Dines Bjørner

DOMAIN ENGINEERING

December 17, 2008. Compiled January 24, 2009, 17:21

To be submitted, late 2009, for evaluation, to:

Springer
Berlin Heidelberg New York
Hongkong London
Milan Paris Tokyo

History Logbook

- **17-Dec-2008:** Creation of all front and main matter files
- **18-Dec-2008:** Creation of all example and ”back” matter files
-
-
-
-
-

Dedication

BEING CONTEMPLATED

Preface

A Different Textbook !

This textbook shall teach you a modern, mathematics-based approach to one aspect of software development: domain engineering.

It does so in a novel way: Chaps. 1–21 of this book is a guide to the study of Appendices A–N of this book. Appendices A–N contains a fairly large 'support' example of a software development. It is carried out according to the principles and techniques outlined in Chaps. 1–21.

For lecturers there are electronic (postscript and pdf) slides covering all of Chaps. 1–21 and Appendices A–N. One way of lecturing based on this book is to display lecture slides (i.e., Chaps. 1–21) on one screen and lecture support slides (i.e., Appendices A–N) 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 Chaps. 1–21 and Appendices A–N have cross-references to corresponding slide versions !

Background

I wrote [25, 26, 27] as "The Mother of all Books on Software Engineering" !

Since the 2006 publication of [25, 26, 27] a few clarifications of some domain engineering principles and techniques have come about — and been published [31, 28, 32, 29, 34, 30].

The book [25, 26, 27], 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 [25, 26, 27] is the triptych of Vol. 3 [26].

The present book focuses on that domain engineering aspects of 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.

So, in two small volumes, one in paper format, the other probably as an enclosed CD ROM, you get the very essence of domain 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 O, summarises the syntactic aspects of the language.

Methodology: A 141 Page Guide to Domain Engineering

The methodology "volume" consists of

- Chaps. 1–21 (Pages 3–141) and
- Appendices T–V (Pages 339–431).

The two "volumes" are to be studied in companion: You put both volumes in front of you, perhaps Chaps. 1–21 in paper form, as a booklet, and Appendices A–N you may then display on your PC screen. Chaps. 1–21 makes numerous references to "this or that" appendix chapter and section of Appendices A–N. So you read Chaps. 1–21, get references to, and thus checks with "such and such" an appendix chapter and section of Appendices A–N. In lecture form slides will be available for the entire book. The lecturer will have both volumes displayable on two "parallel" lecture room overhead screens and can alternate between lecture parts from Chaps. 1–21 and example (support) parts from Appendices A–N.

There are three "administrative" appendices:

- Appendix T lists some 197 bibliography entries.
- Appendix U contains a rather complete (and hence large) Glossary (Pages 351–404). You may wish to study it all by itself ! It explains some 523 terms.
- Appendix V contains extensive Indexes:
  - ⋆ Sect. V.1 lists methodology concepts (Chaps. 1–21),
  - ⋆ Sect. V.2 lists methodology definitions,
  - ⋆ Sect. V.3 lists methodology principle enunciations,
  - ⋆ Sect. V.4 lists methodology technique enunciations,
  - ⋆ Sect. V.5 lists methodology tool enunciations,
  - ⋆ Sect. V.6 lists defining appearance os RSL symbols,
  - ⋆ Sect. V.7 lists examples and
  - ⋆ Sect. V.8 lists domain concepts (Appendices A–N).

Example: A Supporting Software Development

- Appendices B–M: A Domain Model Development (151–285);
- Appendices O–R: RSL, Petri Net, MSC and DC Primers (291–319);
- Appendix S: Solutions to Exercises of Chaps. 1–20 and Appendices A–M.

Volume I will exclusively consist of informal English text. That text explains the Triptych approach to domain engineering. Volume II provides all supporting examples on pages 151 to 120. Hence Chaps. 1–21 will make numerous references to sections and pages of Appendices A–N.

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 [82], imperative programming (as in a suitable subset of C, Java or C#), logic programming [124, 97, 5, 6] and parallel programming using, for example CSP [58, 96] 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 [190] 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 Appendices A–N 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 Appendices A–N, namely that of an oil & natural gas industry, suggests that students work out term reports much in the style of the model development of Appendices A–N, but for different domains. Any domain could be chosen, but we offer guidance, also in Appendices B–M of Appendices A–N, to such domains as: the transport, financial service industry and the 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 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. Exercise sections of most chapters of Chaps. 1–21 outline such group projects.

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

A Possible Lecture Plan

1: Introduction: The Triptych Approach to Software Engineering

- A Specification and Abstraction Ontology
  - 2: Entities
  - 3: Operations
  - 4: Events and Behaviours
- RSL: The Main Specification Language
  - 5: Types and Values
  - 6: Expressions, Statements and Processes

7: An Overview of Domain Engineering

- Preparation
  - 8: Information Documents
  - 9: Stakeholders
  - 10: Domain Acquisition
  - 11: Business Processes
    - Domain Analysis and Concept Formation
      - 12: Mereology
      - 13: Static and Dynamic Attributes
  - 14: Terminology
- Domain Modelling
  - 15: Intrinsics

Acknowledgements

Since my retirement from 31 years at the Technical University of Denmark I have enjoyed the support of Kokichi Futatsugi (JAIST[1] Japan), Dominique Méry (INRIA, Nancy, France), José Nuno de Oliveira (Univ. of Minho, Braga, Portugal), Bernhard Aichernig, Hermann Maurer and Franz Wotawa (T.U. Graz, Austria), Wolfgang Paul (Saarbrücken, Germany), *... a few more to come ...*, and collaboration with Asger Eir (Denmark), *... a few more to come ....* A number of PhD students at JAIST, INRIA, Univ. of Minho, T.U. Graz and Univ. of Saarland have also helped me clarifying methodological issues.

[1] Japan Adv. Inst. of Sci. and Techn., Ishikawa Province, near Kanazawa

Contents

Part I

OPENING

1 The Triptych Paradigms

slide 3

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. 1.2.1. 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 Sects. 1.2.2–1.2.3, and are covered in detail in Chaps. 3–18. The software engineering of these phases, their stages and steps are focused on constructing 'documents' — and the nature of these is covered in Sects. 1.3.1–1.3.3. 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. 1.4 briefly elaborates on these terms. The term 'software' is given a proper definition — one that most readers should find surprising — in Sect. 1.5. Section 1.6 covers the ideas behind pursuing software development both using informal and formal techniques. And Sect. 1.7 — another major study section of chapter — finally introduces the notions of entities, functions, events and behaviours. slide 4

1.1 The Domain Paradigm

slide 5

1.1.1 What is a Domain? – 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 6

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 7

(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 Development Paradigm

slide 8

> *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'. 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'.

1.2.1 The Triptych Phases of Software Development

slide 9

The Three Phases

As a consequence of the "dogma" we view software development as ideally progressing 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.

Attempts at Definitions

slide 10

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 11

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. ∎

Comments on The Three Phases

slide 12

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 13

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 14

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

Domain engineering is covered as follows: Chapters 3–18 outlines all the stages and steps of domain engineering. It does not bring examples. Instead Appendices A–N provide for one large example, the 'Model Development'. Hence Appendices A–N provides in "excruciating" detail examples of all the relevant aspects of domain engineering. These are then being referred to in Chaps. 1–21.

Requirements engineering is covered as follows: Chapter 20 outlines all the stages and steps of requirements engineering. Like Chaps. 3–18 Chap. 20 does not bring examples. Instead Appendix M provides a brief example of all the relevant aspects of requirements engineering. These are then being referred to in Chap. 20.

Software design is not covered in this book.

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.

1.2.2 The Triptych Stages of Development slide 15

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. 3–20 shall define the specific stages relevant to those phases of development.

1.2.3 The Triptych Steps of Development slide 16

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. ▪

1.3 The Document Paradigm slide 17

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 18

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:

• Informative Documents	Sect. 1.3.1 (Page 7)
• Modelling Documents	Sect. 1.3.2 (Page 7)
• Analysis Documents	Sect. 1.3.3 (Page 8)

1.3.1 Informative Documents slide 19

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 20

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.

1.3.2 Modelling Documents slide 21

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.

Domain Modelling Documents slide 22

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

1. stakeholder identification and liaison records,	Chap. 3
2. domain acquisition sketches,	Chap. 4
3. domain business process rough sketches,	Chap. 5
4. domain terminologies,	Chap. 6
5. and domain models proper.	Chaps. 7–15

slide 23

Chapters 3–18 will cover the domain engineering phase with its

• (i) stakeholder identification,	Chap. 3

[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 47.

- (ii) domain acquisition, Chap. 4
- (iii) domain analysis and concept formation, Chap. 5
- (iv) business process rough sketching, Chap. 6
- (v) terminology, Chap. 7
- (vi) domain modelling, Chaps. 8–15
- (vii) domain model verification, Chap. 16
- (viii) domain model validation, Chap. 17
- and (ix) domain theory formation Chap. 18

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.

Requirements Modelling Documents

slide 24

Chapter 20 covers requirements engineering in general and Sects. 20.3–20.6 cover requirements modelling in particular.

Requirements prescriptions are documents. They are usually rather substantial. They usually include the following kinds of 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 25

Chapter 20 will covers the requirements engineering phase with its

- (i) stakeholder identification, Sect. 20.2.2
- (ii) requirements acquisition, Sect. 20.2.3
- (iii) requirements analysis and concept formation, Sect. 20.2.5
- (iv) business process re-engineering rough sketching, Sect. 20.2.4
- (v) terminology, Sect. 20.2.6
- (vi) requirements modelling, Sects. 20.2.7, 20.3–20.6
- (vii) requirements model verification, Sect. 20.2.8
- (viii) requirements model validation, Sect. 20.2.9
- (ix) requirements feasibility and satisfiability analysis, Sect. 20.2.10
- and (x) requirements theory formation. Sect. 20.2.11

stages. Documents emerge from each of these stages.

1.3.3 Analysis Documents

slide 26

Verification, Model Checks and Tests

Characterisation 10 (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). ■

Concept Formation slide 27

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

Characterisation 11 (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. ■

Domain Analysis Documents slide 28

Stages (iii, vii, viii, ix) listed in Sect. 1.3.2 are analytic. They result in the following kinds of documents:

1. domain analysis (and concept formation)	Chap. 7
2. domain model verification,	Chap. 16
3. domain model validation,	Chap. 17
4. and domain theory formation.	Chap. 18

Requirements Analysis Documents slide 29

Stages (iii, vii, viii, ix, x) listed in Sect. 1.3.2 are analytic. They result in the following kinds of documents:

1. requirements analysis (and concept formation),	Sect. 20.2.5
2. requirements model verification,	Sect. 20.2.8
3. requirements model validation,	Sect. 20.2.9
4. requirements feasibility and satisfiability,	Sect. 20.2.10
5. and requirements theory formation.	Sect. 20.2.11

1.4 The Description, Prescription, Specification Paradigm slide 30

1.4.1 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.

1.4.2 Reiteration of Differences slide 31

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 32

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 33

(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.

1.4.3 Rôle of Domain Descriptions slide 34

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 35

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 slide 36

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 (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, slide 37

The Natural Sciences

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 38

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. slide 39

Research Areas of the Human Domains

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[3] human domain research areas as there are reasonably clearly separable such areas within the human domain. slide 40

Rôle of Domain Descriptions — Summarised

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.

[3] and we think: exciting research areas

1.4.4 Rôle of Requirements Prescriptions slide 41

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

The Machine

Characterisation 12 (Machine) By 'the machine' we shall mean a combination of hardware and software. ■

Machine Properties

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

1.4.5 Rough Sketches slide 42

Characterisation 13 (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.

1.4.6 Narratives slide 43

Characterisation 14 (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 44

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

1.4.7 Annotations slide 45

Characterisation 15 (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 46

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

1.5 The Software Paradigm slide 47

1.5.1 What is Software ?

Characterisation 16 (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. ■

1.5.2 Software is Documents ! slide 48

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 slide 49

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 slide 50

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.

Software System Documents slide 51

Characterisation 17 (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 52

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.

1.6 Informal and Formal Software Development slide 53

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.

1.6.1 Characterisations

Informal Development

Characterisation 18 (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 slide 54

Characterisation 19 (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. ■

A Spectrum of Developments slide 55

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

Formal Software Development

Characterisation 20 (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 [106], ASM (Abstract State Machines) [161], B and event-B [2, 44], DC (Duration Calculus) [196], MSC and LSC (Message and Live Sequence Charts) [87, 102, 103, 104], Petri Nets [152, 108, 158, 157, 159], Statecharts [83, 84, 86, 88, 85], RAISE (Rigorous Approach to Industrial Software Engineering) [25, 26, 27, 72, 74, 73], TLA+ (Temporal Logic of Actions) [115, 116, 134], VDM (Vienna Development Method) [36, 37, 69] and Z [173, 174, 194, 93]. The EATCS[4] Monograph [35] arose from [161, 44, 61, 142, 73, 134, 93] 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 [25, 26, 27]. slide 56

Systematic (Formal) Development !

Characterisation 21 (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. ∎

slide 57

Rigorous (Formal) Development !

Characterisation 22 (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. ∎

slide 58

Formal (Formal) Development !

Characterisation 23 (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. ∎

[4] EATCS: European Association for Theoretical Computer Science

1.6.2 Recommendations slide 59

This book advocates that software development be pursued according to the triptych paradigm, and that the phases, stages and steps, be pursued in a combination of both informal and formal descriptions, prescriptions and specifications, in a systematic to rigorous fashion. slide 60

slide 61–62

1.7 The Entity, Operation, Event and Behaviour Paradigm slide 63

We (forward) refer to appendix example Sect. C on page 153. It follows up on this methodology concept.

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 domains, requirements and software designs. slide 64 In the domain we observe phenomena. 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). slide 65 Concepts are defined.

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.

1.7.1 Discrete and Continuous Entities slide 66

The concepts of discrete and continuous are closely interwoven and are mainly, or best, understood in a physical context. From Wikipedia[5] we bring: *Discreteness constituting a separate thing; consisting of* non-overlapping, but possibly adjacent, but *distinct parts. Discreteness are fundamentally discrete in the sense of not supporting or requiring the notion of continuity. Continuity is seen as consistency, over time, of the characteristics of persons, plot, objects, places and events as observed.*

An Analysis slide 67

For our purposes we shall limit our consideration of entity discreteness and continuity to the physically, more specifically tactile manifested forms: if a

[5] Wikipedia was, around 2009 *The* Internet-based *Free Encyclopedia*.

physical, simple entity is fixed, that is does not change physical, spatial form, then we shall consider it a discrete, simple entity; if, instead, a physical, simple entity is liquid or gaseous, that is can, say through the force of gravity, change physical, spatial form, then we shall consider it a continuous, simple entity.

Structures

slide 68

Let us try encircle these concepts. To do so we introduce a notion of entity structure.

Characterisation 24 (Entity Structure) By the structure of an entity we understand how that entity is "made up", whether a simple entity, operation, event or behaviour; whether atomic, composite or continuous; whether static:fixed number of possible subentities and/or possible attributes, or dynamic:variable number of possible subentities and/or possible attributes. ■

slide 69

Observations

Note that the values of attributes and the number of alike sub-entities of composite entities may change while the structure remains the same. Thus the structure concept implies that if two or more simple entities, or one simple entity over time, has the same, fixed, invariant structure but with possibly changing values of attributes or changing number of sub-entities, then they are discrete, simple entities. We (forward) refer to appendix example Sect. E.1.2 on page 172. It follows up on this methodology concept. slide 70

Finite Structures

A simple entity structure is finite if it is either atomic or, if composite, then consists of a finite number of finite subentities, or, if continuous, its measure of "size", i.e., its amount of substance, or, if time, is finite.

Characterisations

slide 71

Characterisation 25 (Discrete) Being discrete is a property associated with entities.

An entity is discrete if it is timewise fixed, i.e., does not change spatial extent with time but could change value of possible sub-entities or of attributes.
■

Characterisation 26 (Continuous) Being continuous is a property associated with entities. An entity is continuous if it is timewise variable, i.e., can change spatial extent with time, or if any subpart of it is also an entity of the same structure. ■

Examples

slide 72

Our examples will be taken from the physical world as observed by physicists[6]. They are[7]:

- length (**meter**, m),
- mass (**Kilogram**, kg),
- **Time** (or just **T**) (**Second**, sec),
- electric current (**Ampere**, A),
- thermodynamic temperature (**Kelvin**, K),
- luminous intensity (**Candela**, cd) and
- amount of (chemical) substance (**Mol**))[8].

slide 73

Analysis of Time

There is a slight problem with the *time* example. There is *absolute time* and there is *relative time*.By *absolute time* we understand *time* since some "point in time". By *relative time* we understand a *time interval* between two *times*, i.e., two 'points in time'. slide 74 The difference between these concepts can, perhaps, best be understood in terms of the operations that one can perform on them: One can compare two *absolute times* in order to find out which *absolute time* is the later (earlier); and one can compare two *relative times* in order to find out which *time interval* is larger of the two. One cannot add two *absolute times*, but one can add *time intervals*. One can subtract two *absolute times*, one, a smaller, from the other, the larger, to obtain the elapsed *time interval*. And so forth. slide 75 In practice absolute times are abstracted as time and relative time are abstracted as time intervals; and both are abstracted in terms of continuous quantities (reals, **Real**). slide 76 In practice absolute time is concretised in terms of date (year (AD), month (Jan., Feb., ..., Dec.), day of month (1, 2, ..., 28, 29, 30 or 31)) and hour (0, 1, ..., 23), minute (0, 1, 2, ..., 59) and second (0, 1, 2, ..., 59)); and relative time is concretised in terms of number of elapsed years, months, days, hours, minutes and seconds; and these are expressed in terms of discretised quantities (say natural numbers).

We refer to Exercises 1.10.20 on page 36 and 1.10.21 on page 36.

[6] http://en.wikipedia.org/wiki/SI

[7] For each of the seven SI units we state its general (class) name and, in parentheses, the sort (i.e., type) name and, in *italic*, commonly used (textbook) abbreviations.

[8] The SI unit for amount of substance is the mole (type name: **mol**), which is defined as the amount of substance that has an equal number of elementary entities as there are atoms in 12g of carbon-12. That number is equivalent to (but not defined as) the Avogadro constant, NA, which has a value of 6.0221417910^{23} mol^{1}.

1.7.2 Entity Classification

slide 77

We shall now present a classification of simple entities. This classification shall serve and serves our purposes. The classification is not claimed to constitute a scientific theory or fact. But it is claimed to reflect our engineering approach to modelling.

The classification of simple entities partition their universe into three parts.

- continuous simple entities,
- atomic simple entities and
- composite simple entities.

slide 78

The classification allows composite simple entities to be composed from subentities that are either continuous or atomic or themselves composite. So it is not a matter of a simple entity which is not continuous instead being discrete. A discrete simple entity is an entity which is not continuous, and, if composite, then all of its subentities are discrete simple entities. Since all simple entities are of finite structure the above recursive characteristation stops.

1.7.3 Simple Entities

slide 79

We (forward) refer to appendix example Sect. C.5 on page 164. It follows up on this methodology concept.

Characterisation 27 (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 continuous, atomic or composite. The decision as to which simple entities are considered continuous, atomic or composite is a decision sôlely taken by the describer.

Atomic Simple Entities

slide 80

Characterisation 28 (Atomic Simple Entity) By an *atomic simple entity* we intuitively understand a simple entity which “cannot be taken apart” (into other, the sub-entities) and which possess one or more attributes. ■

Attributes — Types and Values

slide 81

With any entity we can associate one or more attributes.

Characterisation 29 (Attribute) By an *attribute* we understanda pair of a **type** and a **value**. ■

slide 82

Example 1 (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 slide 83

Characterisation 30 (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 slide 84

Characterisation 31 (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 [127, 138, 176, 177, 183].

