come and go, are established and go out of business. But assuming constancy does not materially invalidate our model. Its emphasis is on contracts and their implied actions — and these are unchanged wrt. constancy or variability of contract holders.

141 There is an initial bus traffic, tr.

142 There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.

143 To maintain the bus traffic "spelled out", in total, by timetable tt one needs a number of buses.

144 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.
145 We leave it to the reader to define a function which ascertain the minimum number of buses needed to implement traffic tr.

"slide 1338"

value
139. net : N,
140. leens : LeeNm-**set**,
141. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
142. t_0 : T • $t_0 \geq$ **min dom** tr,
143. min_no_of_buses : **Nat** • necessary_no_of_buses(itt),
144. busnos : BusNo-**set** • **card** busnos $\geq$ min_no_of_buses

145. necessary_no_of_buses: TT → **Nat**

"slide 1339"

146 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]).
147 The initial, i.e., the distinguished, contract has a name, root_licn.
148 The initial contract can only perform the `"sublicense"` operation.
149 The initial contract has a timetable, tt.
150 The initial contract can thus be made up from the above.

"slide 1340"

value
146. root_leen,init_ln : LeeNm • root_leen $\notin$ leens $\wedge$ initi_leen $\in$ leens,
147. root_licn : LicNm
148. iops : Op-**set** = {″`sublicense`″},
149. itt : TT,
150. init_lic:License = (root_licn,root_leen,(iops,itt),init_leen)

"slide 1341"

Initial sub-contractor contract States:

type
InitLicΣs = LeeNm $\xrightarrow[m]{}$ LicΣ
value
ilσ:LicΣ=([init_leen ↦ [root_leen ↦ [iln ↦ init_lic]]]
∪ [leen ↦ [ ] | leen:LeeNm • leen ∈ leenms\{init_leen}],[ ],[ ])

"slide 1342"

Initial sub-contractor Bus States:

151 Initially each sub-contractor possesses a number of buses.
152 No two sub-contractors share buses.
153 We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.
154 We do not prescribe a "satisfiable and practical" such initial assignment (ibσs).
155 But we can constrain ibσs.
156 The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.
157 Active bus states must be empty.
158 No two free bus states must share buses.
159 All bus histories are void.

"slide 1343"

type
151. AllBuses′ = LeeNm $\overrightarrow{m}$ BusNo-**set**
152. AllBuses = {|ab:AllBuses′•∀ {bs,bs′}⊆**rng** ab∧bns≠bns′⇒bns ∩ bns′={}|}
153. InitBusΣs = LeeNm $\overrightarrow{m}$ BusΣ
value
152. allbuses:Allbuses • **dom** allbuses = leenms ∪ {root_leen} ∧ ∪ **rng** allbuses = busnos

153. ibσs:InitBusΣs
154. wf_InitBusΣs: InitBusΣs → **Bool**
155. wf_InitBusΣs(iσs) ≡
156. **dom** iσs = leenms ∧
157. ∀ (_,abσ,_):BusΣ•(_,abσ,_) ∈ **rng** iσs ⇒ abσ=[ ] ∧
158. ∀ (fbiσ,abiσ),(fbjσ,abjσ):BusΣ •
158. {(fbiσ,abiσ),(fbjσ,abjσ)}⊆**rng** iσs
158. ⇒ (fbiσ,actiσ)≠(fbjσ,actjσ)
158. ⇒ **rng** fbiσ ∩ **rng** fbjσ = {}
159. ∧ actiσ=[ ]=actjσ

"slide 1344"

Communication Channels: The running state is a meta notion. It reflects the channels over which contracts are issued; messages about committed, cancelled and inserted bus rides are communicated, and fund transfers take place.

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_[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.

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 = ...

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 = ...

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)

Buses↔Time Channel Each bus needs to know what time it is.

channel { b_to_t[b] | b:BNo • b ∈ busnos } BTMSG
type
BTMSG ...

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

"slide 1348"

of communication between a sub-contractor and the traffic. 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. "slide 1349"

The System Behaviour

The system behaviour starts by establishing a number of licenseholder and bus_ride behaviours and the single time_clock and bus_traffic behaviours "slide 1350"

```
value
  system: Unit → Unit
  system() ≡
    licenseholder(init_leen)(ilσ(init_leen),ibσ(init_leen))
    ‖ (‖ {licenseholder(leen)(ilσ(leen),ibσ(leen))
          | leen:LeeNm•leen ∈ leens\{init_leen}})
    ‖ (‖ {bus_ride(b,leen)(root_lorn,"nil")
          | leen:LeeNm,b:BusNo •leen ∈ dom allbuses ∧ b ∈ allbuses(leen)})
    ‖ time_clock(t0) ‖ bus_traffic(tr)
```

"slide 1351"

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.

Only a designated licenseholder behaviour is initialised to a single, received license. "slide 1352"

Semantic Elaboration Functions

The Licenseholder Behaviour:

160 The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.

161 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.

160. **licenseholder**: LeeNm → (LicΣ×BusΣ) → **Unit**
161. **licenseholder**(leen)(licσ,busσ) ≡
161. **licenseholder**(leen)((**lic_ops**⌈|**bus_mon**⌈|**neg_licenses**)(leen)(licσ,busσ))

"slide 1353"

The Bus Behaviour:

162 Buses ply the network following a timed bus route description.
A timed bus route description is a list of timed bus stop visits.
163 A timed bus stop visit is a pair: a time and a bus stop.
164 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.
165 Bus behaviours start with a "nil" bus route description.

"slide 1354"

type
162. TBR = TBSV*
163. TBSV = Time × BusStop
value
164. conTBR: BusRoute × BusSched → TBR
164. conTBR((dt,til,at),(bs1,bsl,bsn)) ≡
164(a) ⟨(dt,bs1)⟩
164(b) ⌢ ⟨(til[i],bsl[i])|i:**Nat**•i:⟨1..**len** til⟩⟩
164(c) ⌢ ⟨(at,bsn)⟩
  pre: **len** til = **len** bsl
type
165. BRD == "nil" | TBR

"slide 1355"

166 The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,
167 The bus repeatedly observes the time, t, and its position, po, in the traffic.
168 There are now four case distinctions to be made.
169 If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.
170 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.
171 If the bus is 'en route', at a bus stop, then it so informs its owner and continues the journey.
172 In all other cases the bus continues its journey

"slide 1356"

```
value
166. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
166.     in,out l_to_b[leen,bno], in,out b_to_tr[bno], in b_to_t[bno] Unit
166. bus_ride(leen,bno)(licn,brd) ≡
167.   let t = b_to_t[bno]? in
167.   let (bus,pos) = (b_to_tr[bno]!reqBusAndPos(bno,t) ; b_to_tr[bno]?) in
168.   case (brd,pos) of
169.     ("nil",mkAtBS(_,_,_,_)) →
169.           let (licn,brd′) = (l_to_b[leen,bno]!reqBusRid(pos);l_to_b[leen,bno]?) in
169.           bus_ride(leen,bno)(licn,brd′) end
170.     (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
170s           l_to_b[l,b]!BusΣMsg(t,pos);
170          l_to_b[l,b]!BusHistΣMsg(licn,bno);
170          l_to_b[l,b]!FreeΣ_ActΣMsg(licn,bno) ;
170           bus_ride(leen,bno)(ilicn,"nil"),
171.     (⟨(t,pos),(t′,bs′)⟩^brd′,mkAtBS(_,_,_,_)) →
171s           l_to_b[l,b]!BusΣMsg(t,pos) ;
171           bus_ride(licn,bno)(⟨(t′,bs′)⟩^brd′),
172.     _ → bus_ride(leen,bno)(licn,brd) end end end
```

"slide 1357"

In formula line 167 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 170s and 171s:

```
Simple:   170s   l_to_b[l,b]!BusΣMsg(pos);
Revised: 170r   l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));

Simple:   171s   l_to_b[l,b]!BusΣMsg(pos);
Revised: 171r   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.
-

•
•

"slide 1358"

The Global Time Behaviour:

173 The time_clock is a never ending behaviour — started at some time t_0.
174 The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.
175 At any moment the time_clock behaviour may not be inquired.
176 After a skip of the clock or an inquiry the time_clock behaviour continues, non-deterministically either maintaining the time or advancing the clock!

```
value
173. time_clock: T →
173.     in,out {l_to_t[leen] | leen:LeeNm • leen ∈ leenms}
173.     in,out {b_to_t[bno] | bno:BusNo • bno ∈ busnos} Unit
173. time_clock:(t) ≡
175.   (skip ⌈⌉
174.   ([]{l_to_t[leen]? ; l_to_t[leen]!t | leen:LeeNm•leen ∈ leens})
174.   ⌈⌉ ([]{b_to_t[bno]? ; b_to_t[bno]!t | bno:BusNo•bno ∈ busnos})) ;
176.   (time_clock:(t) ⌈⌉ time_clock(t+δt))
```

"slide 1359"

The Bus Traffic Behaviour:

177 There is a single bus_traffic behaviour. It is, "mysteriously", given a constant argument, "the" traffic, tr.
178 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.
179 The request for a bus position comes from some bus.
180 The bus positions are part of the traffic at time t.
181 The bus_traffic behaviour, after informing of a bus position reverts to "itself".