Composite Entities — Continued slide 85

Example 2 (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 86

To put the above example of a composite entity in context we give an example of both an informal narrative and a corresponding formal specification: slide 87

Example 3 (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 88

```
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 . ∎

States slide 89

Characterisation 32 (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 Simple Entities slide 90

The Basics

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 91

Values of the type are then expressed as:

```
value
    a:A
```

As our main support example unfolds in Appendices A–N we shall illustrate sorts with their observer, generator and classifier functions and concrete types over either basic types (**Bool**eans, **Int**egers, **Nat**ural numbers, **Real**s, etc., or over composite types (sets, Cartesians, records, lists, maps, functions). Appendix Sect. O.2 (Pages 292–296) gives a terse introduction to the type system of our main formal specification language RSL. slide 92

Some Caveats

This section has dealt with discrete and continuous, and atomic and composite simple entities. where composite simple entities possessed subentities, a mereology of these, and attributes. We can (and must) express these distinctions (i.e., properties) of domains clearly in narratives, but cannot do so in our chosen, or for that matter in any formal specification language ! slide 93

Thus we shall, without discrimination, use the RSL type system to express both the possible subentity types of composite types and the attributes of these composite types. Whether a type is continuous, discrete or composite will only transpire rather indirectly from the formulas: in the form, for example, of observer functions, but these will not discriminate between observing attributes or subentities. slide 94

One could, of course, extend the (or any) formal specification language with the following literals (allowing plural forms): **attribute**, **subentity**, **discrete**, **continuous**, **atomic** and **composite**. These could be used, for example, in the following type expressions:

```
type
    atomic: A, B, ...
    composite: C, D, ...
    subentities: A, D, ...
    attributes: B, E, ...
    continuous: D, F, ...
```

or:

```
type
    atomic A, ... and
        composite D, ... as
            subentities of C, ...
```

Etcetera. We shall consider all such use of the literals as *pragmas*, that is, as pragmatic comments. Their presence (or absence) has no semantic importance.

Discussion slide 95

Simple Entities, Operations, Events and Behaviours as Entities

We have focused in this section on a concept of simple entities. We have used the prefix 'simple' since we consider the totality of simple entities, operations, events and behaviours to all be entities. As entities they are potentially arguments of operations, events and behaviours. We shall not here pursue this possibility further. slide 96

A Possible Critique of Our 'Simple Entity' Ontology

We advice the reader that the concepts of discrete and continuous, and attribute and subentities, and atomic and composite are "non-standard", i.e., not commonly accepted or in widespread usage. The reader might have guessed that from the "Some Caveats" paragraphs. Nevertheless we bring them here in textbook form since we think that they are indeed useful. slide 97

We also remind the reader that the concepts of, for example, physical types: **meter**, **kilogram**, **second**, **Ampere**, **Kelvin**, **candela** and **mol** and their derivatives as part of a specification language is not new, it was the subject of a number of (different) chapter exercises in Vol.2 of my book: and of a MSc. Thesis project in the late 1990s. Add to this "applied science" typology one slide 98 of "business" typology **Danish Kroner**, **Euro**, **revenue**, **expense**, **budget**, **debit**, **credit**, **spent**, **committed**, **asset**, **liability**, **account**, **balance sheet**, **general ledger**, **balance**, **sales**, **accounts receivable**, **inventory,** etc., etc., and you have exciting new projects in the design and implementation of domain specific programming languages that allow the separate definition of application specific type systems, of corresponding scale and conversion systems (millimeter, centimeter, meter, kilometer, inch, foot, yard, mile), etc., and of axioms that govern laws of physics, or laws of accounting, etc. slide 99
slide 100–101

1.7.4 Operations slide 102

We can, from a pragmatic viewpoint, distinguish between several classes of domain operations: (i) There are the operations that a stakeholder in the domain apply to one or more simple entities of the domain state in order to change that state (we shall refer to these operations as state-changing operations); (ii) there are the operations that a stakeholder in the domain apply to a simple entity of the domain state and in order to observe a property of that entity or "extract" a value of an attribute of that entity (we shall refer to these operations as state-observing or extracting operations); and (iii) there are the operations that a stakeholder in the domain apply to results of having applied state-observing or extracting operations in order to ascertain ("calculate") values of domain concepts.

We (forward) refer to appendix example Sect. C.6 on page 164. It follows up on this methodology concept.

Characterisations slide 103

Characterisation 33 (Operation) By an *operation* we shall understand something (i.e., a function) which when *applied* to what we shall call *arguments* (i.e., entities) *yield* some entities called the *result* of the function (application). ■

slide 104

The observer functions of the formal example above are not the kind of functions we are (later) seeking to identify in domains. 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.

Describing Operations slide 105

One can describe functions in a variety of ways. We shall briefly mention four: "direct" definitions, **pre/post** condition definitions, "mixtures" of the former two, and axiomatically. Each of these can be formulated either informally or formally. But first we introduce the concepts of operation (or function) signatures and of actions and their signatures.

Operation Signatures slide 106

Characterisation 34 (Operation Signature) By a *operation signature* we mean the *name* and *type* of a operation.

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

The last line above expresses a schematic operation signature. Narratively, it expresses that the function f takes ordered arguments of the types $A, B, ..., C$ and yields results of the (Cartesian) type $X \times Y \times ... \times Z$. ■

Actions slide 107

Characterisation 35 (Action) By an *action* we shall understand the same thing as applying a state-changing operation (function) to its arguments (including the state). ■

Action Signatures

One can speak of three kinds of actions, and hence of action signatures. Let Σ denote the type of states. slide 108

type
  A, B, Σ
value
  $\mathcal{V}al$: A $\rightarrow \Sigma \rightarrow$ B
  $\mathcal{I}nt$: A $\rightarrow \Sigma \rightarrow \Sigma$
  $\mathcal{E}lab$: A $\rightarrow \Sigma \rightarrow \Sigma \times$ B

$\mathcal{V}al$uation functions inspect the state, do not change it, and yield a value. $\mathcal{I}nt$erpretation functions inspect the state, change it, but do not yield a (further) value. $\mathcal{E}lab$oration functions inspect the state, change it, and yield a value. slide 109

Operation Definitions

Characterisation 36 (Operation Definiton) By an operation definition we mean an operation signature and something which describes the relationship between operation arguments (the a:A's, b:B's, ..., c:C's and the x:X's, y:Y's, ..., z:Z's). ∎

slide 110

Example 4 (Well Formed Routes) Presupposing material presented in Example 3 on page 21:

```
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" ∎

slide 111

Direct Operation Definitions

In a narrative direct operation definition the signature is described followed by an abstracted description of the operation, for example: "Operation f applies to arguments, a, of type A, and yields results, b or type B." "Operation f, when applied to argument a, i.e., $f(a)$, yields a result, b, which arises as follows ..." slide 112

Example 5 (The Factorial Function Definition: Direct) Informally and direct formally:

1. The factorial function applies to natural numbers and yields natural numbers. The factorial function is otherwise defined as follows:

2. Applied to the natural number 0 it yields the natural number 1.
   Applied to the natural number n (larger than 0) it yields
3. Applied to the natural number n (larger than 0) it yields
   (a) the result of multiplying the number n
   (b) with the result of applying the factorial function to $n-1$.

value
1 fact: **Nat** → **Nat**
2–3 fact(n) ≡ **if** n=0 **then** 1 **else** n∗fact(n−1) **end**

. ■

slide 113

A formal, direct function definition, including the function signature thus looks like this 'schematic':

value
f: A → B
f(a) ≡ $\mathcal{C}(a)$

Here A and B are some type clauses; $\mathcal{C}$ is some clause: some (process) expression, some (process) statement usually with a free identifier, say x, and $\mathcal{C}(a)$ designates the evaluation of that clause with the argument a bound to all occurrences of the free identifier x. slide 114

Pre/Post Operation Definitions

In a narrative **pre/post** operation definition the signature is described followed by an abstracted description of **pre/post** conditions of the operation, for example: "Operation f applies to arguments, a, of type A, and yields results, b or type B." "Operation f, when applied to argument a, i.e., $f(a)$, yields a result, let us call it b; "A **pre** condition of a is that it satisfies the following predicate: ... , etc." "The relation between proper input arguments a and results, b is expressed by the **post** condition: ..., etc." slide 115

Example 6 (Factorial Function Definition: pre/post) Informally and formally: **Narrative:** The factorial function applies to natural numbers and yields natural numbers. The factorial function is otherwise characterised as follows: Let the factorial of n be called n'; n must be larger than or equal to 0; $n=0$ implies that the factorial of n is 1; $n{>}0$ implies that the factorial of n is the result of multiplying the number n with the result of applying the factorial function to $n-1$. **Formalisation:**

value
fact: N → N
fact(n) **as** n′
pre: n≥0
post: n=0 ⇒ n′=1 ∧ n>0 ⇒ n′=n∗fact(n−1)

. ■

slide 116

A **pre/post** operation definition, including the function signature looks like this 'schematic':

```
value
  f: A → B
  f(a) as b
    pre: $\mathcal{P}(a)$
    post: $\mathcal{Q}(a,b)$
```

Here A and B are some type clauses; $\mathcal{P}$ is some predicate expression with free identifier, say x; $\mathcal{Q}$ is some predicate expression with free identifiers, say y, z; $\mathcal{P}(a)$ designates the evaluation of the predicate with the argument a bound to all occurrences of the free identifier x; and $\mathcal{Q}(a,b)$ designates the evaluation of the predicate with the arguments a, b bound to all occurrences of respecxtive free identifiers y and z. slide 117

"Mixed" Operation Definitions

In a narrative, mixed operation definition the signature is described — usually involving arguments (and possibly also results) types that denote "larger" sets of values than actually accepted (respectively yielded). followed by an abstracted description of the operation, for example: "Operation f applies to arguments, a, of type A, and yields results, b or type B." "Operation f, when applied to argument a, i.e., $f(a)$, yields a result, b, which arises as follows ..."; followed, finally, by a **pre** condition (on the operation input arguments). slide 118

Example 7 (Factorial Function Definition: "Mixed") Informally and formally: **Narrative:** The factorial function applies to integers and yields natural numbers. The factorial function is otherwise characterised as follows: 5 on page 25

Formalisation:

```
value
  fact: Int → Nat
  fact(n) ≡ if n=0 then 1 else n∗fact(n−1) end
    pre: n≥0
```

. ■

slide 119

A formal, "mixed" operation definition, including the function signature thus looks like this 'schematic':

```
value
  f: A → B
  f(a) ≡ $\mathcal{C}(a)$
    pre: $\mathcal{P}(a)$
```

Here A and B are some type clauses; $\mathcal{C}$ is some clause: some (process) expression, some (process) statement usually with a free identifier, say x; $\mathcal{C}(a)$ designates the evaluation of that clause with the argument a bound to all occurrences of the free identifier x; $\mathcal{P}$ is some predicate expression with free identifier, say x; and $\mathcal{P}(a)$ designates the evaluation of the predicate with the argument a bound to all occurrences of the free identifier x. slide 120

Axiomatic Operation Definitions

Example 8 (Factorial Function Definition: axiomatic) Informally and formally: **Narrative:** The factorial function, together with natural numbers satisfy the following two axioms: factorial of 0 is 1, and factorial of n, for n larger than 0, is n multiplied by the factorial of $n-1$. **Formalisation:**

```
value
   fact: N → N
axiom
   ∀ n:Nat •
      n=0 ⇒ fact(0) = 1 ∧
      n>0 ⇒ fact(n) = n∗fact(n−1)
```

. ■

slide 121

Formal axiomatic operation definitions thus looks like this ‘schematic’:

```
type
   A, B
value
   f: A → B
axiom
   $\mathcal{P}_i(a,b)$,
   $\mathcal{P}_j(a,b)$,
   ...,
   $\mathcal{P}_k(a,b)$,
```

$\mathcal{P}_i, \mathcal{P}_j, \ldots, \mathcal{P}_k$ are some predicate expression with free identifiers, say x, y; and $\mathcal{P}(a,b)$ designates the evaluation of the predicate with the argument a, b bound to all occurrences of the free identifiers x, y.

Discussion slide 122

slide 123
slide 124
slide 125
slide 126–127

1.7.5 Events slide 128

We (forward) refer to appendix example Sect. C.7 on page 164. It follows up on this methodology concept.

Characterisation 37 (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 129

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 130

Example 9 (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!

1.7.6 Behaviours

slide 131

We (forward) refer to appendix example Sect. C.8 on page 164. It follows up on this methodology concept.

Simple Behaviours

Characterisation 38 (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 132

Example 10 (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

slide 133

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

slide 134

Concurrent Behaviours

Characterisation 39 (Concurrent Behaviour) By a concurrent behaviour we shall understand a set of behaviours (simple or otherwise). ■

slide 135

Example 11 (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. ■

slide 136

Communicating Behaviours

Characterisation 40 (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 137

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 138

Example 12 (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 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 slide 139

Communicating behaviours, the only really interesting behaviours, can be modelled in a great variety of ways: from set-oriented models in B [2, 45], RSL [72, 74, 25, 26, 27, 71, 33], VDM [36, 37, 69, 68], or Z [173, 174, 194, 93, 92], to models using for example CSP [95, 167, 169, 96],(as for example "embedded" in RSL [72],), or, to diagram models using, for example, Petri nets [108, 152, 158, 157, 159], message [102, 103, 104] or live sequence charts [55, 87, 110], or state-charts [83, 84, 86, 88, 85].

1.7.7 Discussion slide 140

The main aim of Sect. 1.7 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 [172], is not necessary, or rather, is replaced by how we model attributes of, for example, entities, and how we model facets (Chaps. 10–15).

1.7.8 Operations, Events and Behaviours as Entities slide 141

Review of Entities

In the example of Appendices A–N we identify the following as being entities:
(i) (ii) (iii) (iv) (v) (vi) (vii) (viii) (ix) (x) slide 142

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). slide 143

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. slide 144

It may surprise some that we designate positions as entities. They are certainly manifest: one can point to a position. slide 145

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.

Operations as Entities slide 146

TO BE WRITTEN

Events as Entities slide 147

TO BE WRITTEN

Behaviours as Entities slide 148

TO BE WRITTEN

1.8 Domain vs. Operational Research Models slide 149

1.8.1 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.

1.8.2 Reasons for Operational Research Analysis slide 150

Operational research (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.

1.8.3 Domain Models slide 151

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.

1.8.4 Domain and OR Models slide 152

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.

1.8.5 Domain versus Mathematical Modelling slide 153

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 !

1.9 Summary slide 154

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. 1.2.1); (iii) the 'stages' and 'steps' of software development (Sect. **??**); (iv) the three classes of development documents (Sect. 1.3); (v) the detailed nature of 16 kinds of 'informative documents' (Sect. 1.3.1); (vi) the concepts of 'modelling documents' (Sect. 1.3.2); (vii) the concepts of 'analysis slide 155 documents' (Sect. 1.3.3); (viii) the concepts of 'descriptions, prescriptions' and 'specifications' (Sect. 1.4); (ix) the concept of 'software' (Sect. 1.5); (x) the concepts (Sect. 1.6) 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. 1.7); and (xii) the concepts of 'operational research' versus those of 'domain models' (Sect. 1.8). slide 156

1.10 Exercises

1.10.1 What is a Domain?

Solution S.1.1 on page 323, suggests an answer to this exercise.

1.10.2 Are These Domains?

Explain why you think the below are, or are not examples of domains:

0.1. Programming
0.2. Compilers
0.3. Compiler Writing
0.4. Patient Hospitalisation

Solution S.1.2 on page 323, suggests an answer to this exercise.

1.10.3 The Triptych Paradigm

Solution S.1.3 on page 324, suggests an answer to this exercise.

1.10.4 The Three Phases of Software Development

Solution S.1.4 on page 324, suggests an answer to this exercise.

1.10.5 Domain Engineering

Solution S.1.5 on page 324, suggests an answer to this exercise.

1.10.6 Requirements Engineering

Solution S.1.6 on page 324, suggests an answer to this exercise.

1.10.7 Software Design

Solution S.1.7 on page 324, suggests an answer to this exercise.

1.10.8 What is a Model

Solution S.1.8 on page 324, suggests an answer to this exercise.

1.10.9 Phase of Development

Solution S.1.9 on page 324, suggests an answer to this exercise.

1.10.10 Stage of Development

Solution S.1.10 on page 324, suggests an answer to this exercise.

1.10.11 Step of Development

Solution S.1.11 on page 324, suggests an answer to this exercise.

1.10.12 Development Documents

Solution S.1.12 on page 324, suggests an answer to this exercise.

1.10.13 Descriptions, Prescriptions and Specifications

Solution S.1.13 on page 324, suggests an answer to this exercise.

1.10.14 Software

Solution S.1.14 on page 325, suggests an answer to this exercise.

1.10.15 Informal and Formal Software Development

Solution S.1.15 on page 325, suggests an answer to this exercise.

1.10.16 Specification Ontology

Which are the four kinds of phenomena and concepts around which our informal (i.e., narrative) and formal descriptions, prescriptions and specifications evolve ?

Solution S.1.16 on page 325, suggests an answer to this exercise.

1.10.17 Discreteness

How does this chapter characterise ‘being discrete’ ?

Solution S.1.17 on page 325, suggests an answer to this exercise.

1.10.18 Continuous

How does this chapter characterise ‘being continuous’ ?

Solution S.1.18 on page 325, suggests an answer to this exercise.

1.10.19 Discrete and Continuous Entities

Which of these examples refer to discrete entities and which refer to continuous entities:

0.1.
0.2.
0.3.
0.4.

Solution S.1.19 on page 325, suggests an answer to this exercise.

1.10.20 Operations on Time and Time Intervals

We refer to Sect. 1.7.1 (Pages 18–18). State the formal signature of all the operations that you can think of in connection with times and time intervals.

Solution S.1.20 on page 326, suggests an answer to this exercise.

1.10.21 Operations on Oil and Gas

We assume that oil (which is here seen as a liquid) and gas (which is here seen as gaseous), that is, are continuous entities.

Now define the oil and gas sorts (i.e., abstract types) and postulate operations that obs_erve amount of oil respectively amount of gas.[9] Now state the formal signatures of the operations that you can think of in connection with oil and gas.

Solution S.1.21 on page 326, suggests an answer to this exercise.

1.10.22 Simple Entities

Solution S.1.22 on page 327, suggests an answer to this exercise.

1.10.23 Operations

Solution S.1.23 on page 327, suggests an answer to this exercise.

1.10.24 Events

Solution S.1.24 on page 327, suggests an answer to this exercise.

1.10.25 Behaviours

Solution S.1.25 on page 327, suggests an answer to this exercise.

1.10.26 Atomic and Composite Entities

Solution S.1.26 on page 327, suggests an answer to this exercise.

1.10.27 Mereology

What is meant by mereology ?

Solution S.1.27 on page 327, suggests an answer to this exercise.

[9] For the concept of ‘amount of substance (oil or gas)’ please see Page 18.

1.10.28 Operations Research (OR)

How does this chapter define Operations Research (OR)?
Solution S.1.28 on page 327, suggests an answer to this exercise.

1.10.29 OR versus Domain Modelling

How does this chapter characterise differences between ‘OR Modelling’ and ‘Domain Modelling’?
Solution S.1.29 on page 327, suggests an answer to this exercise.

slide 157–158

Part II

DOMAIN ENGINEERING

2

Domain Engineering: An Overview slide 159

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 160

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 161

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 162

As also outlined in Sects. 1.2 and 1.8, 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 163

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 164

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 165

2.2.0 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.

2.2.1 [1] Domain Information slide 166

We are here referring to the construction of informative documents.

We have earlier, as mentioned above, (Page 7) introduced the general issues of informative documents.

Chapter 3 (Pages 47–64) covers the present topic in some detail.

Suffice it here to emphasize that each and every of the items listed on Page 47 must be kept up-to-date during the full development cycle. This means that this activity is of "continuing concern" all during development. slide 167

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.

2.2.2 [2] Domain Stakeholder Identification slide 168

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.

Chapter 4 (Pages 67–68) covers the present topic in some detail.

2.2.3 [3] Domain Acquisition slide 169

The software engineers need a domain description. Software engineers, today, are basically the only ones who have the tools[1], 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 170

Characterisation 41 (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 53 on page 71.

Chapter 5 (Pages 71–73) covers the present topic in some detail.

2.2.4 [4] Domain Analysis and Concept Formation slide 171

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.

Chapter 6 (Pages 75–76) covers the present topic in some detail.

2.2.5 [5] Domain Business Processes slide 172

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.

Chapter 7 (Pages 79–81) covers the present topic in some detail.

2.2.6 [6] Domain Terminology slide 173

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 slide 174 aspect of software development to establish, use and maintain a variety of terminologies. And first comes the domain terminology.

Chapter 8 (Pages 83–85) covers the present topic in some detail.

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

2.2.7 [7] Domain Modelling slide 175

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.

Chapters 10–15 (Pages 93–120) covers the present topic in some detail.

2.2.8 [8] Domain Verification slide 176

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.

Chapter 16 (Pages 123–123) covers the present topic in some detail.

2.2.9 [9] Domain Validation slide 177

Characterisation 42 (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. ■

Chapter 17 (Pages 125–125) covers the present topic in some detail.

2.2.10 [10] Domain Verification versus Domain Validation slide 178

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

2.2.11 [11] Domain Theory Formation slide 179

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.

Chapter 18 (Pages 127–127) covers the present topic in some detail.

2.3 A Summary Enumeration

slide 180

We can now summarise the relevant stages of domain engineering:

1.	Domain Information	Chap. 3 (Pages 47–64)
2.	Domain Stakeholder Identification,	Chap. 4 (Pages 67–68)
3.	Domain Acquisition,	Chap. 5 (Pages 71–73)
4.	Domain Analysis and Concept Formation,	Chap. 6 (Pages 75–76)
5.	Domain [i.e., Business] Processes,	Chap. 7 (Pages 79–81)
6.	Domain Terminology,	Chap. 8 (Pages 83–85)
7.	Domain Modelling,	Chaps. 10–15 (Pages 93–120)
(a)	Intrinsics	Chap. 10 (Pages 93–94)
(b)	Support Technologies	Chap. 11 (Pages 97–98)
(c)	Management & Organistation	Chap. 12 (Pages 101–107)
(d)	Rules & Regulations	Chap. 13 (Pages 109–112)
(e)	Scripts and Contracts	Chap. 14 (Pages 115–117)
(f)	Human Behaviour	Chap. 15 (Pages 119–120)
8.	Domain Verification,	Chap. 16 (Pages 123–123)
9.	Domain Validation and	Chap. 17 (Pages 125–125)
10.	Domain Theory Formation,	Chap. 18 (Pages 127–127)

slide 181

slide 182–183

3 Informative Documents

slide 184

Appendix A (Pages 145–148) complements the present chapter.

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 185

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.

3.1 An Enumeration of Informative Documents

slide 186

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 187

[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 47.

3.2 Project Names and Dates slide 188

We (forward) refer to appendix example Sect. A.1 on page 145. It follows up on this methodology concept.

The first information are those of

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

3.3 Project Partners and Places slide 189

We (forward) refer to appendix example Sect. A.2 on page 145. It follows up on this methodology concept.

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 190
  - ⋆ 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 191
- Project Places: where is the project carried out ? Full addresses: visiting and postal mailing addresses and electronic addresses. slide 192

3.4 Current Situation slide 193

We (forward) refer to appendix example Sect. A.3 on page 145. It follows up on this methodology concept.

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 194 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 195 For a software design project ....

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

3.5 Needs and Ideas slide 196

3.5.1 Needs

We (forward) refer to appendix example Sect. A.4.1 on page 145. It follows up on this methodology concept.

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 197

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 198

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.

3.5.2 Ideas slide 199

We (forward) refer to appendix example Sect. A.4.2 on page 145. It follows up on this methodology concept.

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 !

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. slide 200

By *domain ideas* we mean such concepts "upon" or "around" which one can build, one can model, a domain description.

By *requirements ideas* we mean such concepts "upon" or "around" which one can build, one can model, a requirements prescription.

By *software design ideas* we mean such concepts "upon" or "around" which one can build, one can model, a software design.

3.6 Facilities and Concepts slide 201

We (forward) refer to appendix example Sect. A.5 on page 146. It follows up on this methodology concept.

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 202

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 203

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.

3.7 Scope and Span

slide 204

Characterisation 43 (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. . ■

We (forward) refer to appendix example Sect. A.6.1 on page 146. It follows up on this methodology concept.

Characterisation 44 (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. .■

We (forward) refer to appendix example Sect. A.6.2 on page 146. It follows up on this methodology concept. slide 205

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 206

(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 207

(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.

3.8 Assumptions and Dependencies

slide 208

We (forward) refer to appendix example Sect. A.7 on page 146. It follows up on this methodology concept.

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 209

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 210

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.

3.9 Implicit & Derivative Goals slide 211

We (forward) refer to appendix example Sect. A.8 on page 146. It follows up on this methodology concept.

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 212

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 213

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 214

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

3.10 Synopsis slide 215

We (forward) refer to appendix example Sect. A.9 on page 146. It follows up on this methodology concept.

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 216

Characterisation 45 (Synopsis) By a *synopsis*[3] — 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 217 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). ▪

3.11 Software Development Graphs slide 218

We (forward) refer to appendix example Sect. A.10 on page 146. It follows up on this methodology concept.

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 219 capturing, either beforehand, during, or after the project how that project is to be done, is being done, or was done, respectively !

3.11.1 Graphs slide 220

Characterisation 46 (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 221

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*.

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. ▪

[3] Synopsis: Greek, comprehensive view, from *synopsis:* to be going to see together.

3.11.2 A Conceptual Software Development Graph slide 222

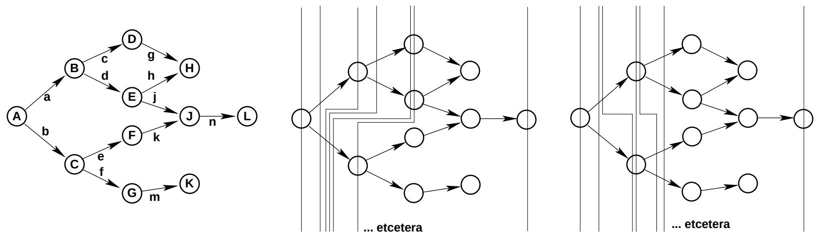

Fig. 3.1. A software development graph (left) and two (incomplete) project behaviour designators (center and right)

The center graph of Fig. 3.1 portrays the following incompletely listed project behaviour designator:

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

slide 223

The "abstracted" software development graph of Fig. 3.1 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.

3.11.3 Who Sets Up the Graphs ? slide 224

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.

3.11.4 How Do Software Development Graphs Come About ? slide 225

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 226

(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. 21.1 (Chap. 21, Sect. 21.1 on page 139) 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 227

If it is a requirements engineering projectthen one starts with the domain engineering process graph of Fig. 20.1 (Chap. 20, Sect. 20.8.1 on page 136) 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.

3.12 Resource Allocation slide 228

We (forward) refer to appendix example Sect. A.11 on page 147. It follows up on this methodology concept.

Characterisation 47 (Software Development Graph Attribute) An *attributed software development graph* is a software development graph whose nodes and edges have been assigned development attributes. ■

slide 229

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; any one of the 16 kinds of information document development steps enumerated on Page 47; or any one of the many stages or steps of the domain modelling and analysis otherwise "revealed" in Chaps. 1–21. slide 230

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 slide 231 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.

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

3.13 Budget (and Other) Estimates

slide 232

3.13.1 Budget

We (forward) refer to appendix example Sect. A.12.1 on page 147. It follows up on this methodology concept.

3.13.2 Other Estimates

slide 233

We (forward) refer to appendix example Sect. A.12.2 on page 147. It follows up on this methodology concept. 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.

3.14 Standards Compliance

slide 234

A distinction is made between development standards and documentation standards.

3.14.1 Development Standards

We (forward) refer to appendix example Sect. A.13.1 on page 147. It follows up on this methodology concept.

Usually development occurs in the context of following some development standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [99]) 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 [101]) and the International Telecommunication Union (ITU [105]), have established similar standards.

3.14.2 Documentation Standards slide 235

We (forward) refer to appendix example Sect. A.13.2 on page 147. It follows up on this methodology concept.

Usually documentation occurs in the context of following some documentation standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [99]) 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 [101]) and the International Telecommunications Union (ITU [105]), have also established similar standards.

3.14.3 Standards Versus Recommendations slide 236

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.

3.14.4 Specific Standards slide 237

We (forward) refer to appendix example Sect. A.13.4 on page 147. It follows up on this methodology concept.

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 238

- *International Organization for Standardization (ISO):* `http://www.iso.ch/`
  - ⋆ 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 239
- 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 240
- ⋆ 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 241
- ⋆ 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 242

- 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.

3.15 Contracts and Design Briefs slide 243

3.15.1 Contracts slide 244

We (forward) refer to appendix example Sect. A.14.1 on page 147. It follows up on this methodology concept.

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 48 (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 245

Items (iii–iv) constitute the main part of a *design brief.* (See below.) slide 246

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 247

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.

3.15.2 Contract Details slide 248

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. slide 249
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.

2. **General Issues of Licensing:** slide 250
   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.
3. **Copyright:** slide 251
   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.
4. **Domain, Requirements and Software Development Specifications:** slide 252
   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 253
5. **Performance Standards:** slide 254
   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 (very lightly) covered in Chap. 20.
6. **Documentation, Modification, Updates and Conversion:** slide 255
   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 256
   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 257
   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 258
   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 259
    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 260
    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 261
    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.
13. **Vendor Liability Limitations:** slide 262
    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.

14. **IT Security:** slide 263
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.
15. **Subcontracting and Multiple Vendor Relationships:** slide 264
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.
16. slide 265 **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.

3.15.3 Design Briefs slide 266

We (forward) refer to appendix example Sect. A.14.2 on page 147. It follows up on this methodology concept.

Characterisation 49 (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. ▪

3.16 Development Logbook slide 267

We (forward) refer to appendix example Sect. A.15 on page 148. It follows up on this methodology concept.

Characterisation 50 (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 268

Example 13 (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. . ▪

3.17 Discussion of Informative Documentation slide 269

3.17.1 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 270

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.

3.17.2 Methodological Consequences: Principle, Techniques and Tools

slide 271

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 272

Principle 2 (Information Documents) The informative documents must be authoritative, definitive and interesting to read. ■

slide 273

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 274

slide 275

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 [51]) and you have a workable system. ■

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

3.18 Exercises

3.18.1 1.a

Solution S.2.1 on page 328, suggests an answer to this exercise.

3.18.2 1.b

Solution S.2.2 on page 328, suggests an answer to this exercise.

3.18.3 1.c

Solution S.2.3 on page 328, suggests an answer to this exercise.

3.18.4 1.d

Solution S.2.4 on page 328, suggests an answer to this exercise. slide 276

4 Stakeholder Identification and Liaison

slide 279

Appendix B (Pages 151–151) complements the present chapter.

4.1 Characterisations

Characterisation 51 (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 280

Characterisation 52 (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.■

4.2 Why Be Concerned About Stakeholders ?

slide 281

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.

4.3 How to Establish List of Stakeholders ?

slide 282

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.

4.4 Form of Contact With Stakeholders slide 283

Chapters 5 and 17 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. slide 284

4.5 Principles, Techniques and Tools slide 285

4.6 Discussion slide 286

4.7 Exercises

4.7.1 2.a

Solution S.3.1 on page 328, suggests an answer to this exercise.

4.7.2 2.b

Solution S.3.2 on page 328, suggests an answer to this exercise.

4.7.3 2.c

Solution S.3.3 on page 328, suggests an answer to this exercise.

4.7.4 2.d

slide 287 Solution S.3.4 on page 328, suggests an answer to this exercise.

5 Domain Acquisition

slide 290

Appendix C (Pages 153–164) complements the present chapter.

5.1 Another Characterisation

Characterisation 53 (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 41 on page 43.

5.2 Sources of Domain Knowledge

slide 291

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[1], from infrastructure books, papers, etc.[2], from owners and staff of the client[3], from customers of

[1] 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.

[2] Similarly to footnote 1.

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

the client[4], possibly from domain regulators[5], from consultancy, equipment and service providers for and to the client[6] and possibly others.

5.3 Forms of Solicitation and Elicitation slide 292

5.3.1 Solicitation

How can the domain engineer solicit[7] 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.

5.3.2 Elicitation slide 293

How does the domain engineer elicit[8] 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.

5.3.3 Solicitation and Elicitation slide 294

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.

5.4 Aims and Objectives of Elicitation slide 295

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.

[4] 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.

[5] 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.

[6] We exclude the developers from this list.

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

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

5.5 Domain Description Units

slide 296

5.5.1 Characterisation

Characterisation 54 (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
. ■

5.5.2 Handling

slide 297

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[9]). slide 298

5.6 Principles, Techniques and Tools

slide 299

5.7 Discussion

slide 300

5.8 Exercises

5.8.1 3.a

Solution S.4.1 on page 328, suggests an answer to this exercise.

5.8.2 3.b

Solution S.4.2 on page 329, suggests an answer to this exercise.

5.8.3 3.c

Solution S.4.3 on page 329, suggests an answer to this exercise.

5.8.4 3.d

Solution S.4.4 on page 329, suggests an answer to this exercise. slide 301

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

slide 302–303

6 Business Processes

slide 304

Appendix D (Pages 167–167) complements the present chapter.

6.1 Characterisation

Characterisation 55 (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. ∎

6.2 Business Process Description

slide 305

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

6.3 Aims & Objectives of Business Process Description

slide 306

6.3.1 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.

6.3.2 Objectives slide 307

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.

6.4 Disposition slide 308

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. slide 309

6.5 Principles, Techniques and Tools slide 310

6.6 Discussion slide 311

6.7 Exercises

6.7.1 4.a

Solution S.5.1 on page 329, suggests an answer to this exercise.

6.7.2 4.b

Solution S.5.2 on page 329, suggests an answer to this exercise.

6.7.3 4.c

Solution S.5.3 on page 329, suggests an answer to this exercise.

6.7.4 4.d

Solution S.5.4 on page 329, suggests an answer to this exercise. slide 312

7 Domain Analysis and Concept Formation slide 315

Appendix E (Pages 169–174) complements the present chapter.

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.

7.1 Characterisations slide 316

First some preliminaries.

7.1.1 Consistency

Characterisation 56 (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. ▪

7.1.2 Contradiction slide 317

Characterisation 57 (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. ▪

7.1.3 Completeness slide 318

Characterisation 58 (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.

7.1.4 Conflict

slide 319

Characterisation 59 (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 320

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.

7.2 Aims and Objectives of Domain Analysis

slide 321

7.2.1 Aims of Domain Analysis

Characterisation 60 (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. ▪

7.2.2 Objectives of Domain Analysis

slide 322

Characterisation 61 (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. ▪

7.3 Concept Formation

slide 323

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 62 (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. ▪

7.3.1 Aims and Objectives of Domain Concept Formation slide 324

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.

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. slide 325

7.4 Principles, Techniques and Tools slide 326

7.5 Discussion slide 327

7.6 Exercises

7.6.1 5.a

Solution S.6.1 on page 329, suggests an answer to this exercise.

7.6.2 5.b

Solution S.6.2 on page 329, suggests an answer to this exercise.

7.6.3 5.c

Solution S.6.3 on page 330, suggests an answer to this exercise.

7.6.4 5.d

Solution S.6.4 on page 330, suggests an answer to this exercise. slide 328

8 Terminology

slide 331

Appendix F (Pages 177–194) complements the present chapter.

8.1 The 'Terminology' Dogma

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

8.2 Characterisations

slide 332

Characterisation 63 (Term) By a *term* is here meant [123]: 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 333

Characterisation 64 (Terminology) By *terminology* is meant [123]: the doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature. ▪

8.3 Term Definitions

slide 334

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 335

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.

8.4 Aims and Objectives of a Terminology slide 336

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[1], the developers and the domain description readers obtain as near, if not, the same understanding of the recorded terms.

8.5 How to Establish a Terminology slide 337

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 338

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. slide 339

8.6 Principles, Techniques and Tools slide 340

8.7 Discussion slide 341

8.8 Exercises

8.8.1 6.a

Solution S.7.1 on page 330, suggests an answer to this exercise.

8.8.2 6.b

Solution S.7.2 on page 330, suggests an answer to this exercise.

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

8.8.3 6.c

Solution S.7.3 on page 330, suggests an answer to this exercise.

8.8.4 6.d

Solution S.7.4 on page 330, suggests an answer to this exercise. slide 342

Part III

MODELLING STAGES

9

Domain Modelling: An Overview

slide 345

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.

9.2 Domain Facets

slide 346

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 347

Characterisation 65 (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. ■

9.3 Describing Facets

slide 348

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. slide 349

9.4 Principles, Techniques and Tools slide 350

9.5 Discussion slide 351

slide 352

10

Domain Modelling: Intrinsics

slide 355

Appendix G (Pages 197–224) complements the present chapter.

Characterisation 66 (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.

10.1 Construction of Model of Domain Intrinsics

slide 356

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.

10.2 Overview of Support Example

slide 357

10.2.1 Entities

slide 358

10.2.2 Operations

slide 359

10.2.3 Events

slide 360

10.2.4 Behaviours

slide 361

10.3 Principles, Techniques and Tools

slide 362

10.4 Discussion

slide 363

10.5 Exercises

10.5.1 7.a

Solution S.8.1 on page 330, suggests an answer to this exercise.

10.5.2 7.b

Solution S.8.2 on page 330, suggests an answer to this exercise.

10.5.3 7.c

Solution S.8.3 on page 330, suggests an answer to this exercise.

10.5.4 7.d

slide 364 Solution S.8.4 on page 331, suggests an answer to this exercise.

11

Domain Modelling: Support Technologies slide 367

Appendix H (Pages 227–227) complements the present chapter.

Characterisation 67 (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. ■

11.1 Technology as an Embodiment of Laws of Physics slide 368

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

11.1.1 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.

11.2 Intrinsics versus Other Facets slide 369

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.

11.3]

The Support Example[s] slide 370

The points made in the last paragraph above are illustrated in the examples of an intrinsic concepts of states versus the examples of a corresponding support technology concepts of states

- (intrinsics)
- (intrinsics)
- (intrinsics)

slide 371

11.4 Principles, Techniques and Tools slide 372

11.5 Discussion slide 373

11.6 Exercises

11.6.1 8.a

Solution S.9.1 on page 331, suggests an answer to this exercise.

11.6.2 8.b

Solution S.9.2 on page 331, suggests an answer to this exercise.

11.6.3 8.c

Solution S.9.3 on page 331, suggests an answer to this exercise.

11.6.4 8.d

slide 374 Solution S.9.4 on page 331, suggests an answer to this exercise.

12 Domain Modelling: Management and Organisation

slide 377

Appendix I (Pages 229–229) complements the present chapter.

12.1 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[1]. 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 378

Characterisation 68 (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. ▪

12.1.1 Management Issues

slide 379

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

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

manipulation of human resources, (xii) financial resources, (xiii) technological resources, and (xiv) natural resources

12.1.2 Basic Functions of Management slide 380

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.

12.1.3 Formation of Business Policy slide 381

(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[2]. 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. Formation of Business Policy 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 382

12.1.4 Implementation of Policies and Strategies slide 383

(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.

[2] 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.

12.1.5 Development of Policies and Strategies slide 384

(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.

12.1.6 Management Levels slide 385

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.

12.1.7 Resources slide 386

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).

12.1.8 Resource Conversion slide 387

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.

12.1.9 Strategic Management slide 388

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 389

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 390

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.

12.1.10 Tactical Management slide 391

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 392

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

12.1.11 Operational Management slide 393

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 394

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 395 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.

12.1.12 Supervisors and Team Leaders slide 396

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 397

Supervisors basically monitor that work processes are carried out as planned and report other than minor discrepancies. Team leaders coordinate work in a group ("the team") while participating in that work themselves; additionally they are also supervisors.

12.1.13 Description of 'Management' slide 398

On the last several pages (101–105) 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 101–105 are generally "too lofty", "too woolly", that is, are more about "feelings" than about "hard, observable facts". slide 399

We, for example, consider the following issues for "too lofty", "too woolly": Item (xix) Page 102: "to enable the successful ..." is problematic; Item (xx)

Page 102: how to check that managers “exhibit these skills” ?; Item (xxi) Page 102: “play an effective part” is problematic; Item (xxvii) Page 102: how to check that vision is being or is achieved ?; Item (xxviii) Page 102: the objectives must, in order to be objectively checked, be spelled out in measurable details; Item (xxxi) Page 102: how to check “flexible” and “easily”; Item (xxxiii) Page 102: how to check that the deployed resources are those that contribute to “achieving vision and long term objectives; Item (xxxiv) Page 102: “guideline”, “factors of production” and “advantage” cannot really be measured; Item (xxxv) Page 102: “what type of business they want to form” is too indeterminate; Item (xxxvi) Page 102: 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 102: how does one describe “managers must, or usually understand where and how” ?; Item (xxxix) Page 102: in what does a review actually consists ?; Item (xli) Page 102: how does one objectively describe “a good environment” ?; Item (xlii) Page 103: how does one objectively describe that which is being “analysed”, the “analysis” and the “determination” processes ?; Item (xliii) Page 103: how is the “development” and a “picture” objectively described ?; etcetera. slide 400

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 401

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 ?

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.

12.2 Organisation slide 402

12.2.1 .1. slide 403

12.2.2 .2. slide 404

12.2.3 .3. slide 405

12.3]

The Support Example[s] slide 406

12.4 Principles, Techniques and Tools slide 407

12.5 Exercises

12.5.1 9.a

Solution S.10.1 on page 331, suggests an answer to this exercise.

12.5.2 9.b

Solution S.10.2 on page 331, suggests an answer to this exercise.

12.5.3 9.c

Solution S.10.3 on page 331, suggests an answer to this exercise.

12.5.4 9.d

Solution S.10.4 on page 331, suggests an answer to this exercise. slide 408

13 Domain Modelling: Rules and Regulations slide 411

Appendix J (Pages 231–231) complements the present chapter.

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. slide 412

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[1], and/or by public regulatory agencies.

13.1 Domain Rules slide 413

Characterisation 69 (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 414

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.

[1] 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.

13.2 Domain Regulations

slide 415

Characterisation 70 (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 416

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.

13.3 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 417

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 418

A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \rightarrow \Theta$, 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, $(\Theta \times \Theta) \rightarrow$ **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 419

```
type
    Stimulus, Rule, $\Theta$
    STI = $\Theta \rightarrow \Theta$
    RUL = $(\Theta \times \Theta) \rightarrow$ Bool
value
    meaning: Stimulus $\rightarrow$ STI
    meaning: Rule $\rightarrow$ RUL

    valid: Stimulus $\times$ Rule $\rightarrow \Theta \rightarrow$ Bool
    valid(sy_sti,sy_rul)($\theta$) $\equiv$ meaning(sy_rul)($\theta$,(meaning(sy_sti))($\theta$))
    valid: Stimulus × RUL → Θ → Bool
    valid(sy_sti,se_rul)(θ) ≡ se_rul(θ,(meaning(sy_sti))(θ))
```

slide 420

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. slide 421 The two kinds of functions express: If the predicate holds, then the action can be applied.

The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function. slide 422

```
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 423

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 424

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 $\theta' =$ (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 425

```
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 426

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

13.4 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 427 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 [6] or some Fuzzy Logic notations [188].

13.5]

The Support Example[s] slide 428

13.6 Principles, Techniques and Tools slide 429

13.7 Exercises

13.7.1 10.a

Solution S.11.1 on page 332, suggests an answer to this exercise.

13.7.2 10.b

Solution S.11.2 on page 332, suggests an answer to this exercise.

13.7.3 10.c

Solution S.11.3 on page 332, suggests an answer to this exercise.

13.7.4 10.d

slide 430 Solution S.11.4 on page 332, suggests an answer to this exercise.

14

Domain Modelling: Scripts, Licenses and Contracts

slide 433

Appendix K (Pages 233–280) complements the present chapter.

Characterisation 71 (Domain Script) By a **domain script** we mean the structured wording of a set of rules and regulation. ▪

Characterisation 72 (Domain License) By a **domain license** we mean a script that prescribes a desirable set of behaviours. ▪

Characterisation 73 (Domain Contract) By a **domain contract** we mean a license that additionally has legally binding power, that is, which may be contested in a court of law. ▪

14.1 Analysis of Examples

slide 434

We refer to Appendix K.

14.1.1 Timetables

slide 435

We refer to Appendix Sect. K.2.1 Page 233–243

The bus/train timetable is informally sketched. Sect. K.2.1 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.).

14.1.2 Aircraft Flight Simulator Script slide 436

We refer to Appendix Sect. K.2.3 Pages 243–244

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.