```
value
177. bus_traffic: TR → in,out {b_to_tr[bno]|bno:BusNo•bno ∈ busnos} Unit
177. bus_traffic(tr) ≡
179.   [] { let reqBusAndPos(bno,time) = b_to_tr[b]? in assert b=bno
178.        if time ∉ dom tr then chaos else
180.        let (_,bps) = tr(t) in
178.        if bno ∉ dom tr(t) then chaos else
178.        b_to_tr[bno]!bps(bno) end end end end | b:BusNo•b ∈ busnos} ;
181.   bus_traffic(tr)
```

"slide 1360"

License Operations:

182 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 433.
183 To perform any licensed operation the sub-contractor needs to know the time and
184 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.
185 Thus there is a case distinction amongst four alternatives.
186 This case distinction is expressed in the four lines identified by: 186.
187 All the auxiliary functions, besides the action arguments, require the same state arguments.

"slide 1361"

```
value
182. lic_ops: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
182. lic_ops(leen)(licσ,busσ) ≡
183.   let t = (time_channel(leen)!req_Time;time_channel(leen)?) in
184.   let (licn,act) = choice(licσ)(busσ)(t) in
185.   (case act of
186.      mkCon(blid,bid)  → cndct(licn,leenm,t,act),
186.      mkCan(blid,bid)  → cancl(licn,leenm,t,act),
186.      mkIns(blid,bid)  → insrt(licn,leenm,t,act),
186.      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)
```

"slide 1362"

Bus Monitoring: Like for the bus_ride behaviour we decompose the bus_monitoring behaviour into two behaviours. The local_bus_monitoring behaviour monitors the buses that are commissioned by the sub-contractor. The licensor_bus_monitoring 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)
```

"slide 1363"

188 The **local_bus_mon**itoring function models all the interaction between a contract holder and its despatched buses.
189 We show only the communications from buses to contract holders.
190
191
192
193
194
195
196
197
198

"slide 1364"

```
188. local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
189.       → in {l_to_b[leen,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
188. local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
190.   let (bno,msg) = □{(b,l_to_b[l,b]?)|b:BNo•b ∈ allbuses(leen)} in
194.   let (blid,bid,licn,lorn,trace) = abσ(bno) in
191.   case msg of
192.     BusΣMsg(t,bs) →
196.       let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
196.       (licσ,(fbσ,abσ′,histσ)) end,
198.     BusHistΣMsg(licn,bno) →
198.       let lbσ′ =
198.         update_LorBusΣ(obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))(lbσ) in
198.       l_to_l[leen,obs_LorNm(licn)]!Licensor_BusHistΣMsg(licn,leen,bno,blid,bid,tr);
198.       ((rlσ,slσ,lbσ′),busσ) end
197.     Free_Act_ΣMsg(licn,bno) →
198.       let (fbσ′,abσ′) = update_Free_Act_Σ(bno,bs)(fbσ,abσ) in
198.       (licσ,(fbσ′,abσ′)) end
198.   end end end
```

"slide 1365"

199
200
201
202
203

"slide 1366"

```
199. licensor_bus_mon: lorn:LorNm → (LicΣ×BusΣ)
199.       → in {l_to_l[lorn,leen]|leen:LeeNm•leen ∈ leenms\{lorn}} (LicΣ×BusΣ)
199. licensor_bus_mon(lorn)(licσ,busσ) ≡
199.   let (rlσ,slσ,lbhσ) = licσ in
199.   let (leen,Licensor_BusHistΣMsg(licn,leen″,bno,blid,bid,tr))
                                  = □{(leen′,l_to_l[lorn,leen′]?)|leen′:LeeNm•leen′ ∈ leenms\{lorn}} in
```

199. **let** lbhσ' =
199. **update_BusHist**Σ(obs_LorNm(licn),licn,leen$''$,(blid,bid),(bno,trace))(lbhσ) **in**
199. l_to_l[leenm,obs_LorNm(licnm)]!**Licensor_BusHist**Σ**Msg**(b,blid,bid,lin,lee,tr);
199. ((rlσ,slσ,lbhσ'),busσ)
199. **end end end**

"slide 1367"

License Negotiation:

204
205
206
207
208
209
210
211
212
213
214
215

"slide 1368"

204.
205.
206.
207.
208.
209.
210.
211.
212.
213.
214.

"slide 1369"

The Conduct Bus Ride Action:

216 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.

(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.

"slide 1370"

216.
216(a)
216(b)
216(c)
216(d)
216(e)
216(f)
216(g)

"slide 1371"

The Cancel Bus Ride Action:

217 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 216(b) 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 !

"slide 1372"

217.
217(a)
217(b)
217(c)

"slide 1373"

The Insert Bus Ride Action:

218 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 216(b) 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 216(d)–216(g).

"slide 1374"

218.
218(a)
218(b)
218(c)
218(d)

"slide 1375"

The Contracting Action:

219 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.

"slide 1376"

219.
219(a)
219(b)
219(c)
219(d)
219(e)
219(f)

"slide 1377"

J.3.3 Discussion

J.4 Review

"slide 1378"

J.5 Exercises

"slide 1379"

Exercise 47. 61:
Solution 47 Vol. II, Page 538, suggests a way of answering this exercise.

Exercise 48. 62:
Solution 48 Vol. II, Page 538, suggests a way of answering this exercise.

"slide 1380"

K

Human Behaviour

"slide 1382"

K.1 A First, Informal Example: Automobile Drivers

K.1.1 A Narrative

We have already exemplified aspects of human behaviour in the context of the transportation domain, namely vehicle drivers not obeying hub states. Other example can be given: drivers moving their vehicle along a link in a non-open direction, drivers waving their vehicle off and on the link, etcetera. Whether rules exists that may prohibit this is, perhaps, irrelevant. In any case we can "speak" of such driver behaviours — and then we ought formalise them ! "slide 1383"

K.1.2 A Formalisation

But we decide not to. For the same reason that we skimped proper formalisation of the violation of the *"obey traffic signals"* rule. But, by now, you've seen enough formulas and you ought trust that it can be done.

off_on_link: Traffic → (T×T) $\xrightarrow{\sim}$ (V $\xrightarrow[m]{}$ VPos×VPos)
wrong_direction: Traffic → T $\xrightarrow{\sim}$ (V $\xrightarrow[m]{}$ VPos)

"slide 1384"

K.2 A Second Example: Link Insertion

K.2.1 A Diligent Operation

The int_Insert operation of Sect. F.4.2 Page 354 was expressed stating necessary pre-conditions.

value
21′ pre_int_Insert: Ins → N → **Bool**

```
21″  pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2     s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
         case op of
1        2oldH(hi′,l,hi″) → {hi′,hi″}⊆iohs(hs),
2        1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∧h∉ hs∧obs_HI(h)∉ iohs(hs),
3        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**.

"slide 1385"

K.2.2 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
  21′  pre_int_Insert: Ins → N → Bool
  21″  pre_int_Insert(Ins(op))(hs,ls) ≡
  ⋆2     s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
           case op of
  1        2oldH(hi′,l,hi″) → hi′ ∈ iohs(hs)∨hi″isin iohs(hs),
  2        1oldH1newH(hi,l,h) → hi ∈ iohs(hs)∨h∉ hs∨obs_HI(h)∉ iohs(hs),
  3        2newH(h′,l,h″) → {h′,h″}∩ hs={}∨{obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}
           end
```

"slide 1386"

K.3 Review

K.4 Exercises

"slide 1387"

Exercise 49. 71:
Solution 49 Vol. II, Page 538, suggests a way of answering this exercise.

Exercise 50. 72:
Solution 50 Vol. II, Page 538, suggests a way of answering this exercise.

"slide 1388"

L

Postlude Domain Engineering Actions

"slide 1390"

L.1 Domain Verification

"slide 1391"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.10 (Page 102).

L.2 Domain Validation

"slide 1392"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.11 (Page 102).

L.3 Towards a Domain Teory of Transportation

"slide 1393"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 2, Sect. 2.13 (Page 102).

L.4 Review

"slide 1394"

L.5 Exercises

"slide 1395"

Exercise 51. 001:
Solution 51 Vol. II, Page 538, suggests a way of answering this exercise.

Exercise 52. 002:
Solution 52 Vol. II, Page 538, suggests a way of answering this exercise.

"slide 1396"

Part VI

Requirements Engineering

Chapter 3 (Pages 109–166) covered methodology issues of requirements engineering.

This paet cosists of Appendices M–Q (Pages 461–493):

- Appendix M covers initial stages of requirements engineering:
  - ⋆ Informative Documents
  - ⋆ Requirements Stakeholders
  - ⋆ Requirements Acquisition
  - ⋆ Requirements Analysis and Concept Formation
  - ⋆ Business Process Re-engineering
  - ⋆ Requirements Terminology
- Appendices N–P cover requirements modelling in three stages:
  - ⋆ Appendix N covers domain requirements,
  - ⋆ Appendix O covers interface requirements, and
  - ⋆ Appendix P covers machine requirements.
- Appendix Q covers concluding stages of requirements engineering:
  - ⋆ Requirements Verification
  - ⋆ Requirements Validation
  - ⋆ Requirements Feasibility and Satisfiability
  - ⋆ (Towards a) Requirements Theory

M

Prelude Requirements Engineering Actions "slide 1398"

M.1 Informative Requirements Documents "slide 1399"

Recall, from Page 7, that information documents include:

M.1.1 Project Name and Dates "slide 1400"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.1, Page 8.

Project Name and Dates

- **Project Name:** *PtP TollRITS:*
  Requirements for a Point-to-Point Toll Road IT Support System
- **Dates:** Summer 2008 – fall 2009

M.1.2 Project Partners and Places

"SLIDE 1401"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.2, Page 8.