14.1.3 Bill of Lading slide 437

We refer to Appendix Sect. K.2.4 Page 244–246

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. 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.

14.2 Licenses slide 438

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 [179]

slide 439

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 440

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 441

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.

14.3 The Support Example(s) slide 442

14.4 Principles, Techniques and Tools slide 443

14.5 Exercises

14.5.1 11.a

Solution S.12.1 on page 332, suggests an answer to this exercise.

14.5.2 11.b

Solution S.12.2 on page 332, suggests an answer to this exercise.

14.5.3 11.c

Solution S.12.3 on page 332, suggests an answer to this exercise.

14.5.4 11.d

Solution S.12.4 on page 332, suggests an answer to this exercise. slide 444

slide 445–446

15 Domain Modelling: Human Behaviour

slide 447

Appendix L (Pages 283–283) complements the present chapter.

Characterisation 74 (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. ■

15.1 A Meta-characterisation of Human Behaviour

slide 448

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 = $\Theta \overset{\sim}{\to} \Theta$-infset
value
    hum_int: Rule $\to \Theta \to$ RUL-infset
    action: Stimulus $\to \Theta \to \Theta$
    hum_beha: Stimulus $\times$ Rules $\to$ Action $\to \Theta \overset{\sim}{\to} \Theta$-infset
    hum_beha(sy_sti,sy_rul)($\alpha$)($\theta$) as $\theta$set
        post
            $\theta$set = $\alpha(\theta) \wedge$ action(sy_sti)($\theta$) $\in \theta$set
             $\wedge \forall \theta'$:$\Theta$•$\theta' \in \theta$set $\Rightarrow$
               $\exists$ se_rul:RUL•se_rul $\in$ hum_int(sy_rul)($\theta$)$\Rightarrow$se_rul($\theta$,$\theta'$)
```

slide 449

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 450

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.

15.2 Review of Support Examples slide 451

slide 452 TO BE WRITTEN

TO BE WRITTEN

15.3 On Modelling Human Behaviour slide 453

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.

15.4]

The Support Example[s] slide 454

15.5 Principles, Techniques and Tools slide 455

15.6 Exercises

15.6.1 12.a

Solution S.13.1 on page 333, suggests an answer to this exercise.

15.6.2 12.b

Solution S.13.2 on page 333, suggests an answer to this exercise.

15.6.3 12.c

Solution S.13.3 on page 333, suggests an answer to this exercise.

15.6.4 12.d

slide 456 Solution S.13.4 on page 333, suggests an answer to this exercise.

Part IV

ANALYTIC STAGES

16

Verification

slide 459

16.1 Theorem Proving

slide 460

16.2 Model Checking

slide 461

16.3 Formal Testing

slide 462

16.4 Combining Proofs, Checks and Tests

slide 463

16.5 Principles, Techniques and Tools

slide 464

16.6 Discussion

slide 465

16.7 Exercises

16.7.1 13.a

Solution S.14.1 on page 333, suggests an answer to this exercise.

16.7.2 13.b

Solution S.14.2 on page 333, suggests an answer to this exercise.

16.7.3 13.c

Solution S.14.3 on page 333, suggests an answer to this exercise.

16.7.4 13.d

Solution S.14.4 on page 333, suggests an answer to this exercise.

slide 466

slide 467–468

17 Validation

slide 469

17.1 Principles, Techniques and Tools

slide 470

17.2 Discussion

slide 471

17.3 Exercises

17.3.1 14.a

Solution S.15.1 on page 333, suggests an answer to this exercise.

17.3.2 14.b

Solution S.15.2 on page 334, suggests an answer to this exercise.

17.3.3 14.c

Solution S.15.3 on page 334, suggests an answer to this exercise.

17.3.4 14.d

Solution S.15.4 on page 334, suggests an answer to this exercise. slide 472

18 Theory Formation

slide 475

18.1 Principles, Techniques and Tools

slide 476

18.2 Discussion

slide 477

18.3 Exercises

18.3.1 15.a

Solution S.16.1 on page 334, suggests an answer to this exercise.

18.3.2 15.b

Solution S.16.2 on page 334, suggests an answer to this exercise.

18.3.3 15.c

Solution S.16.3 on page 334, suggests an answer to this exercise.

18.3.4 15.d

Solution S.16.4 on page 334, suggests an answer to this exercise. slide 478

Part V

DOMAIN ENGINEERING: A POSTLUDIUM

slide 479–480

19 Domain Engineering: A Postludium slide 481

19.1 Consolidation of Domain Modelling Facets slide 482

The many 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. slide 483

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 484

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

$\boxed{\text{More to come}}$

slide 485

19.2 Domain Engineering Documents slide 486

19.3 Generic Domain Engineering Development Graph slide 487

19.4 Principles, Techniques and Tools slide 488

19.5 Discussion slide 489

19.6 Exercises

19.6.1 15.a.x

Solution S.17.1 on page 334, suggests an answer to this exercise.

19.6.2 15.b.x

Solution S.17.2 on page 334, suggests an answer to this exercise.

19.6.3 15.c.x

Solution S.17.3 on page 335, suggests an answer to this exercise.

19.6.4 15.d.x

slide 490 Solution S.17.4 on page 335, suggests an answer to this exercise.

Part VI

CLOSING

slide 491–492

20 From Domains to Requirements slide 493

Appendix M (Pages 285–285) complements the present chapter.

20.1 Principles and Concepts of Requirements slide 494

20.2 Stages of Requirements Engineering slide 495

20.2.1 Informative Documents slide 496

20.2.2 Stakeholder Identification slide 497

20.2.3 Requirements Acquisition slide 498

20.2.4 Business Process Re-engineering slide 499

20.2.5 Requirements Analysis and Concept Formation slide 500

20.2.6 Requirements Terminology slide 501

20.2.7 Requirements Modelling slide 502

20.2.8 Requirements Model Verification slide 503

20.2.9 Requirements Model Validation slide 504

20.2.10 Requirements Feasibility and Satisfiability slide 505

20.2.11 Requirements Theory Formation slide 506

20.3 Shared Phenomena and Concepts slide 507

We (forward) refer to appendix example Sect. M.1 on page 285. It follows up on this methodology concept.

20.4 Domain Requirements slide 508

We (forward) refer to appendix example Sect. M.2 on page 285. It follows up on this methodology concept.

20.5 Interface Requirements slide 509

We (forward) refer to appendix example Sect. M.3 on page 285. It follows up on this methodology concept.

20.6 Machine Requirements slide 510

We (forward) refer to appendix example Sect. M.4 on page 285. It follows up on this methodology concept.

20.7 Discussion slide 511

20.8 Requirements Engineering Management slide 512

20.8.1 Requirements Engineering Development Graph slide 513

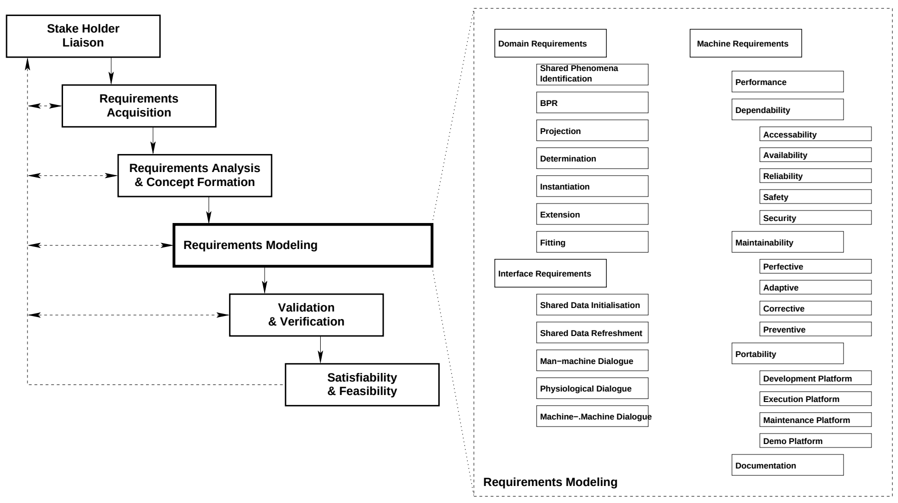

Fig. 20.1. Requirements engineering process graph

20.8.2 Requirements Engineering Development Documents slide 514

20.9 Exercises

20.9.1 16.a

Solution S.18.1 on page 335, suggests an answer to this exercise.

20.9.2 16.b

Solution S.18.2 on page 335, suggests an answer to this exercise.

20.9.3 16.c

Solution S.18.3 on page 335, suggests an answer to this exercise.

20.9.4 16.d

Solution S.18.4 on page 335, suggests an answer to this exercise. slide 515

slide 516–517

21

Summary and Conclusion slide 518

21.1 A Domain Engineering Development Graph slide 519

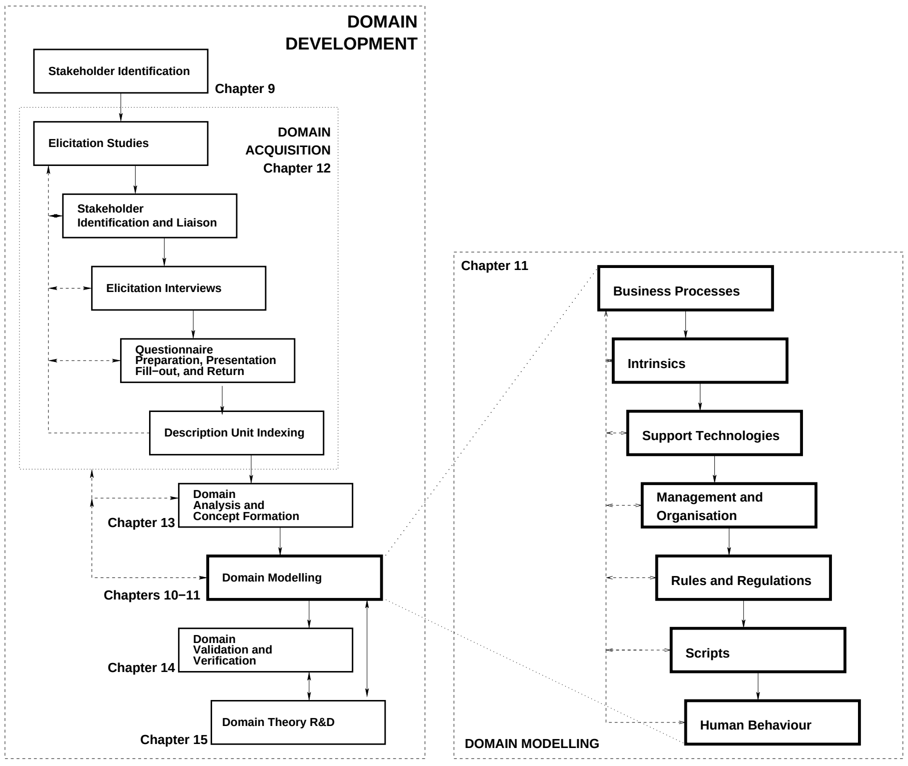
DOMAIN DEVELOPMENT
Stakeholder Identification
Chapter 9
DOMAIN ACQUISITION Chapter 12
Elicitation Studies
Stakeholder Identification and Liaison
Elicitation Interviews
Questionnaire Preparation, Presentation Fill−out, and Return
Description Unit Indexing
Domain Analysis and Concept Formation
Chapter 13
Domain Modelling
Chapters 10−11
Domain Validation and Verification
Chapter 14
Domain Theory R&D
Chapter 15
Chapter 11
Business Processes
Intrinsics
Support Technologies
Management and Organisation
Rules and Regulations
Scripts
Human Behaviour
DOMAIN MODELLING

Fig. 21.1. Domain engineering process graph

21.2 Domain Engineering Development Documents slide 520

There are basically three kinds of domain development documents:

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

We have already covered, in Chap. 3, the concept of informative documents.

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

21.2.1 Description Documents slide 521

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

21.2.2 Analytic Documents slide 522

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

21.3 slide 523

21.4 slide 524

21.5 slide 525

21.6 Exercises

21.6.1 17.a

Solution S.19.1 on page 335, suggests an answer to this exercise.

21.6.2 17.b

Solution S.19.2 on page 335, suggests an answer to this exercise.

21.6.3 17.c

Solution S.19.3 on page 335, suggests an answer to this exercise.

21.6.4 17.d

Solution S.19.4 on page 335, suggests an answer to this exercise. slide 526

Part VII

THE DOMAIN DEVELOPMENT EXAMPLE

slide 527

A Informative Documentation slide 528

Chapter 3 (Pages 47–64) complements the present appendix.

A.1 Project Names and Dates slide 529

We (backward) refer to methodology Sect. 3.2 on page 48. It provides the methodological background for the present section. slide 530

A.2 Project Partners and Places slide 531

We (backward) refer to methodology Sect. 3.3 on page 48. It provides the methodological background for the present section. slide 532

A.3 Current Situation slide 533

We (backward) refer to methodology Sect. 3.4 on page 49. It provides the methodological background for the present section. slide 534

A.4 Needs and Ideas slide 535

A.4.1 Needs slide 536

We (backward) refer to methodology Sect. 3.5.1 on page 49. It provides the methodological background for the present section. slide 537

A.4.2 Ideas slide 538

We (backward) refer to methodology Sect. 3.5.2 on page 50. It provides the methodological background for the present section. slide 539

A.4.3 Discussion slide 540

A.5 Facilities and Concepts slide 541

We (backward) refer to methodology Sect. 3.6 on page 50. It provides the methodological background for the present section. slide 542 slide 543

A.6 Scope and Span slide 544

A.6.1 Scope slide 545

We (backward) refer to methodology Sect. 3.7 on page 51. It provides the methodological background for the present section.

A.6.2 Span slide 546

We (backward) refer to methodology Sect. 3.7 on page 51. It provides the methodological background for the present section.

A.7 Assumptions and Dependencies slide 547

We (backward) refer to methodology Sect. 3.8 on page 51. It provides the methodological background for the present section.

A.7.1 Assumptions slide 548

A.7.2 Dependencies slide 549

A.8 Implicit & Derivative Goals slide 550

We (backward) refer to methodology Sect. 3.9 on page 52. It provides the methodological background for the present section. slide 551

A.9 Synopsis slide 552

We (backward) refer to methodology Sect. 3.10 on page 52. It provides the methodological background for the present section. slide 553

A.10 Domain Development Graph slide 554

We (backward) refer to methodology Sect. 3.11 on page 53. It provides the methodological background for the present section. slide 555

A.11 Resource Allocation slide 556

We (backward) refer to methodology Sect. 3.12 on page 55. It provides the methodological background for the present section. slide 557

A.12 Budget (and Other) Estimates slide 558

A.12.1 Budget slide 559

We (backward) refer to methodology Sect. 3.13.1 on page 56. It provides the methodological background for the present section.

A.12.2 Other Estimates slide 560

We (backward) refer to methodology Sect. 3.13.2 on page 56. It provides the methodological background for the present section.

A.13 Standards Compliance slide 561

We (backward) refer to methodology Sect. 3.14 on page 56. It provides the methodological background for the present section.

A.13.1 Development Standards slide 562

A.13.2 Documentation Standards slide 563

A.13.3 Standards versus Recommendation slide 564

A.13.4 Specific Standards slide 565

A.14 Contracts and Design Briefs slide 566

A.14.1 Contracts slide 567

We (backward) refer to methodology Sect. 3.15.1 on page 59. It provides the methodological background for the present section.

A.14.2 Design Briefs slide 568

We (backward) refer to methodology Sect. 3.15.3 on page 62. It provides the methodological background for the present section.

A.15 Development Logbook slide 569

We (backward) refer to methodology Sect. 3.16 on page 63. It provides the methodological background for the present section. slide 570

A.16 Discussion slide 571

slide 572
slide 573

B

Stakeholders

slide 575

Chapter 4 (Pages 67–68) complements the present appendix. slide 576

slide 577

C

Domain Acquisition slide 578

Chapter 5 (Pages 71–73) complements the present appendix.

C.1 Initial Acquisition Steps slide 579

In the initial domain acquisition we look for a “suitable” number of simple entities, operations, events and behaviours of the domain.

C.1.1 Simple Entities slide 580

It appears that the most obvious simple entities of the oil and natural gas industry are those of oil and gas, oil and gas fields, pipelines with pipe, pumps, valves, forks, joins, etcetera, oil and gas storage (depots), refineries and gas turntables, and oil (and gas) tankers. These are enough, for the moment, for us to get “boot-strapped” into finding more simple entities, and for finding suitable operations, events and behaviours.

C.1.2 Operations slide 581

Similarly, it appears that the most obvious operations of the oil and natural gas industry are those of building pipelines, building refineries, building depots, opening oil fields, and facilities for oil and gasoline distribution; and of pumping oil or gas, controlling valves, storing petroleum, shipping oil by tankers, oil refining, and bringing gasoline to end customers. Many related operations now arise: measuring flow of oil and gas, routing of oil flow, and boosting oil flow, compressing gas, etcetera.

C.1.3 Events slide 582

Events of the oil and natural gas industry are those of empty oil reservoirs, (over)full storage depots, oil and gas leaks, malfunctioning of valves, pumps, etcetera.

C.1.4 Behaviours slide 583

Three examples of behaviours can be identified: (i) The behaviour of a drain pump: its installation at an oil reservoir; its first deployment (start of pumping), i.e., its commissioning, or putting into service; its cyclic alternations between pumping, not pumping, being serviced and repaired; ending with its decommissioning: being taken out of service. (ii) The processing of oil: from being drained from oil fields, into a pipeline, onto a depot, from there, via oil tankers, to other depots, being refined into gasoline etc., and final distribution to end customers; slide 584 and (ii) the behaviour of the whole industry: the interaction between several oil fields, pipeline systems depots, shipping facilities, refineries (and gas processing plants), and end customers.

C.2 System Composition slide 585

We can rough sketch an oil & natural gas industry by diagramming it, as shown in Fig. C.1. Here we show an oil refinery — so the diagram appears to depict a petroleum system. One could show a similar diagram for a natural gas system.

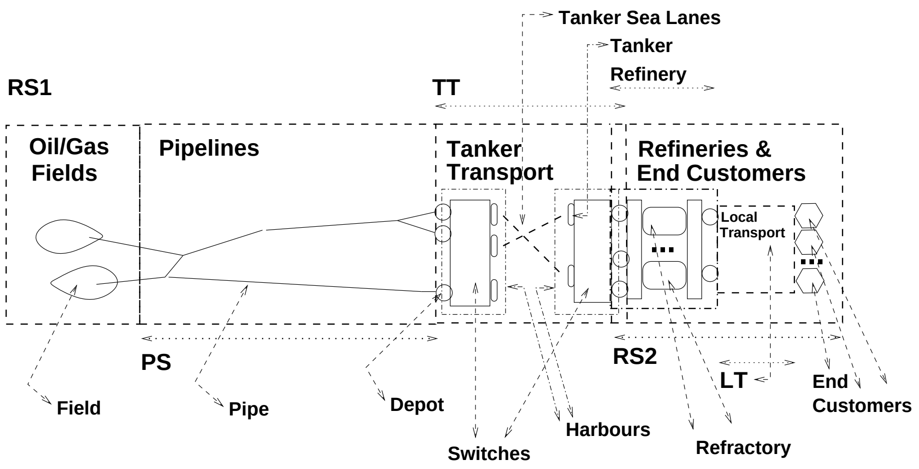

Fig. C.1. A Schematic of an Oil & Natural Gas Industry

slide 586

Narrative

2. The petroleum industry, Ω, is here thought of as consisting of three subdomains:
   (a) oil and natural gas fields, refineries and end-consumers (sinks), Ω_{RS},

(b) pipeline systems, Ω_{PS}, and
(c) tanker transport systems, Ω_{TT}.

There may be many other aspects of the petroleum industry than "strictly" limitable to these three distinct sub-domains: the OPEC[1], issues of the oil market[2], in particular the New York Mercantile Exchange[3] or International Petroleum Exchange[4] in London, etcetera. We leave out modelling these and other aspects of the petroleum industry. We claim that their modelling can relatively easily be "added" in a later phase of domain description. slide 587

Formalisation

type
  2. Ω, Ω_{RS}, Ω_{PS}, Ω_{TT}
value
  2(a). obs_Ω_{RS}: $\Omega \to \Omega_{\mathrm{RS}}$
  2(b). obs_Ω_{PS}: $\Omega \to \Omega_{\mathrm{PS}}$
  2(c). obs_Ω_{TT}: $\Omega \to \Omega_{\mathrm{TT}}$

When we later decide to include considerations of OPEC, trading (e.g., ICE), regulation, etc. then we simply add further observation functions over Ω while ensuring that these further sub-domains "interface" properly with Ω_{RS}, Ω_{PS} and Ω_{TT}.

C.2.1 Pipeline Systems slide 588

slide 589

Narrative

3. A pipeline system consists of one or more pipes and two or more nodes.
4. Pipes and nodes have unique identifiers.
5. A pipeline node is either a drain pump, a fill pump, a valve, a flow pump, a fork, a join, or an end customer, i.e., a sink (node).
6. From a pipe one can observe the (directed) pair of identifiers of the two nodes connected to the pipe, or, put differently, the two nodes that the pipe connects.
7. A drain pump is connected to a reservoir and a pipe and one can thus observe the identifiers of the reservoir and the pipe. slide 590
8. A fill pump is connected to a pipe and a deport and one can thus observe the identifiers of the pipe and depot.

[1] OPEC: The Vienna, Austria-based Organisation for Petroleum Exporting Countries, see, for example, http://www.opec.org/home/
[2] See for example http://www.eia.doe.gov/pub/oil_gas/petroleum/analysis_publications/oil_market_basics/default.htm
[3] NYMEX: http://www.nymex.com/index.aspx
[4] See IPE: http://en.wikipedia.org/wiki/International_Petroleum_Exchange acquired in June 2001 by ICE: https://www.theice.com/homepage.jhtml

9. A valve connects one or more pipes (called input pipes) to one or more (output) pipes, and one can thus observe the set of one or more identifiers of the input pipes and the set of one or more identifiers of the output pipes.
10. A flow pump connects one input pipe to one output pipe, and one can thus observe the pair of identifiers of the input and output pipes.
11. A join connects two or more pipes (called input pipes) to one (output) pipe, and one can thus observe the set of two or more identifiers of the input pipes and the set of the one identifier of the output pipe. slide 591
12. A fork connects one pipe (called the input pipe) to two or more (output) pipes, and one can thus observe the set of one identifiers of the input pipe and the set of two or more identifiers of the output pipes.

By a pipe we mean a circular tube. From one or more pipes (also referred to as pipe segments) one can construct a pipeline by concatenating one or more pipes. slide 592

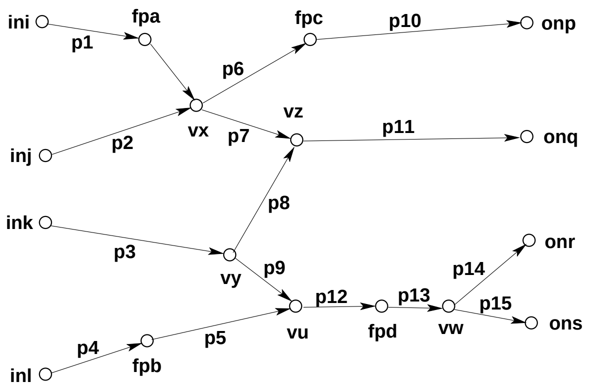

Fig. C.2. A Pipeline System: in: input node (drain pump, valve or flow pump), on: output node (valve or flow pump), fp: flow pump, v: valve

slide 593

Formalisation

type
3. Ω_{PS}, P, N
3. Node == mkN(n:N)
3. Pipe == mpP(p:P)
4. PI, NI
5. DraP, FilP, Valv, FloP, Join, Fork, Sink
5. N == DrP(fdp:DraP) | FiP(fp:FilP) | Val(v:Valv) |
FlP(fp:FloP) | Jn(j:Join) | Fk(f:Fork) | Snk(si:Sink)

value
3. obs_Ps: $\Omega_{\mathrm{PS}} \to$ P-**set**
3. obs_Ns: $\Omega_{\mathrm{PS}} \to$ N-**set**
4. obs_PI: P $\to$ PI
4. obs_NI: N $\to$ NI

axiom
3. $\forall\ \omega_{\mathrm{PS}}{:}\Omega_{\mathrm{PS}}$ • **card** obs_Ps(ω_{PS}) $\geq 1 \wedge$ **card** obs_Ns(ω_{PS}) ≥ 2

value
6. obs_NIp: P $\to$ (NI $\times$ NI)
7.–12. obs_pPIs: N $\to$ (PI-**set** $\times$ PI-**set**)

A Technical Comment:

- The A == mk(s:B) | ... type definition[5] defines type A values to be constructed from type B values by means of the constructor mk.
  Thus is a technicality. It is used in order to secure that no matter which properties with which we are (later) going to endow the various kinds of nodes — that — the node kind will always be uniquely distinguishable.
  That is, it may be that the properties we ascribe, for example, to fill pumps, fp:FilP, and sinks, snk:Sink, are the same (!), but the distinct constructor names, FiP and Snk secure that one can distinguish fill pumps from sinks.

slide 594

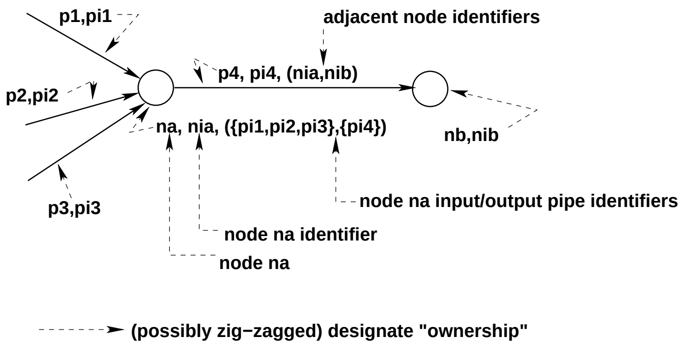
p1,pi1
adjacent node identifiers
p4, pi4, (nia,nib)
p2,pi2
na, nia, ({pi1,pi2,pi3},{pi4})
nb,nib
p3,pi3
node na input/output pipe identifiers
node na identifier
node na
(possibly zig−zagged) designate "ownership"

Fig. C.3. Nodes and pipes, node and pipe identifiers and adjacent node and pipe identifiers

C.2.2 Tanker Transport Systems

slide 595

Narrative

13. A tanker transport system consists of two or more harbours, one or more tanker ships (tanker), and one or more sea lanes.

[5] Where A could be N and mk, s and B could be FiP, fp and FilP.

14. Harbours, tankers and sea lanes have unique identifiers.

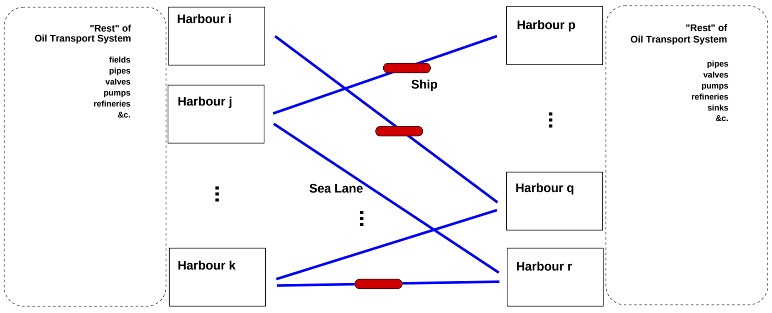

Fig. C.4. Tanker Transport System with Harbours, Sea Lanes and Ships

slide 596

Formalisation

type
  13. H, T, L
value
  13. obs_Hs: $\Omega_{TT} \to$ H-**set**
  13. obs_Ts: $\Omega_{TT} \to$ T-**set**
  13. obs_Ls: $\Omega_{TT} \to$ L-**set**
axiom
  13. $\forall\ \omega_{TT}:\Omega_{TT}$ •
      card obs_Hs(ω_{TT})$\geq$2 $\wedge$
      card obs_Ts(ω_{TT})$\geq$1 $\wedge$
      card obs_Ls(ω_{TT})$\geq$1
type
  14. HI, TI, LI
axiom
  14. $\forall$ hi:HI, ti:TI, li:LI • hi$\neq$ti $\wedge$ ti$\neq$li $\wedge$ hi$\neq$li
value
  14. obs_HI: H $\to$ HI
  14. obs_TI: T $\to$ TI
  14. obs_LI: L $\to$ LI

slide 597

Narrative

slide 598

15. From a harbour one can observe
    (a) a set of one or more depots,

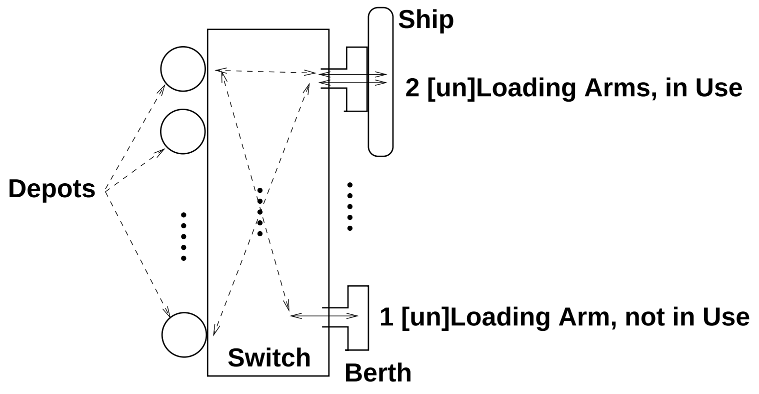

Fig. C.5. Harbour with Depots, Switch, Berths and Ship

(b) a switch[6] (connecting depots to loading arms, see below, Item 15(f)),
(c) a set of one or more berths,
(d) the identification of the sea lanes emanating from that harbour,
(e) the identifications of the sea lanes incident upon that harbour,
(f) and, from a berth, one or more loading arms

slide 599

Formalisation

type
 15(a). Depot, Switch, Berth, LdArm, Tank, Length
value
 15(a). obs_Depots: H → Depot-**set**
 15(b). obs_Switch: H → Switch
 15(c). obs_Berths: H → Berth-**set**
 15(d). obs_eLIs: H → LI-**set**
 15(e). obs_iLIs: H → LI-**set**
 15(f). obs_LdArms: Berth → LdArm-**set**
axiom
  ∀ h:H •
 15(a). **card** obs_Depots(h)≥1 ∧
 15(c). **card** obs_Berths{h}≥1 ∧
 15(d). **card** obs_eLIs(h)≥1 ∧
 15(e). **card** obs_eLIs(h)≥1 ∧
 15(f). ∀ b:B • b ∈ obs_Berths{h} ⇒ obs_LdArms(b)≥1

slide 600

[6] Whether we consider an oil loading/unloading, i.e., exporting/importaing harbour to contain one swithch or more switches turns out, as we shall see, to be immaterial.

Narrative, Continued

16. From a sea lane one can observe
    (a) the pair of distinct identifiers of the harbours from, respectively to which the lane is directed, and
    (b) its length.

Formalisation, Continued

```
value
   16(a).  obs_HIp: L → HI×HI
   16(b).  obs_Len: L → Length
axiom
   16(a).  ∀ l:L • let (hi,hi′)=obs_HIp(l) in hi≠hi′ end,
```

slide 601

Narrative, Continued

17. From a tanker one can observe
    (a) a set of one or more tanks
    (b) whether the tanker is at sea,
    (c) and, if so, along which sea lane it sails,
    (d) and, if not, at which berth and in which harbour it is moored.

slide 602

Formalisation, Continued

```
value
   17(a).  obs_Tanks: Tanker → Tank-set
   17(b).  is_at_Sea: T → Bool
   17(c).  obs_L: T $\overset{\sim}{\to}$ L, pre: obs_L(t): is_at_Sea(t)
   17(d).  obs_HIBI: T $\overset{\sim}{\to}$ (HI×BI), pre: obs_HIBI(t): ∼is_at_Sea(t)
axiom
   17(a).  ∀ t:T • card obs_Tanks(t)≥1 ∧
        ∀ $\omega_{TT}$:$\Omega_{TT}$ •
           ∀ t:T • t ∈ obs_Ts($\omega_{TT}$) •
   17(d).  ∼is_at_Sea(t) ⇒
                  let (hi,bi)=obs_HIBI(t) in
                  ∃ h:H • h ∈ obs_Hs($\omega_{TT}$) ∧ hi=obs_HI(h) ⇒
                     ∃ b:Berth • b ∈ obs_Berths(h) • bi=obs_BI(b) end
```

C.2.3 Oil Field and Refinery Systems

slide 603

slide 604

Narrative

18. The oil/gas resource management domain consists of
    (a) one or more oil/gas fields,
    (b) one or more depots,
    (c) one or more oil refineries,
    (d) one or more local (oil and/or natural gas) distribution nets, and
    (e) one or more end-customers (referred to as sinks).
19. Oil/gas fields, depots, oil refineries, distribution nets and end-customers all have unique identifiers.

slide 605

```
type
  18. Field, Depot, Refinery, DistrNet, EndCust
  19. FI, DI, RI, NI, EI
value
  18(a). obs_Fields: $\Omega_{RS}$ → Field-set
  18(b). obs_Depots: $\Omega_{RS}$ → Depot-set
  18(c). obs_Refineries: $\Omega_{RS}$ → Refinery-set
  18(d). obs_EndCust: $\Omega_{RS}$ → EndCust-set
  18(e). obs_DistrNets: $\Omega_{RS}$ → DistrNet-set
  19. obs_FI: Field → FI
  19. obs_DI: Depot → DI
  19. obs_RI: Refinery → RI
  19. obs_NI: DistrNet → DNI
  19. obs_SI: EndCust → EI
axiom
  18. ∀ $\omega_{RS}$:$\Omega_{RS}$
          card obs_Fields($\omega_{RS}$)≥1 ∧
          card obs_Depots($\omega_{RS}$)≥1 ∧
          card obs_Refineries($\omega_{RS}$)≥1 ∧
          card obs_EndCusts($\omega_{RS}$)≥1 ∧
          card obs_DistrNetss($\omega_{RS}$)≥1
```

slide 606

Narrative

20. An oil/gas field contains
    (a) a reservoir of oil and/or gas,
    (b) one or more drain pumps and
    (c) zero, one or more fill pumps.
21. And
    (a) Reservoirs and
    (b) oil/gas (drain or fill) pumps

    have unique identifiers.

Formalisation

type
  20. Reservoir, DrP(dp:DraP), FiP(fp:FilP)
value
  20(a). obs_Reservoir: Field → Reservoir
  20(b). obs_DrainPumps: Field → DrP(dp:DraP)-**set**
  20(c). obs_FillPumps: Field → FiP(fp:FilP)-**set**
axiom
  20(c). ∀ f:Field • **card** obs_DrainPumps(f)≥1
type
  21. RI, DPI, FPI
value
  21(a). obs_RI: Reservoir → RI
  21(b). obs_DPI: DrainPump → DPI
  21(b). obs_FPI: FillPump → FPI

slide 607

Narrative

22. An oil refinery consists of
    (a) one or more input depots,
    (b) one or more refractory towers,
    (c) a switch system of pipes “running” from input depots to refractory towers,
    (d) one or more output depots,
    (e) and a switch system of pipes “running” from refractory towers to output depots.
23. Depots and Refractory towers have unique identifiers.

slide 608

Formalisation

type
  22. Refractory, Switch
  23. RTI
value
  22(a). obs_InDepots: Refinery → Depot-**set**
  22(b). obs_Refractories: Refinery → Refractory-**set**
  22(c). obs_InSwitch: Refinery → Switch
  22(d). obs_OutDepots: Refinery → Depot-**set**
  22(e). obs_OutSwitch: Refinery → Switch
  23. obs_RTI: Refractory → RTI
  23. obs_DI: Depot → DI
axiom
    ∀ r:Refinery •

```
22(a).  obs_InDepots(r)≥1 ∧
22(b).  obs_Refractories(r)≥1 ∧
22(d).  obs_OutDepots(r)≥1 ∧
23.     let ids=obs_InDepots(r), ods=obs_InDepots(r) in
          ∀ d,d′:Depot•d≠d′∧{d,d′}⊆ids ∪ ods ⇒ obs_DI(d)≠obs_DI(d′) end
```

C.3 Petroleum and Gas

slide 609

So far, we have not distinguished between (crude) oil (i.e., petroleum) and gas retrieval, transportation (pipe lines and tanker and truck transport), processing and disposal. To complement the concept of refineries we introduce the concept of gas separators. slide 610

For convenience we now introduce the notion of units: Pipes and nodes, that is, reservoirs, pumps, valves, forks, joins, depots, switches, refractory towers, gas separators, loading arms, tanks, sinks (i.e., end customers) are units.

Units will be properly formalised in a subsequent appendix, cf. Sect. E.1.1 Items 29–33 (Pages 169–172). slide 611

24. The various sub-domains treated so far either processes oil or natural gas.[7]
25. To each and every node and pipe, that is, unit we associate the "kind" attribute: `"oil"` or `"gas"`.
26. For an entire system, $\omega:\Omega$,
    (a) the system is either of kind `"oil"` or of kind `"gas"`
    (b) and all units are of that same kind.
27. Thus a `"gas"` system, $\omega:\Omega$,
    (a) has no refineries;
    (b) instead it has gas separators.
28. Vice versa, an `"gas"` system, $\omega:\Omega$,
    (a) has no refineries;
    (b) instead it has gas separators.

slide 612

```
type
  24.  Oil, Gas, U, GaFi, Refi
  25.  Kind == oil | gas
value
  25.  obs_Kind: (Ω|U) → Kind
  25.  obs_Us: Ω → U-set
  25.  is_GaFi, is_Refi: U → Bool
axiom
```

[7] Without loss of generality we do not consider varieties of crude oil (Brent oil, etc.) or naural gas, nor do we consider reservoirs and drain pumps that contain, respectively handle both oil and gas.

```
26.    ∀ ω:Ω •
26.      ∀ u:U • u ∈ obs_Us(ω) • obs_Kind(ω) ≡ obs_Kind(u) ∧
27.      gas=obs_Kind(ω) ⇒
27(a).        ∼exist u:U • u ∈ obs_Us(ω) ∧ is_Refi(u) ∧
27(b).        exist u:U • u ∈ obs_Us(ω) ∧ is_GaFi(u)
28.      oil=obs_Kind(ω) ⇒
28(a).        ∼exist u:U • u ∈ obs_Us(ω) ∧ is_GaFi(u) ∧
28(b).        exist u:U • u ∈ obs_Us(ω) ∧ is_Refi(u)
```

C.4 Preliminary Analysis slide 613

C.5 Simple Entities slide 614

C.5.1 slide 615

We (backward) refer to methodology Sect. 1.7.3 on page 19. It provides the methodological background for the present section.

C.6 Operations slide 616

We (backward) refer to methodology Sect. 1.7.4 on page 23. It provides the methodological background for the present section.

C.7 Events slide 617

We (backward) refer to methodology Sect. 1.7.5 on page 28. It provides the methodological background for the present section.

C.8 Behaviours slide 618

We (backward) refer to methodology Sect. 1.7.6 on page 29. It provides the methodological background for the present section.

slide 619
slide 620
slide 621
slide 622

D

Business Processes

slide 624

Chapter 6 (Pages 75–76) complements the present appendix. slide 625

slide 626

E

Domain Analysis and Concept Formation slide 627

Chapter 7 (Pages 79–81) complements the present appendix.

E.1 Simple Entities slide 628

E.1.1 Oil Industry Simple Entity Phenomena

29. Pipe, reservoir, drain pump, flow pump, join, fork, valve, depot, switch, refinery refractory tower, gas separator, gas purifier, (harbour berth to tanker) load arm, and tanker tank and sink (“casings”. “mechanisms”) — i.e., when viewed void of oil — all have the following in common: they are discrete, atomic and can be (correctly or erroneously) connected in simple ways to one another. We shall therefore suggest that we “lift” these phenomena into concepts of type (or sort) unit (U). slide 629
30. Oil (and natural gas) is liquid (gaseous) and can be contained in units.
31. Units, that is, Pipes, reservoirs, drain pumps, flow pumps, joins, forks, valves, depots, switches, refinery refractory towers, gas separators, gas purifiers, (harbour berth to tanker) load arms, tanker tanks and sinks when (partially) filled with (even no) oil (or gas) are dynamic, composite simple entities.
32. A unit, when not considering its possible oil (or natural gas) content, will be referred to as the unit casing. slide 630
33. Some unit casings can, when disregarding for example ambient temperature and humidity, be considered static and some are dynamic, that is their attribute values may be constants, respectively variables. Examples are:
    (a) A pipe casing is a static simple entity with, for example, these constant-valued attributes: length, diameter, geographical location, etc. slide 631
    (b) A reservoir is a dynamic simple entity with, for example, these constant-valued attributes: geographical location, maximum, original

capacity, number of attached drain pumps and their individual maximum pumping capacity. and with, for example, these variable-valued attributes: (per attached drain pump) their possible pumping states (being filled or being drained), their individual pumping state: full "speed", or less and remaining capacity slide 632

(c) A drain pump is a dynamic simple entity with, for example, these constant-valued attributes: geographical location, maximum pumping capacity, etc., and with, for example, these variable-valued attributes: possible pumping states (being filled or being drained), pumping state: full "speed", or less. slide 633

(d) A flow pump is a dynamic simple entity with, for example, these constant-valued attributes: geographical location, maximum pumping capacity, etc., and with, or example, these variable-valued attributes: possible pumping states, pumping state: full "speed", or less. slide 634

(e) A join slide 635

(f) A fork slide 636

(g) A valve casing is a dynamic simple entity with, for example, these constant-valued attributes: set of possible valve states, geographical location, etc., and with, for example, this variable-valued attribute: current valve state slide 637

(h) A depot, casing plus oil, is a dynamic simple entity with, for example, these constant-valued attributes: number of inlets and outlets, maximum containable volume, maximum filling volume per second (max filling rate), maximum draining volume per second (max draining rate), geographical location, etc.; and with, for example, these variable-valued attributes: which inlets, respectively which outlets are open or closed, current contained volume, whether being filled, or drained, or both, and then at which respective rates. The depot casing disregarding oil only has the first of the above variable-valued attributes; thus it is still a dynamic simple entity. slide 638

(i) A switch slide 639

(j) An oil refractory tower slide 640

(k) A gas separator slide 641

(l) A gas purifier slide 642

(m) A load arm slide 643

(n) A tank slide 644

(o) An end customer sink

slide 645

```
type
  3.  P
  3.  Pipe == Pi(pi:P)
  5.  Resr|DraP|FloP|Join|Fork|Valv|Depot|
                     Switch|Refr|Sepa|Puri|LdArm|Tank|Sink
  5.  Node == Re(re:Resr)|Dr(dr:DraP)|Fl(fl:FloP)|
                Jo(jo:Join)|Fo(fo:Fork)|Va(va:Valv)|De(de:Depot)|
                    Sw(sw:Switch)|Re(re:Refr)|Se(se:Sepa)|Pu(pu:Puri)|
                    LA(la:LdArm)|Ta(ta:Tank)|Si(si:Sink)
  29. U = Pipe | Node
  30. Oil, Gas
value
  31. obs_Oil: U → Oil, obs_Gas: U → Gas
```

slide 646