PtP TollRITS: Project Partners and Places

- **Client:**
  - ⋆ *Institution: ...*
  - ⋆ *Address: ...*
- **Developer:**
  - ⋆ *Company: ...*
  - ⋆ *Address: ...*

M.1.3 Current Situation

"SLIDE 1402"

PtP TollRITS: Current Situation

- Currently the road net has no toll roads, but toll road systems are being considered.
- Currently such tool roads are thought of as point-to-point, that is:
  - ⋆ End points connect to an existing road net; and
  - ⋆ between the two end-points there may be zero, one or more intermediate connections to that existing road net.

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.3, Page 9.

M.1.4 Needs and Ideas

"SLIDE 1403"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.4, Page 9.

PtP TollRITS: Needs and Ideas

- **Needs:**
  - ⋆ The needs are for such a toll road system to
    - · have a number of phenomema and concepts monitored and controlled by an IT system, and
    - · have that IT system be domain described, requirements prescribed and, eventually designed
    - · in a trustworthy manner.
- **Ideas:**
  - ⋆ The idea, for this phase of requirements development.

- ⋆ is to provide a requirements, "derived" from an existing domain description
- ⋆ according to the TripTych dogma.

M.1.5 Concepts and Facilities

"SLIDE 1404"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.5, Page 10.

To bring the concepts and facilities of the toll road system in context we show and instance of such a toll road system in Fig. M.1.

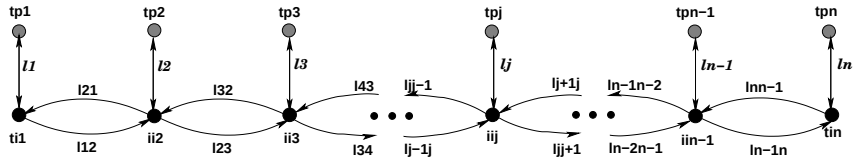


Fig. M.1. A simple, linear toll road net:
tp_i: *t*oll *p*laza i,
ti_1, ti_n: *t*erminal *i*ntersection k,
ii_k: *i*ntermediate *i*ntersection k, $1<k<n$
ℓ_i: toll plaza *l*ink i,
l_{xy}: tollway *l*ink from i_x to i_y, $y=x+1$ or $y=x\text{-}1$ and $1\leq x<n$.

"slide 1405"

PtP TollRITS: Concepts and Facilities

The concepts and facilities of the point-to-point toll road system being contemplated should become clear from the Business Process Re-engineering section, Sect. M.5 Pages 467–469, check with Fig. M.1 (Page 463)

Some (main) entities of the point-to-point toll road system are: (i) "tickets", (ii) toll plazas — toll plazas have one or more "ticket" issuing and one or more "ticket" and, subsequently, fee collecting, i.e., (iii-iv) entry, respectively exit toll booths, (v) terminal (end-point) intersections, (vi) intermediate (in-between) intersections, (vii) toll plaza links, (viii) toll way links, etcetera.

PG.:1406

Some functions of the point-to-point toll road system are: (ix) outside car enters toll road system by entering toll plaza entry booth, collects ticket, becomes inside car and enters toll plaza link; (x) inside car leaves toll plaza link and enters intersections and toll ways; (xi) inside car leaves toll ways and intersections and enters toll plaza link; and (xi) inside car leaves toll plaza link, enters toll plaza exit booth and delivers ticket and fees, and leaves toll road system, etcetera.

PG.:1407

Some events of the point-to-point toll road system are: (xii) an outside car appears at an entry booth, but is hindered to proceed onto the connecting toll plaza link; (xiii) a car accepts an antry booth ticket and is allowed to proceed onto the connecting toll plaza link; (xvi) a car leaves the entry booth and enters the connecting toll plaza link; (xv) an inside car appears at an exit booth, but is hindered to proceed beyond the toll road system; (xvi) an inside car presents a "ticket" to the exit booth, but is still hindered in proceeding beyond the toll road system; (xvii) an inside car is presented with a fee to pay, but is still hindered in proceeding beyond the toll road system; (xviii) an inside car presents an appropriate fee and is allowed to proceed beyond the toll road system; (xix) an inside car leaves the exit booth and thus leaves the toll road system; etcetera. PG.:1408

Some behaviours of the point-to-point toll road system are: (xx) a car enters the toll road system, obtains a ticket; proceeds onto a connecting toll plaza link, enters an intersection, proceeds onto a toll way link, and, in succession, enters zero, one or more intermediate intersections and connected toll way links until, at either an intermediate intersection or at an end intersection it enters the connecting toll plaza link, presents the ticket at an exit booth, pays the requested fee and leaves the toll road system.; (xxi) a toll plaza entry booth successively alternates between issuing tickets, allowing a car to proceed at a time and stopping cars from proceeding; (xxii) a toll plaza exit booth successively alternates between acceptinh tickets, posting fees, accepting fees, allowing a car to proceed at a time and stopping cars from proceeding; etcetera.

M.1.6 Scope and Span

"SLIDE 1409"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.6, Page 11.

PtP TollRITS: Scope

- The scope of the toll road system is that of the domain described road transportation system.

PtP TollRITS: Span

- The span of the toll road system is a subset of the domain described road transportation system — one that excludes insertion and removal of links, transport timetables, and many other things.

M.1.7 Assumptions and Dependencies "slide 1410"

PtP TollRITS: Assumptions

- There is an already existing, accessible road transport domain description, and
- there is access to appropriate requirements stakeholders.

PtP TollRITS: Dependencies

-
-

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.7, Page 11.

TO BE WRITTEN

M.1.8 Implicit/Derivative Goals "slide 1411"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.8, Page 12.

TO BE WRITTEN

M.1.9 Concepts and Facilities "slide 1412"

M.1.10 Synopsis "slide 1413"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.9, Page 13.

TO BE WRITTEN

M.1.11 Software Development Graphs "slide 1414"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.10, Page 13.

TO BE WRITTEN

M.1.12 Resource Allocation "slide 1415"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.11, Page 15.

TO BE WRITTEN

M.1.13 Budget Estimate

"SLIDE 1416"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.12, Page 16.

M.1.14 Standards Compliance

"SLIDE 1417"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.13, Page 16.

TO BE WRITTEN

M.1.15 Contracts and Design Briefs

"SLIDE 1418"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.14, Page 19.

TO BE WRITTEN

M.1.16 Logbook

"SLIDE 1419"

For the motivation and the principles and techniques for carrying out this step of development we refer to Chap. 1, Sect. 1.6.15, Page 23.

TO BE WRITTEN

M.2 Requirements Stakeholder Identification

"SLIDE 1420"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.4 (Page 119).

TO BE WRITTEN

M.3 Requirements Acquisition

"SLIDE 1421"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.5 (Page 119).

TO BE WRITTEN

M.4 Requirements Analysis and Concept Formation "slide 1422"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.6 (Page 121).

TO BE WRITTEN

M.5 Business Process Re-engineering "slide 1423"

We omit frames around the "running" *PtP TollRITS* example.

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.7 (Page 121).

M.5.1 The Example Requirements

The domain was that of transportation. The requirements is now basically related to the issuance of tickets upon vehicle entry to a toll road net[1] and payment of tickets upon the vehicle leaving the toll road net both issuance and collection/payment of tickets occurring at toll booths[2] which are hubs somehow linked to the toll road net proper. Add to this that vehicle tickets are sensed and updated whenever the vehicle crosses an intermediate toll road intersection. "slide 1424"

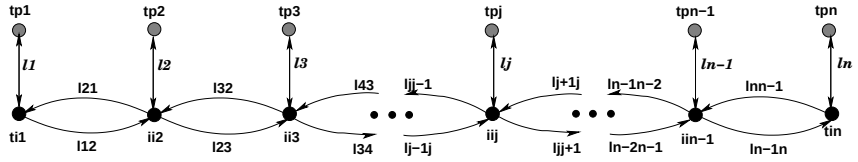


Fig. M.2. A simple, linear toll road net:
tp_i: *t*oll *p*laza i,
ti_1, ti_n: *t*erminal *i*ntersection k,
ii_k: *i*ntermediate *i*ntersection k, $1<k<n$
l_{xy}: tollway *l*ink from i_x to i_y, $y=x+1$ or $y=x\text{-}1$ and $1\leq x<n$.

"slide 1425"

Business process re-engineering (BPR) re-evaluates the intrinsics, support technologies, management & organisation, rules & regulations, scripts, and human behaviour facets while possibly changing some or all of these, that is, possibly rewriting the corresponding parts of the domain description.

[1] Toll road: in other forms of English; tollway, turnpike, pike or toll-pike, in French péage.

[2] Toll plazas, toll stations, or toll gates

Re-engineering Domain Entities "slide 1426"

The net is arranged as a linear sequence of two or more (what we shall call) intersection hubs. Each intersection hub has a single two-way link to (what we shall call) an entry/exit hub (toll plaza); and each intersection hub has either two or four one-way (what we shall call) tollway links: the first and the last intersection hub (in the sequence) has two tollway links and all (what we shall call) intermediate intersections has four tollway links. We introduce a pragmatic notion of net direction: "up" and "down" the net, "from one end to the other". This is enough to give a hint at the re-engineered domain.