```
type
  33(a).  Attrs = (StaAtr $\underset{m}{\rightarrow}$ VAL) ∪ (DynAtr $\underset{m}{\rightarrow}$ VAL)
       StaAtr == ″lgth″ | ″diam″ | ″loca″ | ″max_dr″ | ″vlv_sts″ | ″max_vol″
                | ″max_fi″ | ″no_in″ | ″no_out″ | ...
       DynAtr == ″cur_vol″ | ″cur_vlv_sta″ | ″cur_pmp_sta″ |  ″in_sta″
                | ″out_sta″ | ...
value
  33(b).  obs_Attrs: U → Attrs
```

slide 647

```
axiom
  ∀ u:U •
     let (sas,das) = obs_Attrs(u) in
     (dom sas,dom das)=
        case u of
  33(a).     Pi(pi:P) →
                 ({″lgth″,″diam″,″loca″},{″cur_vol″}),
  33(b).     Re(re:Reserv) →
                 ({″loca″,″max_vol″,″no_out″,″no_in″},
                  {″cur_vol″,″out_sta″,″in_sta″}),
  33(c).     Dr(dr:DraP) →
                 ({″″,″″,″″},
                  {″″,″″,″″}),
  33(d).     Fl(fl:FloP) →
                 ({″″,″″,″″},
                  {″″,″″,″″}),
  33(e).     Jo(jo:Join) →
                 ({″″,″″,″″},
                  {″″,″″,″″}),
  33(f).     Fo(fo:Fork) →
                 ({″″,″″,″″},
                  {″″,″″,″″}),
  33(g).     Va(va:Valv) →
                 ({″″,″″,″″},
                  {″″,″″,″″}),
  33(h).     De(de:Depot) →
                 ({″″,″″,″″},
                  {″″,″″,″″}),
33(i).     Sw(si:Switch) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′}),
33(j).     Re(re:Refr) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′}),
33(k).     Se(se:Sepa) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′}),
33(l).     Pu(pu:Puri) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′}),
33(m).     LA(la:LdArm) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′}),
33(n).     Ta(ta:Tank) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′}),
33(o).     Si(si:Sink) →
                ({′′′′,′′′′,′′′′},
                 {′′′′,′′′′,′′′′})
    end end
```

E.1.2 Discrete and Continuous Simple Entities slide 648

We (backward) refer to methodology Sect. 1.7.1 (Pages: 16–18). It provides the methodological background for the present section.

34. Oil industry units: an oil pump (without oil) is a discrete simple entity; so are pipes (pipe segments), valves, depots, switches — when considered without oil; oil is a continuous simple entity: a barrel of oil can be divided into two fragments of the barrel in an infinity of ways, “down to the minutest level” of molecules.
35. Time, when viewed as a time axis, is a continuous simple entity, but when viewed as a specific time, a time point, it is a discrete entity.

E.2 Analysis of Domain Sketches slide 649

The English narrative items of Appendix C (Domain Acquisition) can be said to represent some of the domain acquisition (i.e., some of the description units) gathered from domain stakeholders and the formalisations can be said to represent initial parts of domain analysis. Of course many more such units should be gathered before completing the initial actions[1] of the domain acquisition stage. But it is useful, before settling on (choices of) formalisation, to alternate the domain acquisition stage with steps of the domain analysis and concept formation stage. So, although these stages, methodology-wise, are usually presented as ordered stages (acquisition before analysis), but usually pursued in an in interleaved fashion, we shall, "table-of-contents"-wise, present them "sequentially" !

E.2.1 Pipeline Systems

slide 650

In order to prepare for a major subsection (Sect. **??**) of this section we discuss relations between units, whether pipes or nodes.

Narrative

36. From a petroleum industry we can observe all units.
37. From a unit we can observe its unique identification.
38. From a unit we can observe the identification of the distinct (neighbour) units that are connected to the inputs and of the therefrom distinct (neighbour) units that are connected to the outputs.
39. For units such as pipes, drain and flow pumps, gas separators, gas purifiers and load arms there are exactly one connected predecessor unit and one connected successor unit.
40. For units such as tanks and sinks there are exactly one connected predecessor unit and no successor units.
41. And for units such as reservoirs, depots, switches and refineries there is a set of one or more predecessor units and a set of one or more successor units.
42. Whichever neighbour unit identifiers can be observed from a unit there is a unit in the petroleum industry of that identification.

slide 651

Formalisation

value
  36. obs_Us: $\Omega \to$ U-**set**
  37. obs_UI: U $\to$ UI
  38. obs_Neighbour_UIs: U $\to$ (UI-**set**$\times$UI-**set**)
axiom
  $\forall\ \omega:\Omega$ •
    $\forall$ u:U • u $\in$ obs_Us(ω)

[1] Initial actions of the domain acquisition stage are then to be followed up by supplementary actions of the domain acquisition stage as warranted, for example by the domain analysis and concept formation stage — as well as by the later business process sketching and domain modelling stages.

```
38. let (iuis,ouis) = obs_Neighbour_UIs(u) in iuis ∩ ouis = {} ∧
39. is_P(u)∨is_DraP(u)∨is_FloP(u)∨is_Sepa(u)∨is_Puri(u)∨is_LdArm(u)
              ⇒ card iuis = 1 = card ouis ∧
40. is_Tank(u)∨is_Sink(u)
              ⇒ card iuis = 1 ∧ card ouis = 0 ∧
41. is_Resr(u)∨is_Depot(u)∨is_Switch(u)∨isRefi(u)
              ⇒ card iuis ≥ 1 ∧ card ouis ≥ 1 ∧
42. ∀ ui:UI • ui ∈ iuis ∪ ouis
              ⇒ ∃ u′:U • u′ ∈ obs_Us(ω) ∧ ui=obs_UI(u′)
    end
```

E.2.2 Discussion of Emerging Concepts slide 652

And we shall generalise the notions of nodes and pipes into units of oil and natural gas systems. These units are (to be) composed, i.e., connected to one another in orderly ways.

The "point" (or 'spatial extent') where two units are joined is now to be referred to as a connection as well as a connector. This spatial extent, i.e., a connector (a connection), is a concept. We argue that it is not a phenomenon that has separate existence. If one end of a pipe can be connected to a valve (at a certain place "on" that valve), that pipe end is said, by us, here in our modelling, to have connector potential, so we treat it as a connector, but it really only becomes a connector once it is actually joined to the corresponding valve connector.

slide 653 And so on for all joinable pairs of units.

We shall make a kind of 'Gedanken Experiment'. Assume that two oil and natural gas system units are lying, separate from one another on the ground, say in the desert, near some oil fields, ready to be assembled, but not yet assembled. (We thus assume that the two units of a kinds that can indeed be assembled.) Can we refer to their connectors ? To answer this question let us — temporarily — assume that they have been assembled. Their assembly defines a connection and hence those two connectors of respective units that have been joined. slide 654 Those two connectors are now "the same", that is identical ! And we refer to them as one. Thus we shall take the position that before they were joined we could, and can, indeed, refer to the two connectors by the same, identical reference. The consequence of this "theory" is that there can at most be two units ("lying around in the desert", or being manufactured, shipped and made ready for assembly) sharing any give connector. There may be zero such units, or there may be just one such unit ! In the former case the connector is "purely" hypothetical. In the latter case the one-and-only-unit of a given connector appears to be one that is (was) never meant to be assembled ! slide 655

slide 656

F Domain Terminology slide 658

Chapter 8 (Pages 83–85) complements the present appendix.

We illustrate two steps of terminology development: A first rough sketching step and a more-or-less final step. The first rough sketching step reflects initial information gained in earliest steps of domain acquisition, while the "more-or-less" final step definitions reflect that considerable work has taken place between the earliest steps of domain acquisition and the end of the domain modelling, verification and validation stages.

F.1 Rough Sketch Characterisations slide 659

slide 660

1. **Berth:** rough

Berth is the term used in ports and harbours to define a specific location where a vessel may be berthed, usually for the purposes of loading and unloading.

See Item 35 on page 188. slide 661

Fig. F.1. Oil Tanker Berths

slide 662
slide 663

2. **Connector:** rough slide 664

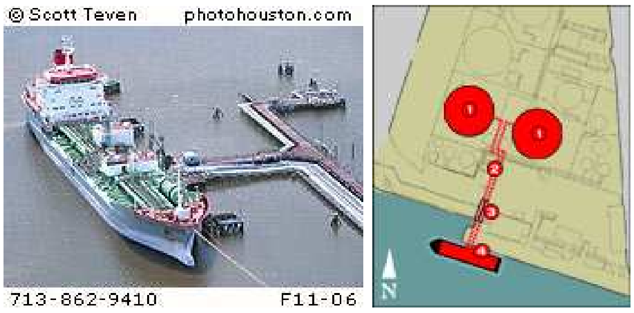

Fig. F.2. Gas Tanker Berth and Oil Tanker Berth to (Oil Storage) Depot

Fig. F.3. Oil Tanker Berth

A connector is a(n abstract) concept. Its physical manifestations are the areas where two unit[45]s of the oil industry are joined together. slide 665

3. **Connection Point:** rough

A connection point is a(n abstract) concept. Its physical manifestations are the areas where a unit of the oil industry may be joined together with another unit. For units see the enumeration at the end of Item 2 on the previous page. slide 666

4. **Crude Oil:** rough

Crude oil is the term for "unprocessed" oil, the stuff that comes out of the ground. It is also known as petroleum. Crude oil is a fossil fuel, meaning that it was made naturally from decaying plants and animals living in ancient seas millions of years ago – most places you can find crude oil were once sea beds. Crude oils vary in color, from clear to tar-black, and in viscosity, from water to almost solid.

Crude oils are such a useful starting point for so many different substances because they contain hydrocarbons. Hydrocarbons are molecules that contain hydrogen and carbon and come in various lengths and structures, from straight chains to branching chains to rings. slide 667

5. **Depot (Oil or Gas Storage):**rough

An oil depot (sometimes called a tank farm, installation or oil terminal) is an industrial facility for the storage of oil and/or petrochemical products and from which these products are usually transported to end users or further storage facilities. An oil depot typically has tankage, either above ground or underground, and gantries for the discharge of products into road tankers or other vehicles (such as barges) or pipelines.

Oil depots are usually situated close to oil refineries or in locations where marine tankers containing products can discharge their cargo. Gas is usually stored in underground depots. Some depots are attached to pipelines from which they draw their supplies and depots can also be fed by rail, by barge and by truck (sometimes known as "bridging") slide 668

Fig. F.4. Oil Depots

slide 669

Fig. F.5. Underground Gas Depot and Switch

slide 670
slide 671

6. **Drain Pump:** rough

Fig. F.6. Oil Depots

See pump[36]. By a drain pump we understand a pump which drains a reservoir of gas or oil, be it under land or under sea.

Hence oil rigs, pump jacks, gas lifts and submersible pumps are drain pumps. See Figs. F.7 to F.8. slide 672

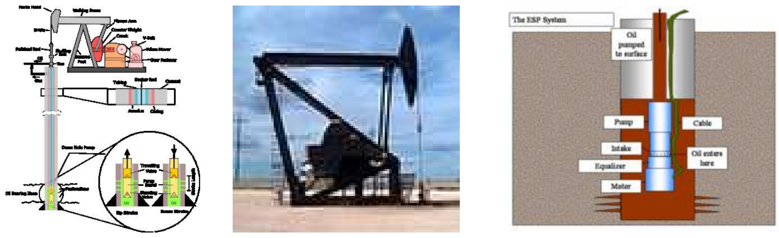

Fig. F.7. Pump Jack, Drain Pump and Submersible Pump

slide 673

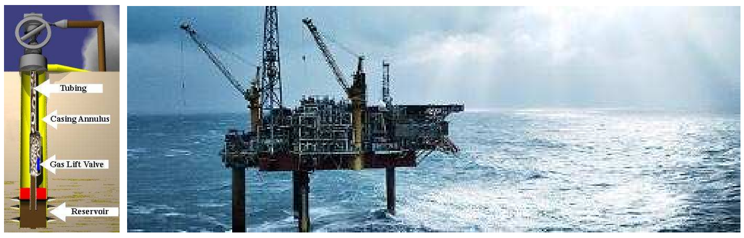

Fig. F.8. Gas Lift and Oil Rig

slide 674

7. **End Customer:** rough

By an end customer is meant a sink where oil, gas or products refined from these may be finally be delivered (i.e., “leaves” the oil industry). slide 675

8. **Fork:** rough

By a fork we understand the splitting of one pipe[31] into two (or more) pipe[31]s. slide 676

9. **Field:** rough

By an oil[26] and/or a gas[15] field we understand an oil[26] and/or a gas[15] reservoir[39] with one or more drain pump[6]s. slide 677

10. **Fill Pump:** rough

See pump[36]. Same as a flow pump[12] only used specifically for pumping oil or gas into a depot, a tank or a sink.
See Fig. F.9. slide 678

11. **Flow:** rough

In graph theory, a flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. Often in Operations Research, a directed graph is called a network, the vertices are called nodes[1] and the edges are called arcs (here pipe[31]s). A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, except when it is a source (i.e., reservoir[39], depot[5] or tank[42]), which has more outgoing flow, or sink (i.e., depot[5], tank[42] or end customer[7]), which has more incoming flow. A network can be used to model traffic in a road system, fluids in pipe[31]s, currents in an electrical circuit, or anything similar in which something travels through a network of nodes. slide 679

12. **Flow Pump:** rough

See pump[36]. By a flow pump we understand a See Fig. F.9. slide 680

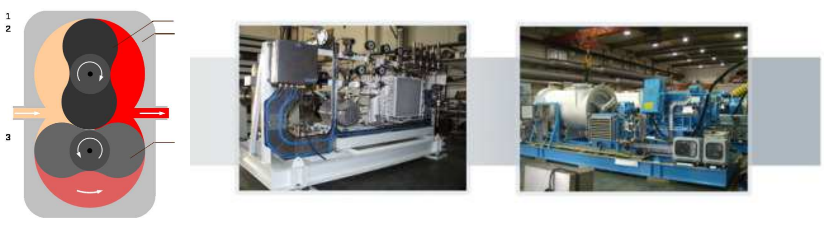

Fig. F.9. Rotary Piston Oil Pump and Two Gas Compressors

slide 681

13. **Flow Rate, Volumetric:** rough

[1] In our main example nodes are the non-pipe units such as reservoirs, drain pumps, flow pumps, valves, switches, depots, refineries, tanks, etc.

The volumetric flow rate in fluid dynamics and hydrometry, (also known as volume flow rate or rate of fluid flow) is the volume of fluid which passes through a given surface per unit time (for example cubic meters per second, $\frac{m^3}{s}$, in SI units. slide 682

14. **Flux:** rough

In the study of transport phenomena (heat transfer, mass transfer and fluid dynamics), flux is defined as the amount that flows through a unit area per unit time. Volumetric flow rate should not be confused with volumetric flux, as defined by Darcy's law with units of $\frac{m^3}{(m^2 \cdot s)}$, that is, $\frac{m}{s}$. The integration of a flux over an area gives the volumetric flow rate. slide 683

15. **Gas:** rough

Natural gas is a gaseous fossil fuel consisting primarily of methane but including significant quantities of ethane, propane, butane, and pentane as well as carbon dioxide, nitrogen, helium and hydrogen sulfide.

Fossil natural gas is found in oil fields (associated) either dissolved or isolated in natural gas fields (non-associated), and in coal beds (as coal bed methane). slide 684

16. **Gasoline:**

Gasoline or petrol is a petroleum-derived liquid mixture, primarily used as fuel in internal combustion engines.

It consists mostly of aliphatic hydrocarbons, enhanced with iso-octane or the aromatic hydrocarbons toluene and benzene to increase its octane rating. Small quantities of various additives are common, for purposes such as tuning performance or reducing emissions. Some mixtures also contain significant quantities of ethanol as a partial alternative fuel.

Gasoline is the American term where petrol is the British term. slide 685

17. **Gas Processor:** rough

A gas turntable is (like an oil refinery), a place where natural gas is processed (cleansed, purified, etc.). Water vapor is separated from the gas in drying columns by a glycol drying process. Incoming gas goes through a purification process to remove solid and liquid impurities. This is performed by so-called filter separators. Afterwards the gas is subject to quality controls. Gas pipelines are regularly inspected, maintained, and cleaned. Entry points known as "pig traps" are installed at the beginning and end of each section of pipeline. Here, cleaning and "smart" pigs are fed into the pipeline. These are used to clean and inspect pipelines. slide 686

slide 687

slide 688

18. **Gas Tanker:** rough

A usually ocean-going vessel with one or more natural gas[24] (or liquified petroleum) tank[42]s. slide 689

slide 690

slide 691

Fig. F.10. Gas Cleansing: Glycon Drying Station and Gas Pig

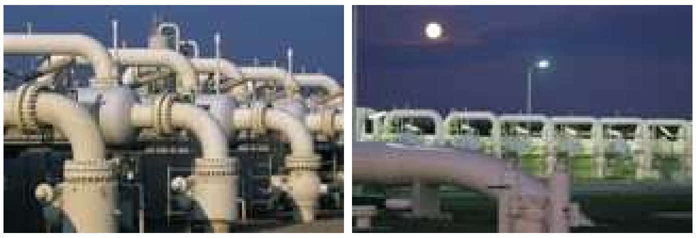

Fig. F.11. Gas Separators

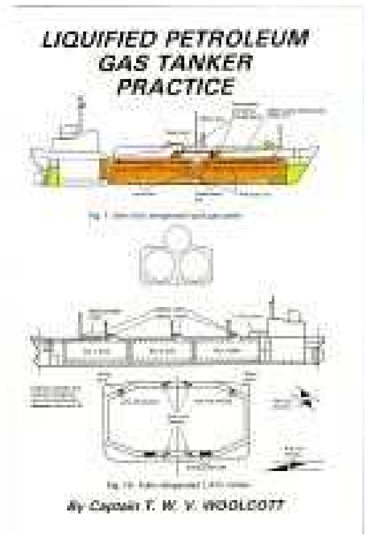

Fig. F.12. Gas Tanker

19. **Graph:** rough

Graphs are finite connected sets of unit[45]s, in particular such that pipe[31]s link all node[25]s. slide 692

Fig. F.13. Gas Tanker

20. **Harbour:** rough

A location at the interface between land and sea where (usually ocean-going) vessels can sail sea lanes to and from other harbours. A harbour contains one or more depot[5]s and one or more berth[1]s. slide 693

21. **Join:** rough

By a join we understand the merge of usually two pipe[31]s into one pipe[31]. See leftmost component of Fig. F.19 on page 187. slide 694

22. **Land Switch:** rough

slide 695 A land switch is a switch[41] built on land.

Fig. F.14. Oil and Gas Switches

slide 696
slide 697

23. **Loading Arm:** rough

A loading arm is a means of moving, in our case, liquid (oil[26]) or gaseous material (gas[15]) either from a land switch[22] to a tanker switch[44] or from a tanker switch[44] to a land switch[22]. slide 698

slide 699

24. **Natural Gas:** rough

slide 700 See Item F.2 on page 193 gas[15].

Fig. F.15. Depot Switches

Fig. F.16. FMS Loading Arms

25. **Node:** rough

Nodes are the unit[45]s that are pre-, in- or suffixing pipe[31]s to form pipeline[32]s, yes, indeed, petroleum industry[30] (sub)systems. slide 701

26. **Oil:** rough

See Item 4 on page 178, crude oil. slide 702

27. **Oil Tanker:** rough

A usually ocean-going vessel with one or more oil tank[42]s.

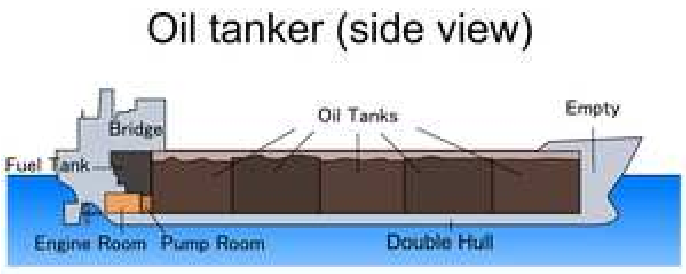

Fig. F.17. Oil Tanker

slide 703

28. **Petrol:** .

slide 704 Same as gasoline[16].

29. **Petroleum:** .

slide 705 Same as crude oil[4].

30. **Petroleum Industry:**

A petroleum industry consist of subsystems of one or more of each of gas[15] and oil[26] field[9]s, pipeline[32] systems, refineries[37] and harbour[20], berth[1]s and tanker[43]s. slide 706

31. **Pipe:** rough

A pipe is a usually circular tube of some length. Oil and gas pipes are usually of some considerable diameter, say typically 48" with typical lengths of from $20m$ to $50m$. slide 707

32. **Pipeline:** rough

A pipeline is basically a serial (ordered) composition of pipe[31]s. In addition, inserted in-between pipes, at suitable (distance) intervals, are valve[46]s and pump[36]s. A more detailed sketch is given below. slide 708

Fig. F.18. Pipelines

slide 709

Pipeline Components See Fig. F.19 on the next page.

Pipeline networks are composed of several pieces of equipment that operate together to move products from location to location. The main elements of a pipeline system are:

- Initial Injection Station - Known also as Supply or Inlet station, is the beginning of the system, where the product is injected into the line. Storage facilities, pumps or compressors are usually located at these locations.

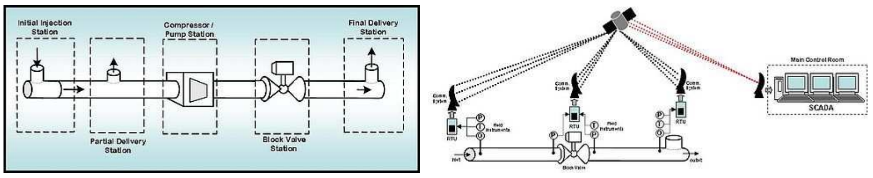

Fig. F.19. Pipeline Composition

- Compressor/Pump Stations - Pumps for liquid pipelines and Compressors for gas pipelines, are located along the line to move the product through the pipeline. The location of these stations is defined by the topography of the terrain, the type of product being transported, or operational conditions of the network. slide 710
- slide 711 Partial Delivery Station - Known also as Intermediate Stations, these facilities allow the pipeline operator to deliver part of the product being transported.
- Block Valve Station - These are the first line of protection for pipelines. With these valves the operator can isolate any segment of the line for maintenance work or isolate a rupture or leak. Block valve stations are usually located every 20 to 30 miles (48 km), depending on the type of pipeline. Even though it is not a design rule, it is a very usual practice in liquid pipelines. The location of these stations depends exclusively on the nature of the product being transported, the trajectory of the pipeline and/or the operational conditions of the line. slide 712
- Regulator Station - This is a special type of valve station, where the operator can release some of the pressure from the line. Regulators are usually located at the downhill side of a peak.
- Final Delivery Station - Known also as Outlet stations or Terminals, this is where the product will be distributed to the consumer. It could be a tank terminal for liquid pipelines or a connection to a distribution network for gas pipelines.

slide 713

33. **Pipeline System:** rough

A pipeline system consists of one or more pipe[31]s that infix (i.e., connect) zero, one or more pump[36]s (drain pump[6]s and flow pump[12]s), join[21]s, fork[8]s and valve[46]s. slide 714
Figure F.20 on the following page indicates the route of the proposed Nabucco 56″ natural gas pipeline system: http://en.wikipedia.org/wiki/Nabucco_Pipeline: from the gas fields of Adzerbajian, Georgia and Iran — and, eventually, Central Asia — to Europe. slide 715

34. **Pipeline Transport:** rough

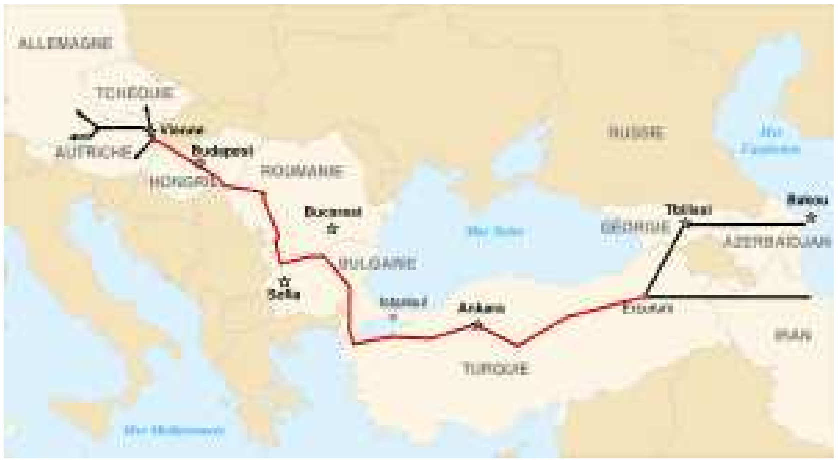

Fig. F.20. The Nabucco 3.300 km Pipeline System (by 2020)

Pipeline transport is the transportation of goods through a pipe. Most commonly, liquid and gases are sent, but pneumatic tubes that transport solid capsules using compressed air have also been used. As for gases and liquids, any chemically stable substance can be sent through a pipeline. Therefore sewage, slurry, water, or even beer pipelines exist; but arguably the most important are those transporting oil and natural gas. The idea was first brought up by Dmitri Mendeleev[2] in 1863. He suggested to use a pipe for transporting Petroleum. slide 716

35. **Product Berth:** rough

A product berth[1] is used to handle oil and gas related products, usually in liquid form. Vessels are loaded via loading arm[23]s containing the pipe lines. Storage (depot[5]s) facilities for the products are usually some distance away from the berth and connected by several pipes, a land switch[22], to ensure fast loading. slide 717

36. **Pump:** rough

A pump is a device used to move fluids, such as gases, liquids or slurries. A pump displaces a volume by physical or mechanical action. One common misconception about pumps is the thought that they create pressure. Pumps alone do not create pressure they only displace fluid causing a flow. Adding resistance to flow causes pressure.

Cf. drain pump[6] (Page 179), fill pump[10] (Page 181) and flow pump[12] (Page 181). slide 718

[2] Credited with having proposed the Period Table.

37. **Refinery:** rough

An oil refinery is an industrial process plant where crude oil[4] is processed and refined into more useful petroleum[29] products, such as gasoline[16], diesel fuel, kerosene, and liquefied petroleum.

Refineries are typically large sprawling industrial complexes with extensive piping running throughout, carrying streams of fluids between large chemical processing units.

Crude oil distillation is separated into fractions by fractional distillation (in a refractory[38] tower). The fractions at the top of the fractionating column (or refractory[38] tower) have lower boiling points than the fractions at the bottom. The heavy bottom fractions are often cracked into lighter, more hundreds of different hydrocarbon molecules in crude oil are separated in a refinery into components that can be used as fuels, lubricants, and as feed stock in petrochemical processes that manufacture such products as plastics, detergents, solvents, elastomers and fibers such as nylon and polyesters. slide 719
These different hydrocarbons have different boiling points, which means they can be separated by distillation. Since the lighter liquid products are in great demand for use in internal combustion engines, a modern refinery will convert heavy hydrocarbons and lighter gaseous elements into these higher value products.

Typical refinery products are liquid petroleum gas (LPG), gasoline (also known as petrol) naphtha, kerosene and related jet aircraft fuels diesel fuel, fuel oils, lubricating oils, paraffin wax, asphalt and tar, petroleum coke, etc.

See Fig. F.21 to Fig. F.25 on the following page. slide 720

Fig. F.21. Oil Refineries

slide 721
slide 722
slide 723
slide 724
slide 725

38. **Refractory:** rough

By a refractory (tower) is meant a core part of a refinery[37]. A refinery[37] may contain several refractories. A refractory usually has one pipe[31] leading into it (from a land switch[22]) and several pipe[31]s leading out from it where each such pipe[31], via another land switch[22], leads the refined product to appropriate (storage) depot[5]s. We refer to the sketch given of refineries, Item 37. slide 726

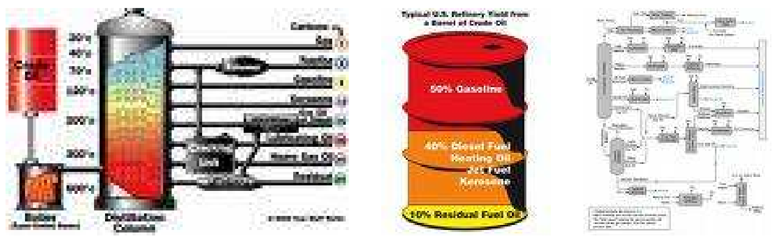
Boiler
Distillation Column
Typical U.S. Refinery Yield from a Barrel of Crude Oil
50% Gasoline
40% Diesel Fuel Heating Oil Jet Fuel Kerosene
10% Residual Fuel Oil

Fig. F.22. Oil Refinery Processes

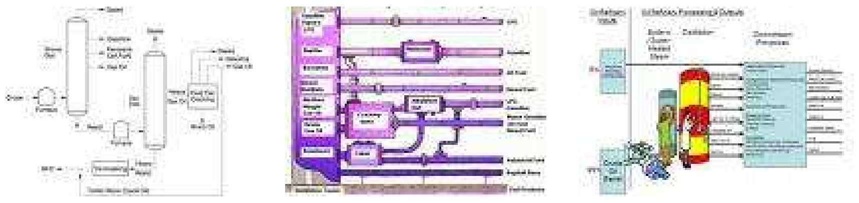

Fig. F.23. Oil Refinery Processes

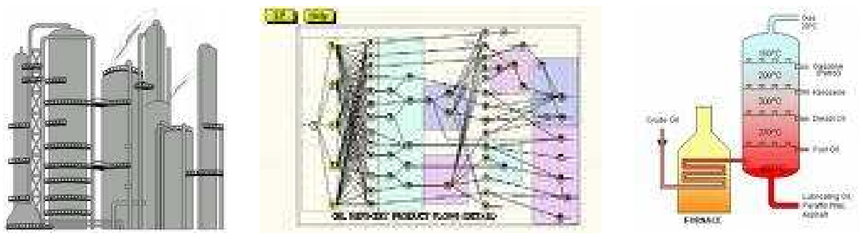

Fig. F.24. Oil Refinery Processes

Fig. F.25. Oil Refineries

39. **Reservoir:** rough

By a reservoir is meant a usually very large volume of oil[26] or gas[15], either under the surface of land. From a reservoir one can drain oil[26] and/or gas[15] by means of drain pump[6]s. slide 727

40. **Sea Lane:** rough

By a sea lane is understand a course of sea voyage for a gas tanker[18] or an oil tanker[27], from one harbour[20] to another, that is, from one berth[1] and loading arm[23] of origin to another berth[1] and loading arm[23] of destination. slide 728

41. **Switch:** rough

A switch is an abstract concept. Its concrete manifestations are systems of pipe[31]s, valve[46]s and flow pump[12]s all contained with a close "neighbourhood" of either (α) depot[5]s, refractory[38] towers, and berth[1] loading arm[23]s or (β) berth[1] loading arm[23]s, and tanker[43] tank[42]s
See land switch, Item 22 on page 184, and tanker switch, Item 44. slide 729

42. **Tank:** rough

By a tank is meant a compartment on a oil tanker[27] or a gas tanker[18] capable of holding a large volume of oil[26], respectively gas[15]. slide 730

43. **Tanker:** rough

See gas tanker[18], Item 18 (Page 193), and oil tanker[27], Item 27 (Page 193).
slide 731

44. **Tanker Switch:** rough

A tanker switch is a switch[41] built on a tanker[43].
See text right after (β) in Item 41: switch[41]. slide 732

45. **Unit:** rough

Units are the components from which the petroleum industry is put together. These units are either reservoir[39]s, drain pump[6]s, pipe[31]s, join[21]s, fork[8]s, flow pump[12]s, valve[46]s, depot[5]s, land and tanker switch[41]es, harbour[20] berth[1] loading arm[23]s, tanker[43] tank[42]s, refinery[37] refractories[38], end customer[7]s, etc. slide 733

46. **Valve:** rough

By a valve is understood a (node) component which is usually inserted between pipe[31]s with the purpose of controlling the flow of oil[26] (or gas[15]) from incoming pipe[31]s to outgoing pipe[31]s.
Thus a valve can be a $1 \rightarrow 1$ valve, one input, one output, or, in general, a $m \rightarrow n$ valve, $m \geq 1, n \geq 1$.
Joins[21] and fork[8]s are special cases of valves: these "spacial case" valves are always open. slide 734
slide 735

Fig. F.26. Pipeline Ball Valves for Oil and Gas Transport

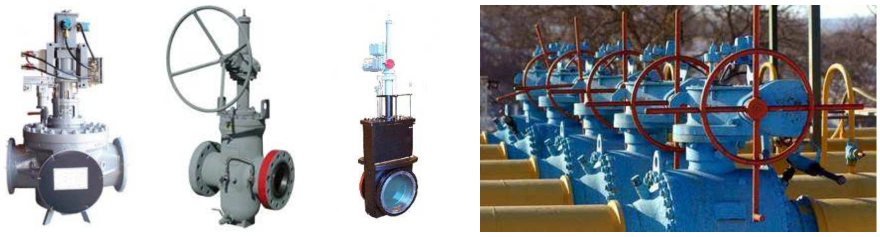

Fig. F.27. Multiway and Expanding Gate + Russia-Ukraine Gas Valves

F.2 "More-or-Less" Final Definitions slide 736

slide 737

1. **Berth:** final

   See Item 35, product berth (Page 194). slide 738

2. **Connector:** final slide 739

3. **Connection Point:** final slide 740

4. **Crude Oil:** final slide 741

5. **Depot (Oil or Gas Storage):**final slide 742

6. **Drain Pump:** final slide 743

7. **End Customer:** final slide 744

8. **Fork:** final slide 745

9. **Field:** final slide 746

10. **Fill Pump:** final

slide 747

11. **Flow:** final

slide 748

12. **Flow Pump:** final

slide 749

13. **Flow Rate, Volumetric:** final

slide 750

14. **Flux:** final

slide 751

15. **Gas:** final

slide 752

16. **Gasoline:**

slide 753

17. **Gas Processor:** final

slide 754

slide 755

18. **Gas Tanker:** final

slide 756

19. **Graph:** final

slide 757

20. **Harbour:** final

slide 758

21. **Join:** final

slide 759

22. **Land Switch:** final

slide 760

23. **Loading Arm:** final

slide 761

24. **Natural Gas:** final

    See Item 15 gas (Page 193).

slide 762

25. **Node:** final

slide 763

26. **Oil:** final

    See Item 4 crude oil (Page 192).

slide 764

27. **Oil Tanker:** final

slide 765

28. **Petrol:** See Item 16, Page 193.

slide 766

29. **Petroleum:** See Item 4, Page 192.

slide 767

30. **Petroleum Industry:**

slide 768

31. **Pipe:** final

slide 769

32. **Pipeline:** final

slide 770

33. **Pipeline System:** final

slide 771

34. **Pipeline Transport:** final

slide 772

35. **Product Berth:** final

slide 773

36. **Pump:** final

slide 774

37. **Refinery:** final

slide 775

38. **Refractory:** final

slide 776

39. **Reservoir:** final

slide 777

40. **Sea Lane:** final

slide 778

41. **Switch:** final

slide 779

42. **Tank:** final

slide 780

43. **Tanker:** final

slide 781

44. **Tanker Switch:** final

slide 782

45. **Unit:** final

slide 783

46. **Valve:** final

slide 784

G

Intrinsics

slide 786

Chapter 10 (Pages 93–94) complements the present appendix.

G.1 Domain Entities

slide 787

G.1.1 Composite Entities

Ω: The Overall System

Narrative

43. The overall petroleum and natural gas industry, *Ω*, consists of
    (a) One or more crude oil and/or natural gas reservoirs,
    (b) one or more crude oil, oil product (gasoline etc.) and/or natural gas pipeline systems,
    (c) one or more refineries or gas processors,
    (d) two or more crude oil, oil product and/or natural gas shipment harbours,
    (e) one or more oil product truck distribution nets,
    (f) one or more crude oil, oil product and/or natural gas tankers, and
    (g) one or more oil product and/or natural gas trucks.
44. We shall not consider truck distribution nets (cf. Item 43(e)) and trucks (cf. Item 43(g)).

slide 788

Formalisation

```
type
   43. Ω,
   43(a). Reservoir
   43(b). Pipeline
   43(c). Refinery, GasProcessor
```

```
    43(d). Harbour
    43(f). Tanker
value
    43(a). obs_Reservoirs: Ω → Reservoir-set
    43(b). obs_Pipelines: Ω → Pipeline-set
    43(c). obs_Refineries: Ω → Refinery-set
    43(d). obs_Harbours: Ω → Harbour -set
    43(f). obs_Tankers: Ω → Tanker-set
axiom
            ∀ ω:Ω •
    43(a). card obs_Reservoirs(ω) ≥ 1
    43(b). card obs_Pipelines(ω) ≥ 1
    43(c). card obs_Refineries(ω) ≥ 1
    43(d). card obs_Harbours(ω) ≥ 2
    43(f). card obs_Tankers(ω) ≥ 1
```

Reservoirs

slide 789

Narrative

45. We shall consider reservoirs to be atomic units, see Item 54 on page 201.
46. Pipelines consists of atomic units such as
    (a) one or more drain pumps,
    (b) one or more pipes,
    (c) zero, one or more flow pumps,
    (d) zero, one or more joins,
    (e) zero, one or more forks and
    (f) zero, one or more valves.
    All these will be further treated below.

slide 790

Formalisation

```
type
        DraP, Pipe, FloP, Join, Fork, Valv,
        DrP(b:DraP), mkP(b:Pipe), FlP(b:FloP), Jn(b:Join), Fk(b:Fork), Val(b:Valv)
value
    46(a). obs_DrainPumps: Pipeline → DrP(b:DraP)-set
    46(b). obs_Pipes: Pipeline → mkP(b:Pipe)-set
    46(c). obs_FlowPumps: Pipeline → FlP(b:FloP)-set
    46(d). obs_Joins: Pipeline → Jn(b:Join)-set
    46(e). obs_Forks: Pipeline → Fk(b:Fork)-set
    46(f). obs_Valves: Pipeline → Val(b:Valv)-set
axiom
    ∀ pl:Pipeline • card obs_DrainPumps(pl)≥1 ∧ card obs_pipes(pl)≥1
```

Refineries slide 791

Narrative

47. Refineries consists of
    (a) one or more land-based refinery input depots,
    (b) one or more refinery input depot to refractory tower switches,
    (c) one or more refractory towers,
    (d) one or more refractory tower-to-refinery output depot land-based switches, and
    (e) one or more refinery output depots.
48. Refinery switches are said to be land-based.

slide 792

Formalisation

```
type
      Depot, Switch, OilRef
      Refinery = Depot-set×Switch-set×OilRef-set×Switch-set×Depot-set
value
   47(a),47(e).   is_input_Depot, is_output_Depot: Depot → Bool
   48.            is_Land_Switch, is_Tanker_Switch: Switch → Bool
axiom
   48.     ∀ sw:Switch • is_Tanker_Switch(sw) ≡ ∼ is_Land_Switch(sw) ∧
           ∀ d:Depot • is_input_Depot(d)≡∼is_output_Depot(d) ∧
           ∀ (ids,drss,rs,rdss,ods):Refinery •
   47(a)-47(b).     card ids ≥ 1 ∧ card drss ≥ 1 ∧
   47(c)-47(e).     card rs ≥ 1 ∧ card rdss ≥ 1 ∧ card ods ≥ 1 ∧
   47(a).           ∀ d:D•d ∈ ids ≡ is_input_Depot(d) ∧
   47(e).           ∀ d:D•d ∈ ods ≡ is_output_Depot(d)
   48.     ∀ s:Switch • s ∈ ss ≡ is_Tanker_Switch(sw)
```

Gas Processors slide 793

Narrative

49. Gas Processors are very much similarly cmposed as are oil refineries — but will not be further treated in this example.

Harbours slide 794

Narrative

50. Harbours consists of
    (a) one or more harbour input depots,

   (b) one or more land-based harbour input depot-to-loading arm switches,
   (c) one or more loading arms,
   (d) one or more land-based loading arm-to-harbour output depot switches, and
   (e) one or more harbour output depots.

51. Harbour switches are said to be land-based.

slide 795

Formalisation

```
type
        Depot, Switch, LdArm
        Harbour = Depot-set×Switch-set×LdArm-set×Switch-set×Depot-set
value
51.  is_Land_Switch, is_Tanker_Switch: Switch → Book
axiom
51.  ∀ sw:Switch • is_Land_Switch(sw) ≡ ∼ is_Tanker_Switch(sw) ∧
        ∀ (ids,oss,las,rss,ids):Harbour •
    50(a)-50(b).      card ids ≥ 1 ∧ card oss ≥ 1 ∧
    50(c)-50(e).      card las ≥ 1 ∧ card rss ≥ 1 ∧ card ids ≥ 1 ∧
    50(a).             ∀ d:D•d ∈ ids ≡ is_input_Depot(d) ∧
    50(e).             ∀ d:D•d ∈ ods ≡ is_output_Depot(d) ∧
51.  ∀ s:Switch • s ∈ oss ∪ rss ≡ is_Land_Switch(sw)
```

Tankers

slide 796

Narrative

52. Tankers consists of
   (a) one or more tanks and
   (b) one or more tanker-based switches.

53. Switches of tankers are said to be tanker-based.

slide 797

Formalisation

```
type
52(a).  Tank, Switch
52(b).  Tanker = Tank-set × Switch-set
value
53.  is_Tanker_Switch, is_Land_Switch: Switch → Bool
axiom
53.  ∀ sw:Switch • is_Tanker_Switch(sw) ≡ ∼ is_Land_Switch(sw) ∧
        ∀ (ts,ss):Tanker •
52(a).    card ts ≥ 1 ∧
52(b).    card ss ≥ 1 ∧
53.  ∀ s:Switch • s ∈ ss ≡ is_Tanker_Switch(sw)
```

G.1.2 Atomic Entities

slide 798

Narrative

54. The atomic units of a petroleum industry include nodes: reservoirs, drain pumps, flow pumps, joins, forks, valves, switches, depots, refractories, gas processors, berth loading arms, tanker tanks and sinks (i.e., end customers).
55. The atomic units of a petroleum industry further include pipes.

Formalisation

```
type
  54.  Resr, DraP, FloP, Join, Fork, Valv, Switch,
        Depot, OilRef, GasPro, LdArm, Tank, Sink
  55.  Pipe
```

slide 799

Narrative

56. Units are either nodes or pipes.

Formalisation

```
type
  56.  U = N | P
```

A Technicality

slide 800

Narrative

57. To ensure that the units of the different kinds (Resr, DraP, FloP, Join, Fork, Valv, Switch, Depot, Refr, Sepa, Puri, LdArm, Tank, Sink and Pipe) are uniquely distinguishable we model them as abstract record structures.

Formalisation