Re-engineering Domain Operations "slide 1427"

We first briefly sketch the tollgate Operations. Vehicles enter and leave the tollway net only at entry/exit hubs (toll plazas). Vehicles collect and return their tickets from and to tollgate ticket issuing, respectively payment machines. Tollgate ticket-issuing machines respond to sensor pressure from "passing" vehicles or by vehicle drivers pressing ticket-issuing machine buttons. Tollgate payment machines accept credit cards, bank notes or coins in designated currencies as payment and returns any change.

"slide 1428"

We then briefly introduce and sketch an operation performed when vehicles cross intersections: The vehicle is assumed to possess the ticket issued upon entry (in)to the net (at a tollgate). At the crossing of each intersection, by a vehicle, its ticket is sensed and is updated with the fact that the vehicle crossed the intersection.

The updated domain description section on support technology will detail the exact workings of these tollgate and internal intersection machines and the domain description section on human behaviour will likewise explore the man/machine facet.

Re-engineering Domain Events "slide 1429"

The intersections are highway-engineered in such a way as to deter vehicle entry into opposite direction tollway links, yet, one never knows, there might still be (what we shall call ghost) vehicles, that is vehicles which have somehow defied the best intentions, and are observed moving along a tollway link in the wrong direction.

Re-engineering Domain Behaviours "slide 1430"

The intended behaviour of a vehicle of the tollway is to enter at an entry hub (collecting a ticket at the toll gate), to move to the associated intersection, to move into, where relevant, either an upward or a downward tollway link, to proceed (i.e., move) along a sequence of one or more tollway links via

connecting intersections, until turning into an exit link and leaving the net at an exit hub (toll plaza) while paying the toll.

• • •

This should be enough of a BPR rough sketch for us to meaningfully proceed to requirements prescription proper.

M.6 Requirements Terminology

"SLIDE 1431"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.8 (Page 127).

TO BE WRITTEN

M.7 Exercises

"SLIDE 1432"

Exercise 53. Reqs. Prelude 1:
Solution 53 Vol. II, Page 538, suggests a way of answering this exercise.

Exercise 54. Reqs. Prelude 2:
Solution 54 Vol. II, Page 538, suggests a way of answering this exercise.

"SLIDE 1433"

N

Domain Requirements

"SLIDE 1435"

In this appendix we shall show examples of requirements for two transportation systems. Both are based on the domain model given in earlier appendices. The two examples will intertwain in this appendix: a smaller example, always shown in frames, RoMaS*: Road Maintenance System*, and a larger example, not shown in frames, PtPToll*: Point-to-Point Toll Road IT System*. This latter example was already strongly hinted at in Sect. M.5 (business process reengineering).

N.1 Domain Projection

"SLIDE 1436"

N.1.1 RoMaS: A Road Maintenance System

Narrative

RoMaS: Road Maintenance System, Narrative

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.

Formalisation[1]

RoMaS: Road Maintenance System, Formalisation

type

[1] "SLIDE 1437"

```
    5 on page 343:  H, L,
    6 on page 344:  N = H-set × L-set
axiom
    6 on page 344:  ∀ (hs,ls):N • card hs≥2 ∧ card ks≥1
type
    7 on page 344:  HI, LI
value
    8 on page 345a:  obs_HI: H → HI, obs_LI: L → LI
axiom
    8 on page 345b:  ∀ h,h′:H, l,l′:L •
       h≠h′⇒obs_HI(h)≠obs_HI(h′) ∧ l≠l′⇒obs_LI(l)≠obs_LI(l′)
value
    9 on page 345a:  obs_HIs: L → HI-set
    10 on page 345a:  obs_LIs: H → LI-set
```

PG.:1438

```
    9 on page 345b:  ∀ l:L • card obs_HIs(l)=2 ∧
    10 on page 345b:  ∀ h:H • card obs_LIs(h)≥1 ∧
    ∀ (hs,ls):N •
    9(a on page 345):      ∀ h:H • h ∈ hs ⇒ ∀ li:LI • li ∈ obs_LIs(h) ⇒
              ∃ l′:L • l′ ∈ ls ∧ li=obs_LI(l′) ∧ obs_HI(h) ∈ obs_HIs(l′) ∧
    10(a on page 345):      ∀ l:L • l ∈ ls ⇒
              ∃ h′,h″:H • {h′,h″}⊆hs ∧ obs_HIs(l)={obs_HI(h′),obs_HI(h″)}
    11 on page 345:  ∀ h:H • h ∈ hs ⇒ obs_LIs(h) ⊆ iols(ls)
    12 on page 345:  ∀ l:L • l ∈ ls ⇒ obs_HIs(h) ⊆ iohs(hs)
value
   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}
```

N.1.2 PtPToll: Toll Road IT System

"SLIDE 1439"

Narrative

For the 'Toll Road IT System', in addition to what was projected for the 'Road Management System', the following entities and most related functions are projected:

- hubs, links, hub identifiers and link identifiers;
- nets,
- hub state and hub state spaces and
- link states and link state spaces;
- corresponding observer functions and

- corresponding axioms and syntactic and semantic wellformedness predicates.

Formalisation[2]

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 RoMaS projection; and for the missing axioms and wellformedness predicates see Sect. G.1 Pages 367–370.

N.2 Domain Instantiation

"slide 1441"

N.2.1 RoMaS: Road Maintenance System

Narrative

RoMaS: Instantiation, Narrative

220 The road net consist of a sequence of one or more road segments.
221 A road segment can be characterised by a pair of hubs and a pair of links conected to these hubs.
222 Neighbouring road segments share a hub.
223 All hubs are otherwise distinct.
224 All links are distinct.
225 The two links of a road segment connects to the hubs of the road segment.
226 We can show that road nets are specific instances of concretisations of the former, thus more absttract road nets.

[2] "slide 1440"

Formalisation[3]

RoMaS: Instantiation, Formalisation

```
type
220 RN = RS*,
221 RS = H × (L × L) × H
axiom
        ∀ rn:RN •
222 ∀ i:Nat • {i,i+1}⊆inds rn ⇒
                  let (_,_,h)=rn(i),(h′,_,_)=rn(i+1) in h=h′ end ∧
223 len rn + 1 = card{h,h′|h,h′:H•(h,_,h′)∈ elems rn} ∧
224 2∗(len rn) = card{l,l′|l,l′:L•(_,(l,l′),_)∈ elems rn} ∧
225 ∀ (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
226   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})
```

N.2.2 PtPToll: Toll Road IT System

"slide 1443"

We omit frames around the 'running' PtPToll example:

Narrative

The 1st version domain requirements prescription is now updated with respect to the properties of the toll way net: We refer to Fig. M.2 and the preliminary description given in Sect. M.5.1. 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. 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.

"slide 1444"

"slide 1445"

[3] "slide 1442"

Formalisation[4]

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** hlll}
  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** hlll}

theorem ∀ tn:TN • wf_TN(tn) ⇒ wf_N(abs_N(tn))

"slide 1447"

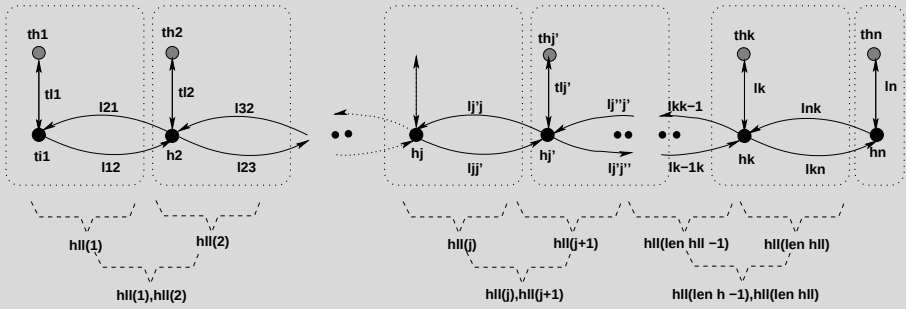


Fig. N.1. A simple, linear toll road net:
th_i: *t*oll *p*laza *i*,
h_1, h_n: *t*erminal *i*ntersections,
h_2, h_j, h'_j, h_k: *i*ntermediate *i*ntersections, $1<j\leq k$, $k=n$-1
l_{xy}, l_{yx}: tollway *l*ink from h_x to h_y and from h_y to h_x, $1\leq x<n$.
l_{x-1x}, l_{xx-1}: tollway *l*ink from h_{x-1} to h_x and h_x to h_{x-1}, $1\leq x<n$,
dashed links are not in formulas.

Formalisation of Well-formedness[5]

type

[4] "SLIDE 1446"
[5] "SLIDE 1448"

```
  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 ]
```

"slide 1449"

```
value
  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
```

"slide 1450"

```
value
  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
```

N.3 Domain Determination

"slide 1451"

N.3.1 RoMaS: Road Management System

We shall, in this example, claim that the following items constitute issues of more determinate nature for RoMaS. fixing the states of links and hubs; endowing links and hubs with such attributes as road surface material (concrete, asfalt, 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.

Narrative[6]

RoMaS: State Space Determination, Narrative

227 The two links of a road segment are open for traffic in one direction and in opposite directions only.
228 Hubs are always in the same state, namely one that allows traffic from incoming links to continue onto all outgoing links.
229 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.

Formalisation[7]

RoMaS: State Space Determination, Formalisation

```
axiom
  ∀ rn:RN •
227 ∀ (h,(l,l′),h′):RS • (h,(l,l′),h′) ∈ elems rn ⇒
      obs_LΣ(l) = {(obs_HI(h),obs_LI(l),obs_HI(h′))} ∧
      obs_LΣ(l′) = {(obs_HI(h′),obs_LI(l′),obs_HI(h))} ∧
228 ∀ 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_LI(l),obs_HI(h),obs_LI(l′))},
        len rn → obs_HΣ(h′) = {(obs_LI(l′),obs_HI(h′),obs_LI(l))},
        _ → obs_HΣ(h′) = {(obs_LI(l),obs_HI(h′),obs_LI(l′)),(obs_LI(l),obs_HI(h′),obs_LI(l′))}
      end end
type
229 Surface, WearTear, Area, OrdTime, OrdCost, RepTime, RepCost, ...
value
229 obs_Surface: (H|L)→Surface, obs_WearTear: (H|L)→WearTear, ...
```

N.3.2 PtPToll: Toll Road IT System

"slide 1454"

Narrative

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

[6] "slide 1452"
[7] "slide 1453"

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.

Formalisation[8]

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**

"slide 1456"

wf_Plaza(th,tl,h) ≡
  obs_HΣ(th) = [crossings at toll plazas]
    {(″external″,obs_HI(th),obs_LI(tl)),
      (obs_LI(tl),obs_HI(th),″external″),
      (obs_LI(tl),obs_HI(th),obs_LI(tl))} ∧
    obs_HΩ(th)={obs_HΣ(th)} ∧
    obs_LΩ(tl) = {obs_LΣ(tl)}

wf_End(h,tl,l,l′) ≡
  obs_HΣ(h) = [crossings at 3−link end hubs]
    {(obs_LI(tl),obs_HI(h),obs_LI(tl)),(obs_LI(tl),obs_HI(h),obs_LI(l)),
     (obs_LI(l′),obs_HI(h),obs_LI(tl)),(obs_LI(l′),obs_HI(h),obs_LI(l))} ∧
  obs_HΩ(h) = {obs_HΣ(h)} ∧
  obs_LΩ(l) = {obs_LΣ(l)} ∧ obs_LΩ(l′) = {obs_LΣ(l′)}

"slide 1457"

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)}

[8] "SLIDE 1455"

Not all determinism issues above have been fully explained. But for now we should — in principle — be satisfied.

N.4 Domain Extension "slide 1458"

N.4.1 RoMaS: Road Management System

Narrative

Formalisation[9]

N.4.2 PtPToll: Toll Road IT System "slide 1460"

Narrative

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; 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. "slide 1461"

One added (extended) domain requirements: as vehicles are allowed to cruise the entire net payment is a function of the totality of links traversed, possibly multiple times. This requires, in our case, that tickets be made such as to be sensed somewhat remotely, and that 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.

Formalisation[10]

We suggest only some signatures:

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)

[9] "slide 1459"
[10] "slide 1462"

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]

N.4.3 Discussion

"SLIDE 1463"

This example provides a classical requirements engineering setting for embedded, safety critical, real-time systems, requiring, ultimately, the techniques and tools of such things as Petri nets, statecharts, message sequence charts or live sequence charts and temporal logics (DC, TLA+).

N.5 Requirements Fitting

"SLIDE 1464"

N.5.1 RoMaS & PtPToll Narrative

We postulate two domain requirements: We have outlined a domain requirements development, RoMaS, for software support for road maintenance; and we have outlined a domain requirements development, for PtPToll, software support for a toll road (IT) system. We can therefore postulate that there are two domain requirements developments, both based on the transport domain: one, $d_{r_{\text{RoMaS}}}$, for a toll road computing system monitoring and controlling vehicle flow in and out of toll plazas, and another, $d_{r_{\text{PtPToll}}}$, for a toll link and intersection (i.e., hub) building and maintenance system monitoring and controlling link and hub quality and for development.

"slide 1465"

The fitting procedure now identifies the shared of awareness of the net by both $d_{r_{\text{RoMaS}}}$ and $d_{r_{\text{PtPToll}}}$ 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, DB_{rel}, may already be available with the customer.

"slide 1466"

In any case, where there before were two requirements $(d_{r_{\text{RoMaS}}}, d_{r_{\text{PtPToll}}})$ there are now four: (i) $d'_{r_{\text{RoMaS}}}$, a modification of $d_{r_{\text{RoMaS}}}$ which omits the description parts pertaining to the net; (ii) $d'_{r_{\text{PtPToll}}}$, a modification of $d_{r_{\text{PtPToll}}}$ which likewise omits the description parts pertaining to the net; (iii) $d_{r_{\text{net}}}$, which contains what was basically omitted in $d'_{r_{\text{RoMaS}}}$ and $d'_{r_{\text{PtPToll}}}$; 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.

N.5.2 RoMaS & PtPToll Formalisation

"slide 1467"

We leave [lengthy] formalisation as an exercise !

N.6 Requirements Consolidation

"slide 1468"

TO BE WRITTEN

N.7 Exercises

"slide 1469"

Exercise 55. 81:
Solution 55 Vol. II, Page 539, suggests a way of answering this exercise.

Exercise 56. 82:
Solution 56 Vol. II, Page 539, suggests a way of answering this exercise.

"slide 1470"

O

Interface Requirements

"SLIDE 1472"

O.1 Shared Entities

"SLIDE 1473"

The main shared entities are the net, hence the hubs and the links. As domain entities they continuously undergo changes with respect to the values of a great number of attributes and otherwise possess attributes — most of which have not been mentioned so far: length, cadestral information, namings, wear and tear (where-ever applicable), last/next scheduled maintenance (where-ever applicable), state and state space, and many others. "slide 1474"

We "split" our interface requirements development into two separate steps: the development of $d_{r_{\text{net}}}$ (the common domain requirements for the shared hubs and links), and the co-development of $d_{r_{\text{db:i/f}}}$ (the common domain requirements for the interface between $d_{r_{\text{net}}}$ and DB_{rel} — under the assumption of an available relational database system DB_{rel} "slide 1475"

When planning the common domain requirements for the net, i.e., the hubs and links, we enlarge our scope of requirements concerns beyond the two so far treated ($d_{r_{\text{toll}}}$, $d_{r_{\text{maint.}}}$) in order to make sure that the shared relational database of nets, their hubs and links, may be useful beyond those requirements. We then come up with something like hubs and links are to be represented as tuples of relations; each net will be represented by a pair of relations a hubs relation and a links relation; each hub and each link may or will be represented by several tuples; etcetera. In this database modelling effort it must be secured that "standard" operations on nets, hubs and links can be supported by the chosen relational database system DB_{rel}. "slide 1476"

O.1.1 Data Initialisation

As part of $d_{r_{\text{net}}}$ one must prescribe data initialisation, that is provision for an interactive user interface dialogue with a set of proper display screens, one for establishing net, hub or link attributes (names) and their types and, for example, two for the input of hub and link attribute values. Interaction

prompts may be prescribed: next input, on-line vetting and display of evolving net, etc. These and many other aspects may therefore need prescriptions.

"slide 1477" Essentially these prescriptions concretise the insert link operation.

O.1.2 Data Refreshment

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

These prescriptions concretise remove and insert link operations.

O.2 Shared Operations "slide 1478"

The main shared operations are related to the entry of a vehicle into the toll "slide 1479" road system and the exit of a vehicle from the toll road system.

O.2.1 Interactive Operation Execution

As part of $d_{r_{\text{toll}}}$ we must therefore prescribe the varieties of successful and less successful sequences of interactions between vehicles (or their drivers) and the toll gate machines.

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

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

O.3 Shared Events "slide 1480"

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

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

The prescription of the above necessitates extensive fault analysis.

O.4 Shared Behaviours

"SLIDE 1481"

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

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

O.5 Exercises

"SLIDE 1482"

Exercise 57. 91:
Solution 57 Vol. II, Page 539, suggests a way of answering this exercise.

Exercise 58. 92:
Solution 58 Vol. II, Page 539, suggests a way of answering this exercise.

"SLIDE 1483"

P

Machine Requirements

"SLIDE 1485"

P.1 Performance Requirements

"SLIDE 1486"

"slide 1487"

P.1.1 Machine Storage Consumption

"slide 1488"

P.1.2 Machine Time Consumption

"slide 1489"

P.1.3 Other Resource Consumption

P.2 Dependability Requirements

"slide 1490"

"slide 1491"

P.2.1 Accesability Requirements

"slide 1492"

P.2.2 Availability Requirements

"slide 1493"

P.2.3 Integrity Requirements

"slide 1494"

P.2.4 Reliability Requirements

"slide 1495"

P.2.5 Safety Requirements

"slide 1496"

P.2.6 Security Requirements

P.3 Maintenance Requirements

"SLIDE 1497"

"slide 1498"

P.3.1 Adaptive Maintenance Requirements

"slide 1499"

P.3.2 Corrective Maintenance Requirements

"slide 1500"

P.3.3 Perfective Maintenance Requirements

"slide 1501"

P.3.4 Preventive Maintenance Requirements

P.4 Platform Requirements

"SLIDE 1502"

"slide 1503"

P.4.1 Development Platform Requirements

"slide 1504"

P.4.2 Execution Platform Requirements

"slide 1505"

P.4.3 Maintenance Platform Requirements

"slide 1506"

P.4.4 Demonstration Platform Requirements

P.5 Development Documentation Requirements "slide 1507"

"slide 1508"

P.5.1 Informative Documents

"slide 1509"

P.5.2 Specification Documents

"slide 1510"

P.5.3 Analytic Documents:

"slide 1511"

P.5.4 Installation Documentation

"slide 1512"

P.5.5 Demonstration Documentation

"slide 1513"

P.5.6 User Documentation

"slide 1514"

P.5.7 Maintenance Documentation

"slide 1515"

P.5.8 Disposal Documentation

P.6 Summary "slide 1516"

P.7 Exercises "slide 1517"

Exercise 59. 101:
Solution 59 Vol. II, Page 539, suggests a way of answering this exercise.

Exercise 60. 102:
Solution 60 Vol. II, Page 539, suggests a way of answering this exercise.

"slide 1518"

“slide 1519”

Q

Postlude Requirements Engineering Actions

"SLIDE 1520"

Q.1 Requirements Verification

"SLIDE 1521"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.10 (Page 163).

Q.2 Requirements Validation

"SLIDE 1522"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.11 (Page 163).

Q.3 Requirements Satisfiability and Feasibility

"SLIDE 1523"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.12 (Page 163).

Q.4 Towards a Requirements Teory of Transportation

"SLIDE 1524"

For the motivation and the principles and techniques for carrying out this stage of development we refer to Chap. 3, Sect. 3.13 (Page 163).

Q.5 Review

"SLIDE 1525"

Q.6 Exercises

"SLIDE 1526"

Exercise 61. Reqs. Postludium 1:
Solution 61 Vol. II, Page 539, suggests a way of answering this exercise.

Exercise 62. Reqs. Postludium 2:
Solution 62 Vol. II, Page 539, suggests a way of answering this exercise.

"SLIDE 1527"

Part VII

Software Design

R

Software Design "slide 1529"

R.1 Informative Software Design Documents "slide 1530"

R.1.1 Project Name and Dates "slide 1531"

R.1.2 Project Places "slide 1532"

R.1.3 Project Partners "slide 1533"

R.1.4 Current Situation "slide 1534"

R.1.5 Needs and Ideas "slide 1535"

R.1.6 Concepts and Facilities "slide 1536"

R.1.7 Scope and Span "slide 1537"

R.1.8 Assumptions and Dependencies "slide 1538"

R.1.9 Implicit/Derivative Goals "slide 1539"

R.1.10 Synopsis "slide 1540"

R.1.11 Software Development Graphs "slide 1541"

R.1.12 Resource Allocation "slide 1542"

R.1.13 Budget Estimate "slide 1543"

R.1.14 Standards Compliance "slide 1544"

R.1.15 Contracts and Design Briefs "slide 1545"

R.1.16 Logbook "slide 1546"

R.2 Software Design Stakeholder Identification "slide 1547"

R.3 Software Design Acquisition "slide 1548"

R.4 Software Design Analysis and Concept Formation

"slide 1549"

R.5 Software Design "BPR" "slide 1550"

R.6 Software Design Terminology "slide 1551"

R.7 Software Design Modelling "slide 1552"

"slide 1553"

R.7.1 Architectural Design

"slide 1554"

R.7.2 Component Design

"slide 1555"

R.7.3 Module Design

"slide 1556"

R.7.4 Coding

"slide 1557"

R.7.5 Programming Paradigms

"slide 1558"

Extreme Programming

"slide 1559"

Aspect-oriented Programming

"slide 1560"

Intensional Programming

"slide 1561"

??? Programming

"slide 1562"

Version Control & Configuration Management

R.8 Software Design Verification

"SLIDE 1563"

R.9 Software Design Validation

"SLIDE 1564"

R.10 Software Design Release, Transfer and Maintenance

"SLIDE 1565"

"slide 1566"

R.10.1 Software Design Release

"slide 1567"

R.10.2 Software Design Transfer

"slide 1568"

R.10.3 Software Design Maintenance

R.11 Software Design Documentation

"SLIDE 1569"

"slide 1570"

R.11.1 Software Design Process Graph

"slide 1571"

R.11.2 Software Design Documents

R.12 Software Design

"SLIDE 1572"

R.13 Exercises

"SLIDE 1573"

Exercise 63. 111:
Solution 63 Vol. II, Page 540, suggests a way of answering this exercise.

Exercise 64. 112:
Solution 64 Vol. II, Page 540, suggests a way of answering this exercise.

"SLIDE 1574"

Part VIII

RAISE

S

An RSL Primer

"SLIDE 1576"

This is an ultra-short introduction to the RAISE Specification Language, RSL.

S.1 Types

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

S.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. "slide 1577"

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".

"slide 1578"

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 × B × ... × C
[10] A*
[11] A$^\omega$
[12] A $\xrightarrow[m]{}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1 The Boolean type of truth values **false** and **true**.
2 The integer type on integers ..., –2, –1, 0, 1, 2, ...
3 The natural number type of positive integer values 0, 1, 2, ...
4 The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5 The character type of character values ″a″, ″b″, ...
6 The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7 The set type of finite cardinality set values.
8 The set type of infinite and finite cardinality set values.
9 The Cartesian type of Cartesian values.
10 The list type of finite length list values.
11 The list type of infinite and finite length list values.
12 The map type of finite definition set map values.
13 The function type of total function values.
14 The function type of partial function values.
15 In (A) A is constrained to be:
   - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,

- or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\xrightarrow[m]{}$ 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.

"slide 1579"

S.1.2 Type Definitions

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
```

"slide 1580"

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) |}
```

"slide 1581"

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**

"slide 1582"

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 • $\mathcal{P}$(b) |}

"slide 1583"

Sorts — Abstract Types

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

Sorts

type
  A, B, ..., C

S.2 The RSL Predicate Calculus

"SLIDE 1584"

S.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 (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*. "slide 1585"

S.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

are simple predicate expressions. "slide 1586"

S.2.3 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.

S.3 Concrete RSL Types: Values and Operations "slide 1587"

S.3.1 Arithmetic

Arithmetic

type

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


S.3.2 Set Expressions

Set Enumerations

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

Set Enumerations

```
  {{}, {a}, {e1,e2,...,en}, ...} ∈ A-set
  {{}, {a}, {e1,e2,...,en}, ..., {e1,e2,...}} ∈ A-infset
```


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

S.3.3 Cartesian Expressions

Cartesian Enumerations

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

Cartesian Enumerations

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

"slide 1591"

S.3.4 List Expressions

List Enumerations

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

List Enumerations

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

$\langle$ a_i .. a_j $\rangle$

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

"slide 1592"

List Comprehension

The last line below expresses list comprehension.

List Comprehension

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

"slide 1593"

S.3.5 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
```


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) ]
```


S.3.6 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** $\xrightarrow{\sim}$ **Nat**

"slide 1596"

Set Examples

Set Examples

examples
a ∈ {a,b,c}
a ∉ {}, a ∉ {b,c}
{a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
∪{{a},{a,b},{a,d}} = {a,b,d}
{a,b,c} ∩ {c,d,e} = {c}
∩{{a},{a,b},{a,d}} = {a}
{a,b,c} \ {c,d} = {a,b}
{a,b} ⊂ {a,b,c}
{a,b,c} ⊆ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} ≠ {a,b}
card {} = 0, **card** {a,b,c} = 3

"slide 1597"

Informal Explication

19 ∈: The membership operator expresses that an element is a member of a set.
20 ∉: The nonmembership operator expresses that an element is not a member of a set.
21 ∪: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22 ∪: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23 ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.

24 ∩: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
"slide 1598"
25 \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26 ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27 ⊂: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28 =: The equal operator expresses that the two operand sets are identical.
29 ≠: The nonequal operator expresses that the two operand sets are *not* identical.
30 **card**: The cardinality operator gives the number of elements in a finite set.
"slide 1599"

Set Operator Definitions

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

Set Operation Definitions

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

"slide 1600"

S.3.7 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
```

"slide 1601"

S.3.8 List Operations

List Operator Signatures

List Operations

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ⌢: A* × A^ω → A^ω
  =: A^ω × A^ω → Bool
  ≠: A^ω × A^ω → Bool
```

"slide 1602"

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⟩

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.

- $\frown$: 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.

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″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

"slide 1606"

S.3.9 Map Operations

Map Operator Signatures and Map Operation Examples

Map Operations

```
value
  m(a): M → A ~→ B, m(a) = b

  dom: M → A-infset [domain of map]
    dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

  rng: M → B-infset [range of map]
    rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

  †: M × M → M [override extension]
    [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

  ∪: M × M → M [merge ∪]
    [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]
```

\: M × A-**infset** → M [restriction by]
[a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]

/: M × A-**infset** → M [restriction to]
[a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

=,≠: M × M → **Bool**

°: (A $\overrightarrow{m}$ B) × (B $\overrightarrow{m}$ C) → (A $\overrightarrow{m}$ C) [composition]
[a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

"slide 1607"

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. "SLIDE 1608"
- \: 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.

"slide 1609"

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
```

S.4 λ-Calculus + Functions

"SLIDE 1610"

S.4.1 The λ-Calculus Syntax

λ-Calculus Syntax

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

"slide 1611"

S.4.2 Free and Bound Variables

Free and Bound Variables

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

- $\langle$V$\rangle$: Variable x is free in x.
- $\langle$F$\rangle$: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- $\langle$A$\rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

"slide 1612"

S.4.3 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](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),
  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),
  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

"slide 1613"

S.4.4 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)

"slide 1614"

S.4.5 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
```

"slide 1615"

S.4.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions

```
value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments ~→ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)
```

"slide 1616"

Or functions can be defined implicitly:

Implicit Function Definitions

```
value
  f: Arguments → Result
  f(args) as result
  post P1(args,result)

  g: Arguments ~→ Result
  g(args) as result
  pre P2(args)
  post P3(args,result)
```

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

S.5 Other Applicative Expressions

"SLIDE 1617"

S.5.1 Simple let Expressions

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

Let Expressions

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

is an "expanded" form of:

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

"slide 1618"

S.5.2 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 = **Y**F **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

"slide 1619"

S.5.3 Predicative let Expressions

Predicative **let** expressions:

Predicative **let** Expressions

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

"slide 1620"

S.5.4 Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

Patterns

```
let {a} ∪ s = set in ... end
let {a,_} ∪ s = set in ... end

let (a,b,...,c) = cart in ... end
let (a,_,...,c) = cart in ... end

let ⟨a⟩^ℓ = list in ... end
let ⟨a,_,b⟩^ℓ = list in ... end

let [ a↦b ] ∪ m = map in ... end
let [ a↦b,_ ] ∪ m = map in ... end
```

"slide 1621"

S.5.5 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
```

"slide 1622"

S.5.6 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⟩ ::= !

S.6 Imperative Constructs

"SLIDE 1623"

S.6.1 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**.

"slide 1624"

S.6.2 Variables and Assignment

Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

S.6.3 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

S.6.4 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**

"slide 1625"

S.6.5 Iterative Conditionals

Iterative Conditionals

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

S.6.6 Iterative Sequencing

Iterative Sequencing

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

S.7 Process Constructs

"SLIDE 1626"

S.7.1 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 }
```

"slide 1627" declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

S.7.2 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 are forced to communicate only "slide 1628" with one another, until one of them terminates.

S.7.3 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

"slide 1629" expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

S.7.4 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() ≡ ... c ? ... k[i] ! e ...
  Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

S.8 Simple RSL Specifications

"SLIDE 1630"

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
  ...
```

"SLIDE 1631"

Part IX

Solutions to Exercises

T

Solutions

T.1 Chapter 1: Introduction

Solution to Exercise 1. The Triptych Paradigm: The below forms an answer to Exercise # 1 Vol. I, Page 44.

Cf. Sect. 1.2 on page 4:

Before software can be designed one must understand its requirements.

Before requirements can be expressed one must understand the application domain.

Solution to Exercise 2. The Triptych Phases of Software Development: The below forms an answer to Exercise # 2 Vol. I, Page 44.

Sect. 1.3.1 on page 4:

Software development ideally progresses in three *phase*s: In the first phase, 'Domain Engineering', a model is built of the application domain. In the second phase, 'Requirements Engineering', a model is built of what the software should do (but not how it should that). In the third phase, 'Software Design', the code that is subject to executions on computers is designed.

Solution to Exercise 3. Phases, Stages and Steps of Software Development: The below forms an answer to Exercise # 3 Vol. I, Page 44.

By a *phase of development* we shall understand a set of development stages which together accomplish one of the three major development objectives: a(n analysed, validated, verified) domain model, a(n analysed, validated, verified) requirements model, or a (verified) software design.

By a *stage of development* we mean a major set of logically strongly related development steps which together solves a clearly defined development task.

By a *step of development* we mean iterations of development within a stage such that the purpose of the iteration is to improve the precision or

make the document resulting from the step reflect a more concrete description, prescription or specification.

Solution to Exercise 4. Development Documents: The below forms an answer to Exercise # 4 Vol. I, Page 44.

Cf. Sect. 1.5 Page 7:

- Informative Documents — Sect. 1.6 (Page 7)
- Modelling Documents — Sect. 1.7 (Page 25)
- Analysis Documents — Sect. 1.8 (Page 26)

Solution to Exercise 5. Enumeration of Informative Documents: The below forms an answer to Exercise # 5 Vol. I, Page 44.

1	Project Name and Date	Sect. 1.6.1
2	Project Place(s) ('where')	Sect. 1.6.2
3	Partners ('whom')	Sect. 1.6.2
	(a) Current Situation	Sect. 1.6.3
	(b) Needs and Ideas	Sect. 1.6.4
	(c) Concepts and Facilities	Sect. 1.6.5
	(d) Scope and Span	Sect. 1.6.6
	(e) Assumptions and Dependencies	Sect. 1.6.7
	(f) Implicit/Derivative Goals	Sect. 1.6.8
	(g) Synopsis	Sect. 1.6.9
	(a) Software Development Graph	Sect. 1.6.10
	(b) Resource Allocation	Sect. 1.6.11
	(c) Budget Estimate	Sect. 1.6.12
	(d) Standards Compliance	Sect. 1.6.13
4	Contracts and Design Briefs	Sect. 1.6.14
5	Logbook	Sect. 1.6.15

Solution to Exercise 6. Descriptions, Prescriptions, Specifications: The below forms an answer to Exercise # 6 Vol. I, Page 44.

Cf. Sect. 1.9:

(i) *descriptions* are of "what there is", that is, descriptions are, in this book, of domains, "as they are"; (ii) *prescriptions* are of "what we would like there to be", that is, prescriptions are, in this book, of requirements to software; and (iii) *specifications* are of "how it is going to be", that is, specifications are, in this book, of software.

Solution to Exercise 7. Software: The below forms an answer to Exercise # 7 Vol. I, Page 44.

Cf. Sect. 1.10: Software **is**, i.e., consists of the following documents:

1 The domain development documents include
   (a) the informative documents and
   (b) the documents which record
      i. stakeholder identification and relations,
      ii. domain acquisition,
      iii. domain analysis and concept formation,
      iv. rough sketches of the business (i.e., domain) processes,
      v. terminologies,
      vi. domain description,
      vii. domain verification (incl. model check and test),
      viii. domain validation and
      ix. domain theory formation
      documents.
2 The requirements development documents include
   (a) the informative documents and
   (b) the documents which record
      i. stakeholder identification and relations,
      ii. requirements acquisition,
      iii. requirements analysis and concept formation,
      iv. rough sketches of the re-engineered business (i.e., new, revised domain) processes,
      v. terminologies,
      vi. requirements description,
      vii. requirements verification (incl. model check and test),
      viii. requirements validation and
      ix. requirements theory formation
      documents.
   (c) The software design development documents include
      i. the informative documents,
      ii. the documents which record
         A. architectural designs ("how derived from requirements") and verifications (incl. model checks and tests),
         B. component designs and verifications (incl. model checks and tests),
         C. module designs and verifications (incl. model checks and tests),
         D. code designs and verifications (incl. model checks and tests),
      iii. the actual executable code documents,
      iv. as well as
         A. demonstration (i.e., demo) manuals,
         B. training manuals,
         C. installation manuals,
         D. user manuals,
         E. maintenance manuals, and
         F. development and maintenance logbooks.

Solution to Exercise 8. Informal and Formal Software Development: The below forms an answer to Exercise # 8 Vol. I, Page 44.

Cf. Sect. 1.11.1.

By *informal development* we understand a software development which does not use formal techniques, see below; instead it may use UML and an executable programming language.

By *formal development* we mean a software development which uses one or more formal techniques, see below, and it may then use these in a spectrum from systematically via rigorously to formally.

By *formal development* we mean, in this book, a software development which uses both informal and formal techniques. By a *formal development technique* we mean a software development in which specifications are expressed in a formal language, that is, a language with a formal syntax so that all specifications can be judged well-formed or not; a formal semantics so that all well-formed specifications have a precise meaning; and a (relatively complete) proof system such that one may be able to reason over properties of specifications or steps of formally specified developments from a more abstract to a more concrete step. Additionally a formal technique may be a calculus which allows developers to calculate, to refine "next", formally specified development steps from a preceding, formally specified step.

By a *systematic use of a formal technique* we mean a software development which which formally specifies whenever something is specified, but which does not (at least only at most in a minor of cases) reason formally over steps of development.

By a *rigorous use of formal techniques* we mean a software development which which formally specifies whenever something is specified, and which formally express (some, if not all) properties that ought be expressed, but which does not (at least only at most in a minor number of cases) reason formally over steps of development, that is, verify these to hold, either by theorem proving, or by model checking, or by formally based tests.

By *formal use of a formal techniques* we mean, in this book, a software development which which formally specifies whenever something is specified, which formally expresses (most, if not all) properties that ought be expressed, and which formally verifies these to hold, either by theorem proving, or by model checking, or by formally based tests.

Solution to Exercise 9. Entities and States, Functions and Actions, Events and Behaviours: The below forms an answer to Exercise # 9 Vol. I, Page 45.

Cf. Sect. 1.12 on page 34:

By an *entity* we mean something we can point to, i.e., something manifest, or a concept abstracted from, such a phenomenon or concept thereof.

By a domain *state* we shall understand a collection of domain entities chosen by the domain engineer.

By a *function* we shall understand something which when *applied* to what we shall call *arguments* (i.e., entities) *yield* some entities called the *result* of the function (application).

By an *action* we shall understand the same thing as applying a state-changing function to its arguments (including the state).

By an *event* we shall understand something that can be characterised by a predicate, p and a pair of ("before") and ("after") of pairs of states and times: $p((t_b, \sigma_b), (t_a, \sigma_a))$. Usually the time interval $t_a - t_b$ is of the order $t_a \simeq \text{next}(t_b)$.

By a *simple behaviour* we understand a sequence, q, of zero, one or more actions and/or events $q_1, q_2, \ldots, q_i, q_{i+1}, \ldots, q_n$ such that the state resulting from one such action, q_i, or in which some event, q_i, occurs, becomes the state in which the next action or event, q_{i+1}, if it is an action, is effected, or, if it is an event, is the event state. Etcetera for not so simple behaviours !

Solution to Exercise 10. Mereology, Atomic and Composite Entities: The below forms an answer to Exercise # 10 Vol. I, Page 45.

Cf. Sect. 1.12 on page 34:

By *mereology* we understand a theory of part-hood relations. That is, of the relations of part to whole and the relations of part to part within a whole

By an *atomic entity* we intuitively understand an entity which "cannot be taken apart" (into other, the sub-entities) and which possess one or more attributes.

By a *composite entity* we intuitively understand an entity (i) which "can be taken apart" into sub-entities, (ii) where the composition of these is described by its *mereology*, and (iii) which further possess one or more attributes

T.2 Chapter 2: Domain Engineering

Solution to Exercise 11. Route Type: The below forms an answer to Exercise # 11 Vol. I, Page 104.

```
1
2
3
4
5

type
  R =
```

Solution to Exercise 12. Route Generation: The below forms an answer to Exercise # 12 Vol. I, Page 104.

Solution to Exercise 13. Route Lengths: The below forms an answer to Exercise # 13 Vol. I, Page 105.

Solution to Exercise 14. DE4: The below forms an answer to Exercise # 14 Vol. I, Page 105.

Solution to Exercise 15. DE4: The below forms an answer to Exercise # 15 Vol. I, Page 105.

Solution to Exercise 16. DE4: The below forms an answer to Exercise # 16 Vol. I, Page 105.

Solution to Exercise 17. DE4: The below forms an answer to Exercise # 17 Vol. I, Page 105.

Solution to Exercise 18. DE4: The below forms an answer to Exercise # 18 Vol. I, Page 105.

T.3 Chapter 3: Requirements Engineering

Solution to Exercise 19. kap3.xs.1: The below forms an answer to Exercise # 19 Vol. I, Page 166.

Solution to Exercise 20. kap3.xs.2: The below forms an answer to Exercise # 20 Vol. I, Page 166.

Solution to Exercise 21. kap3.xs.3: The below forms an answer to Exercise # 21 Vol. I, Page 166.

Solution to Exercise 22. kap3.xs.4: The below forms an answer to Exercise # 22 Vol. I, Page 166.

Solution to Exercise 23. kap3.xs.5: The below forms an answer to Exercise # 23 Vol. I, Page 166.

Solution to Exercise 24. kap3.xs.6: The below forms an answer to Exercise # 24 Vol. I, Page 166.

Solution to Exercise 25. kap3.xs.7: The below forms an answer to Exercise # 25 Vol. I, Page 166.

Solution to Exercise 26. kap3.xs.8: The below forms an answer to Exercise # 26 Vol. I, Page 166.

T.4 Chapter 4: Software Design

Solution to Exercise 27. kap4.xs.1: The below forms an answer to Exercise # 27 Vol. I, Page 221.

Solution to Exercise 28. kap4.xs.2: The below forms an answer to Exercise # 28 Vol. I, Page 221.

Solution to Exercise 29. kap4.xs.3: The below forms an answer to Exercise # 29 Vol. I, Page 221.

Solution to Exercise 30. kap4.xs.4: The below forms an answer to Exercise # 30 Vol. I, Page 221.

Solution to Exercise 31. kap4.xs.5: The below forms an answer to Exercise # 31 Vol. I, Page 222.

Solution to Exercise 32. kap4.xs.6: The below forms an answer to Exercise # 32 Vol. I, Page 222.

Solution to Exercise 33. kap4.xs.7: The below forms an answer to Exercise # 33 Vol. I, Page 222.

Solution to Exercise 34. kap4.xs.8: The below forms an answer to Exercise # 34 Vol. I, Page 222.

T.5 Appendix D: Prelude Domain Actions

Solution to Exercise 35. Domain Prelude 1: The below formalisation forms an answer to Exercise # 35 Vol. II, Page 339.

Solution to Exercise 36. Domain Prelude 2: The below formalisation forms an answer to Exercise # 36 Vol. II, Page 340.

T.6 Appendix E: Intrinsics

Solution to Exercise 37. Link and Hub Attributes: The below formalisation forms an answer to Exercise # 37 Vol. II, Page 364.

type
37.1
37.2
37.3
37.4
37.5
37.6

Solution to Exercise 38. Link Units: The below formalisation forms an answer to Exercise # 38 Vol. II, Page 364.

type
38.1 U, UI, Len, Loc
value
38.1 obs_UI: U → UI, obs_Len: U → Len, obs_Loc: U → Loc
38.2 obs_Us: L → U-**set**
axiom
38.1 ∀ l:L • ∀ u,u′:U • {u,u′}⊆obs_Us(l) ∧ u≠u′ ⇒ obs_UI(u)≠obs_UI(u′)
38.3 ∀ (hs,ls):N • ∀ l,l′:L • {l,l′}⊆ls ∧ l≠l′ ⇒ obs_Us(l) ∩ obs_Us(l′) = {}
value
38.4 obs_LI: U → LI
axiom
38.4 ∀ l:L • ∀ u,u′:U • {u,u′}⊆obs_Us(l) ⇒ obs_LI(u)=obs_LI(u′)
value
38.5 obs_UQ: L × HI $\xrightarrow{\sim}$ U*
38.5 pre_obs_UQ(l,hi) ≡ hi ∈ obs_HIs(l)
axiom
38.5 ∀ l:L • **let** {hi′,hi″}=obs_HIs(l) **in**

elems obs_UQ(l,hi′)=obs_UQ(l,hi″)=obs_Us(l) ∧
obs_UQ(l,hi′)=reverse(obs_UQ(l,hi′)) **end**
value
38.6 reverse: U* → U*
38.6 reverse(ul) ≡ **if** ul=⟨⟩ **then** ⟨⟩ **else tl** ul ⌢ ⟨**hd** ul⟩ **end**

T.7 Appendix F: Support Technologies

Solution to Exercise 39. Semaphore Technology: The below formalisation forms an answer to Exercise # 39 Vol. II, Page 384.

Solution to Exercise 40. Optical Gate Technology: The below formalisation forms an answer to Exercise # 40 Vol. II, Page 384.

Solution to Exercise 41. : The below formalisation forms an answer to Exercise # 41 Vol. II, Page 384.

T.8 Appendix G: Management and Organisation

Solution to Exercise 42. 41: The below formalisation forms an answer to Exercise # 42 Vol. II, Page 405.

Solution to Exercise 43. 42: The below formalisation forms an answer to Exercise # 43 Vol. II, Page 405.

Solution to Exercise 44. 43: The below formalisation forms an answer to Exercise # 44 Vol. II, Page 405.

T.9 Appendix H: Rules and Regulations

Solution to Exercise 45. 51: The below formalisation forms an answer to Exercise # 45 Vol. II, Page 411.

Solution to Exercise 46. 52: The below formalisation forms an answer to Exercise # 46 Vol. II, Page 412.

T.10 Appendix I: Scripts

Solution to Exercise 47. 61: The below formalisation forms an answer to Exercise # 47 Vol. II, Page 450.

Solution to Exercise 48. 62: The below formalisation forms an answer to Exercise # 48 Vol. II, Page 450.

T.11 Appendix J: Human Behaviour

Solution to Exercise 49. 71: The below formalisation forms an answer to Exercise # 49 Vol. II, Page 454.

Solution to Exercise 50. 72: The below formalisation forms an answer to Exercise # 50 Vol. II, Page 454.

T.12 Appendix K: Postlude Domain Actions

Solution to Exercise 51. 001: The below formalisation forms an answer to Exercise # 51 Vol. II, Page 457.

Solution to Exercise 52. 002: The below formalisation forms an answer to Exercise # 52 Vol. II, Page 457.

T.13 Appendix L: Prelude Requirements Actions

Solution to Exercise 53. Reqs. Prelude 1: The below formalisation forms an answer to Exercise # 53 Vol. II, Page 469.

Solution to Exercise 54. Reqs. Prelude 2: The below formalisation forms an answer to Exercise # 54 Vol. II, Page 469.

T.14 Appendix M: Domain Requirements

Solution to Exercise 55. 81: The below formalisation forms an answer to Exercise # 55 Vol. II, Page 481.

Solution to Exercise 56. 82: The below formalisation forms an answer to Exercise # 56 Vol. II, Page 481.

T.15 Appendix N: Interface Requirements

Solution to Exercise 57. 91: The below formalisation forms an answer to Exercise # 57 Vol. II, Page 485.

Solution to Exercise 58. 92: The below formalisation forms an answer to Exercise # 58 Vol. II, Page 485.

T.16 Appendix O: Machine Requirements

Solution to Exercise 59. 101: The below formalisation forms an answer to Exercise # 59 Vol. II, Page 491.

Solution to Exercise 60. 102: The below formalisation forms an answer to Exercise # 60 Vol. II, Page 491.

T.17 Appendix P: Postlude Requirements Actions

Solution to Exercise 61. Reqs. Postludium 1: The below formalisation forms an answer to Exercise # 61 Vol. II, Page 494.

Solution to Exercise 62. Reqs. Postludium 2: The below formalisation forms an answer to Exercise # 62 Vol. II, Page 494.

T.18 Appendix Q: Software Design

Solution to Exercise 63. 111: The below formalisation forms an answer to Exercise # 63 Vol. II, Page 499.

Solution to Exercise 64. 112: The below formalisation forms an answer to Exercise # 64 Vol. II, Page 499.