```
type
  57.  P == mkP(b:Pipe)
  57.  N == Res(b:Resr) | DrP(b:DraP) | FlP(b:FloP) | Jn(b:Join)
                 | Fk(b:Fork) | Val(b:Valv) | Swi(b:Switch) | OiR(b:OilRef)
                 | GaP(b:GasPro) | LdA(b:LdArm) | Snk(b:Sink)
```

The A == mkB(s:B) — mkC(s′:C) — mkD(s″:D) type definition introduces three record structures mkB(b), mkC(c), and mkD(d) where B, C and D are type names and where b:B, c:C and d:D, and where s, s′ and s″ are selectors.

G.1.3 The Unit Connector Concept

slide 801

Unit Connectors

Narrative

58. Each Unit has Connectors[1].
59. From each units we can observe the possibly empty or singleton set of connectors leading into that unit and the the possibly empty or singleton set of connectors leading out from that unit.
60. All input connectors (obs_iCs(u)) are distinct, all output connectors (obs_oCs(u)) are distinct, and the two sets of input and output connectors have no connectors in common.

slide 802

Formalisation

```
type
   58.  C
value
   60.  obs_pCs: U → (C-set×C-set)
   60.  xtr_iCs: U → C-set, xtr_oCs: U → C-set
axiom
   60.  ∀ u:U • (xtr_iCs(u),xtr_oCs(u)) ≡ obs_pCs(u) ∧
               let (cs,cs′)=obs_Cs(u) in cs ∩ cs′={} end
```

Two Auxiliary Predicates

slide 803

Narrative

Give a unit, u, in a context of units, us, we wish to determine whether that unit, u, is correctly connected to (thus) adjacent units, u', u'', in us. Correct connections is determined by a set of predecessor predicates, *pps* and successor predicates, *sps*. wf_io_Cs applies to units with both predecessor and successor units. wf_i_Cs applies to units with only predecessor units.

slide 804

Formalisation

```
value
   wf_io_Cs: U×U-set → ((U→Bool)-set×(U→Bool)-set) → Bool
   wf_io_Cs(u,us)(pps,sps) ≡
      let (ics,ocs) = obs_pCs(u) in
      ∀ c:C • c ∈ ics ⇒
         ∃ u′:U • u′ ∈ us ∧ u≠u′ ∧
```

[1] — and, pragmatically, connectors are what “binds” units together

```
            ∃ pp:(U→**Bool**)•pp ∈ pps ∧ pp(u′) ⇒ c ∈ xtr_oCs(u′) ∧
      ∀ c:C • c ∈ ocs ⇒
         ∃ u″:U • u″ ∈ us ∧ u≠u″ ∧
            ∃ sp:(U→**Bool**)•sp ∈ sps ∧ sp(u″) ⇒ c ∈ xtr_iCs(u″)
      end
   wf_i_Cs: U×U-set → (U→Bool) → Bool
      let (ics,ocs) = obs_pCs(u) in
   wf_i_Cs(u,us)(pps)(ics) ≡
      ∀ c:C • c ∈ ics ⇒
         ∃ u′:U • u′ ∈ us ∧ u≠u′ ∧
            ∃ pp:(U→Bool)•pp ∈ pps ∧ pp(u′) ⇒ c ∈ xtr_oCs(u′)
      end
```

Wellformedness of Connections slide 805

61. **Reservoirs** have a pair of disjoint sets of zero, one or more connectors.
    Constraints: Reservoir output connectors are (to be) bound to drain pump inputs, one-by-one) and input connectors (for refilling the reservoir), are (to be) bound flow pump outputs, one-by-one.

(61) $\forall\ \omega{:}\Omega \bullet \forall$ u:U • u ∈ obs_Us(ω) ⇒ is_Reserv(u) ⇒
    wf_io_Cs(u,obs_Us(ω)({is_FloPump},{is_DraPump}))

slide 806

62. **Drain Pumps** have a pair of distinct connectors: one input and one output connector.
    Constraints: Drain Pump input connector (to be) bound to a Reservoir output connector and Drain Pump output connector (to be) bound either to a Flow Pump input connector or to a Pipe input connector or to a Switch input connector.

(62) $\forall\ \omega{:}\Omega \bullet \forall$ u:U • u ∈ obs_Us(ω) ⇒ is_DraPump(u) ⇒
    let (ics,ocs)=obs_pCs(u) **in card** ics = 1 = **card** ocs ∧
    wf_io_Cs(u,(ics,ocs))({is_Reserv},{is_FloPump,is_Pipe}) **end**

slide 807

63. **Pipes** have a pair of distinct connectors.
    Pragmatics & Constraints: The pair of distinct connectors thus designates a direction (that is, we assume pipes, like all other units, except [perhaps] swithches, to be directional). Pipe connectors (can) bind two Pipe units together or a Pipe unit (output/input) with a Depot (input/output), or a Pipe unit input Drain Pump output, or a Pipe unit (output/input) with a Valve or Flow Pump (input/output).

(63) $\forall\ \omega:\Omega \bullet \forall$ u:U $\bullet$ u $\in$ obs_Us(ω) $\Rightarrow$ is_Pipe(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in card** ics = 1 = **card** ocs $\wedge$
    wf_io_Cs(u,obs_Us(ω))({is_DraPump,is_Pipe},{is_Pipe,is_FloPump}) **end**

slide 808

64. **Flow Pumps** have a pair of distinct connectors: one input and one output connector.
    Constraints: Flow Pump input connectors are (to be) bound to Pipe or Valve or Join or Fork output connectors and Flow Pump output connectors are (to be) bound to Pipe or Valve or Join or Fork or Sink input connectors.

(64) $\forall\ \omega:\Omega \bullet \forall$ u:U $\bullet$ u $\in$ obs_Us(ω) $\Rightarrow$ is_FloPump(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in card** ics = 1 = **card** ocs $\wedge$
    wf_io_Cs(u,(ics,ocs))({is_Pipe,is_Valve,is_Join,is_Fork},
                {is_Pipe,is_Valve,is_Join,is_Fork,is_Sink}) **end**

slide 809

65. **Joins** have a pair of disjoint sets of one or more connectors, respectively the input and the output connectors.
    Constraints: Join input connectors are (to be) bound to Pipe output connectors, and the Join output connector is (to be) bound to a Pipe input connector.

(65) $\forall\ \omega:\Omega \bullet \forall$ u:U $\bullet$ u $\in$ obs_Us(ω) $\Rightarrow$ is_Join(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in card** ics $\geq$ 2 = **card** ocs = 1 $\wedge$
    wf_io_Cs(u,(ics,ocs))({is_Pipe},{is_Pipe}) **end**

slide 810

66. **Forks** have a pair of disjoint sets of one input connector two or more output connectors.
    Constraints: The fork input connector is (to be) bound to a Pipe output connector, and the Fork output connectors are (to be) bound to Pipe input connectors.

(66) $\forall\ \omega:\Omega \bullet \forall$ u:U $\bullet$ u $\in$ obs_Us(ω) $\Rightarrow$ is_Fork(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in card** ics = 1 = **card** ocs $\geq$ 2 $\wedge$
    wf_io_Cs(u,(ics,ocs))({is_Pipe},{is_Pipe}) **end**

slide 811

67. **Valves** have a pair of disjoint sets of one or more connectors, respectively the input and the output connectors.
    Constraints: Valve input connectors are (to be) bound to Pipe or Flow Pump output connectors, and Valve output connectors are (to be) bound to Pipe or Flow Pump input connectors.

(67) $\forall\ \omega:\Omega \bullet \forall$ u:U $\bullet$ u $\in$ obs_Us(ω) $\Rightarrow$ is_Pipe(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in card** ics $\geq$ 1 = **card** ocs $\geq$ 1 $\wedge$
    wf_io_Cs(u,(ics,ocs))({is_Pipe,is_FloPump},{is_Pipe,is_FloPump}) **end**

slide 812

68. **Switches** have a pair of non-empty disjoint sets of input, respectively output connectors.

    Constraints: There are two kinds of Switches: on land and on tankers.

    (a) Land switches:
        i. Either input connectors are (to be) bound to Depot output connectors and output connectors are (to be) bound to Refractory input connectors,
        ii. or input connectors are (to be) bound to Refractory output connectors and output connectors are (to be) bound to Depot input connectors,
        iii. or input connectors are (to be) bound to Depot output connectors and output connectors are (to be) bound to Berth Loading Arm input connectors;

    or

    (b) Tanker switches
        i. input connectors are (to be) bound to Berth Loading Arm output connectors and output connectors are (to be) bound to Tank input connectors.

slide 813

```
(68)      ∀ ω:Ω • ∀ u:U • u ∈ obs_Us(ω) ⇒ is_Switch(u) ⇒
              let (ics,ocs)=obs_pCs(u) in card ics ≥ 1 = card ocs ≥ 1 ∧
(68(a))     ((is_Land_Switch(u) ⇒
(68(a)i)        (wf_io_Cs(u,obs_Us(ω))({is_Depot},{is_Refrac}) ∨
(68(a)ii)        wf_io_Cs(u,obs_Us(ω))({is_Refrac},{is_Depot}) ∨
(68(a)iii)       wf_io_Cs(u,obs_Us(ω))({is_Depot},{is_LdArm}))) ∨
(68(b)) ∨ (is_Tanker_Switch(u) ⇒
(68(b)i)         wf_io_Cs(u,obs_Us(ω))({is_LdArm},{is_Tank}))) end
```

slide 814

69. **Depots** have a pair of disjoint non-empty sets of connectors: a set of input connectors and a set of output connectors.

    Constraints:

    - (i) Either Depot input connectors are (to be) bound to outputs of Fill Pumps and depot output connectors are (to be) bound to inputs of Switches.
    - (ii) or Depot input connectors are (to be) bound to outputs of Switches and depot output connectors are (to be) bound to inputs of Fill Pumps.

```
(69) ∀ ω:Ω • ∀ u:U • u ∈ obs_Us(ω) ⇒ is_Depot(u) ⇒
         let (ics,ocs)=obs_pCs(u) in card ics ≥ 1 = card ocs ≥ 1 ∧
(i)      wf_io_Cs(u,(ics,ocs))({is_FilPump},{is_Switch}) ∨
(ii)     wf_io_Cs(u,(ics,ocs))({is_Switch},{is_FilPump}) end
```

slide 815

70. **Refractories** and **Gas Processors** have pairs of disjoint non-empty sets of input, respectively output connectors.

    Constraints: Refractory and Gas Processor input connectors are (to be) bound to Switch output connectors and Refractory and Gas Processor output connectors are (to be) bound to Switch input connectors.

(70) $\forall\ \omega{:}\Omega \bullet \forall$ u:U • u $\in$ obs_Us(ω) $\Rightarrow$
  is_Refract(u)$\vee$is_Separator(u)$\vee$is_Purifier(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in** **card** ics $\geq 1 =$ **card** ocs $\geq 1 \wedge$
    wf_io_Cs(u,(ics,ocs))({is_Switch},{is_Switch}) **end**

slide 816

71. Berth **Loading Arms** have a pair of distinct connectors, one input and one output connector.

    Constraints:
    - (i) When loading, i.e. filling, tanker Tanks, Berth Loading Arm input connectors are (to be) bound to a land Switch output connector, and Berth Loading Arm output connectors to a tanker Switch input connector.
    - (ii) Or vice versa: when unloading the tanks.

(71) $\forall\ \omega{:}\Omega \bullet \forall$ u:U • u $\in$ obs_Us(ω) $\Rightarrow$ is_LdArm(u) $\Rightarrow$
    let (ics,ocs)=obs_pCs(u) **in** **card** ics $\geq 1 =$ **card** ocs $\geq 1 \wedge$
(i)   (wf_io_Cs(u,(ics,ocs))({is_LandSwitch},{is_TankSwitch})(ics,ocs) $\vee$
(ii)   wf_io_Cs(u,(ics,ocs))({is_TankSwitch},{is_LandSwitch})(ics,ocs)) **end**

slide 817

72. Tanker **Tanks** have a pair of distinct connectors, one input (fill) and one output (drain) connector.

    Constraints: Both tanker Tank connectors are (to be) bound to the tanker switch, respectively a tanker Switch output and a tanker Switch input connector.

(72) $\forall\ \omega{:}\Omega \bullet \forall$ u:U • u $\in$ obs_Us(ω) $\Rightarrow$ is_Tank(u) $\Rightarrow$
  let (ics,ocs)=obs_pCs(u) **in** **card** ics $\geq 1 =$ **card** ocs $\geq 1 \wedge$
  wf_io_Cs(u,(ics,ocs))({is_Tanker_Switch},{is_Tanker_Switch}) **end**

slide 818

73. **Sinks**, i.e., End Customers, have just a single, the input connector.

    Constraints: This Sink connector is (to be) bound to a Flow Pump output connector.

(73) $\forall\ \omega{:}\Omega \bullet \forall$ u:U • u $\in$ obs_Us(ω) $\Rightarrow$ is_Sink(u) $\Rightarrow$
  let (ics,ocs)=obs_pCs(u) **in** **card** ics $= 1 \wedge$ **card** ocs $= 0 \wedge$
  wf_i_Cs(u,(ics,ocs))({is_Flow_Pump}) **end**

G.1.4 Attributes slide 819

So far the only properties we have ascribed to units are their identifications and their connectors. They are not really of the kind which we think of as attributes. Unique identifications and unique connectors are abstractions of unique spatial unit locations and unique, mereological relations between "adjacent" units, and can be thought off as pragmatic means of "formalising" such spatial locations and mereological relations. Attributes are not necessarily abstractions. In the following we shall give examples of attributes.

Oil and Gas slide 820

Narrative

74. Units are either made for handling fluids: either liquids (i.e., oil) or for handling gaseous substance (i.e., natural gas). Within mode `liquid` there could be several "submodes": `raw oil`, `liquid petroleum gas`, `naphtalin`, `kerosene`, `diesel fuel`, `lubricating oils`, `paraffin vaxes`, `asphalt`, `petroleum coke`, etc. Similar for mode `gaseous` (but we refrain from elaborating !).

Formalisation

type
 74. Mode == liquid | gaseous
 74. SubMode == raw_oil|liquid_petroleum_gas|naphtalin|kerosene|diesel_fuel
  |lubricating_oil|paraffin_vax|asphalt|petroleum_coke| ...
value
 74. obs_Mode: U $\rightarrow$ Mode
 74. obs_SubMode: U $\rightarrow$ SubMode

Oil and Gas Flow slide 821

Narrative

75. With every unit we can associate the the volumetric flow of oil or gas — as measured into the unit at some input connector or out from the unit at some output connector.
76. We can thus observe from every unit the maximum laminar volumetric flow, a static attribute,
77. as well as the current, dynamic attribute of actual volumetric flow.

Formalisation

type
  VolFlo
value
  75. obs_max_VolFlo: U × C $\overset{\sim}{\rightarrow}$ VolFlo
      pre: obs_max_VolFlo(u,c): c ∈ obs_iCs(u)∪ obs_oCs(u)
  75. obs_cur_VolFlo: U × C $\overset{\sim}{\rightarrow}$ VolFlo
      pre: obs_cur_VolFlo(u,c): c ∈ obs_iCs(u)∪ obs_oCs(u)

Oil and Gas Volumes

slide 822

Narrative

78. With every unit we can associate two notion of oil or gas volumes.
79. For reservoirs there are the initial, definitely non-zero volume of oil (or gas) (a static attribute), and the current, including empty (or zero) volume of oil (or gas) (a dynamic attribute).
80. For depots there are the maximum containable volume of oil or gas (a static attribute), and the current, including empty (or zero) or maximum (i.e., full) volume of oil (or gas).
81. All other units (pipes, valves, pumps, refineries, gas separators, etc., have notions of oil or gas volumes like those of depots, but they are not considered to be storage- or reservoir-like units.
82. Thus the three classes of units have comparable attribute notions of volume of oil (or gas), whether static or dynamic.

slide 823

Formalisation

type
  78. Vol, Vol_0
value
  79–82. obs_maxVol: U → Vol, obs_curVol: U → Vol
      +,−: Vol × Vol $\overset{\sim}{\rightarrow}$ Vol
      <,≤,=,≠,≥,>: Vol × Vol $\overset{\sim}{\rightarrow}$ **Bool**

Vol_0 designate “zero” volume of oil (or gas).

• • •

slide 824

In the following we shall examine the attributes of each of the abstract units that make up the physical plant of the pretroleum and natural gas industry. We remind the reader that these are those units: reservoirs, drain pumps, pipes, flow pumps, joins, forks, valves, switches, depots, refractories, gas processors, berth loading arms, tanker tanks and sinks (i.e., end customers).

Reservoir slide 825

Narrative

83.
84.
85.
86.
87.

slide 826

Formalisation

type
83.
84.
85.
86.
87.

Drain Pumps slide 827

Narrative

88.
89.
90.
91.
92.

slide 828

Formalisation

type
88.
89.
90.
91.
92.

Pipes slide 829

Narrative

A pipe ...

93.
94.
95.

slide 830

Formalisation

type
  93.
  94.
  95.

Flow Pumps <span>slide 831</span>

Narrative

A flow pump is a device used to move fluids, such as gases, liquids or slurries. A flow pump displaces a volume by physical or mechanical action. One common misconception about flow pumps is the thought that they create pressure. Flow pumps alone do not create pressure they only displace fluid causing a flow. Adding resistance to flow causes pressure.

96. With a pump, whether a drain pump, a flow pump, or other (not treated here), we can associate the static attribute of that pump's maximum sustainable volumetric flow rate.
97. And, with a pump, we can associate the dynamic attribute of the pump's forced current volumetric flow rate.
98. The observed current volumetric flow rate out of a pump must thus equal the pump's forced current volumetric flow rate.

The volumetric flow rate in fluid dynamics and hydrometry (also known as volume flow rate or rate of fluid flow) is the volume of fluid which passes through a given surface per unit time (for example cubic meters per second).

Formalisation

type
  96. obs_MaxPumVolFloRat: U → VolFlo
  96. obs_ForcedVolFloRat: U → VolFlo
  <,≤,=,≠,≥,>: VolFlo × VolFlo → **Bool**
axiom
  97. ∀ u:FlP(p)•obs_ForcedVolFloRat(u)≤obs_MaxPumVolFloRat(u)
  98. **let** c:C•c ∈ obs_oCs(u) **in** obs_VolFlo(u,c)=obs_ForcedVolFloRat(u) **end**

Joins slide 832

Narrative

99.
100.

101.
102.
103.

slide 833

Formalisation

type
  99.
  100.
  101.
  102.
  103.

Forks slide 834

Narrative

104.
105.
106.
107.
108.

slide 835

Formalisation

type
  104.
  105.
  106.
  107.
  108.

Valves slide 836

Narrative

A valve is a device that regulates the flow of a fluid (gases, fluidized solids, slurries, or liquids) by opening, closing, or partially obstructing various passageways. Valves are technically pipe fittings, but are usually discussed separately.

109. We think of a valve as always having open input connections but where each output connection can be regulated to be closed, partially open, or fully open.

Formalisation

```
type
   109.  Valve_Setting = C $\overrightarrow{m}$ Frac
         Frac = {| f:Real • 0≤f≤1 |}
value
   109.  obs_Valve_Setting: Valve → Valve_Setting
axiom
   109.  ∀ u:Valve • dom obs_Valve_Setting(u) = xtr_oCs(u)
```

slide 837

Flow Properties

Narrative:

110. Let VolFlo_0 designate zero flow.
     Let leakage be a unit determined quantity, VolFlo_{Leak_u}.
111. If a valve output connector is closed then the observed flow at that connector must be zero.
112. If all valve output connectors are closed then the observed sum flow into that unit must be between VolFlo_0 and VolFlo_{Leak_u}.
113. In general, for valves, as for other units, the flow into the unit equals the flow out of the unit minus the leakage loss in the unit, for example due to leaks.

slide 838

Formalisation:

```
value
   110.  VolFlo$_0$, VolFlo$_{Leak_u}$
axiom
   111.  ∀ u:Valv(v)•∃ c:C•c ∈ obs_oCs(u)⇒
              (obs_Valve_Setting(u))(c)=0 ⇒ obs_VolFlo(u,c)=VolFlo$_0$,
   111.  ∀ u:Valv(v)•∀ c:C•c ∈ obs_oCs(u)⇒(obs_Valve_Setting(u))(c)=0⇒
              VolFlo$_0$ ≤ sum_flo(u,obs_oCs(u) ≤ VolFlo$_{Leak_u}$,
   113.  ∀ u:Valv(v)•
              sum_flo(u,obs_iCs(u)) = sum_flo(u,obs_oCs(u)) + VolFlo$_{Leak_u}$
value
   +: Vol_Flo × Vol_Flo → Vol_Flo
   sum_flo: U × C-set → VolFlo
   sum_flo(u,cs) ≡
      case cs of
         {} → VolFlo$_0$,
         (c)∪ cs′ → obs_VolFlo(u,c) + sum_flo(u,cs\{c})
      end
```

Switches

slide 839

An Abstraction

We first refer to Figs. F.14 on page 184 and F.15 on page 185. Instead of describing the class of complicated and composite structures of pipes, forks, joins and valves (shown in those figures) we abstract them into atomic units and call them switches. From earlier we have constrained switches to "bridge" depots with refractory towers (and vice versa, depots with berth loading arms (and vice versa), and berth loading arms with tanker tanks(and vice versa), yes even to bridge pipelines with depots and depots with – say – truck distribution nets. (The latter has not [yet] been mentioned.) We can thus assume the connections on both the input and the output side of a switch to be (like) valves. slide 840

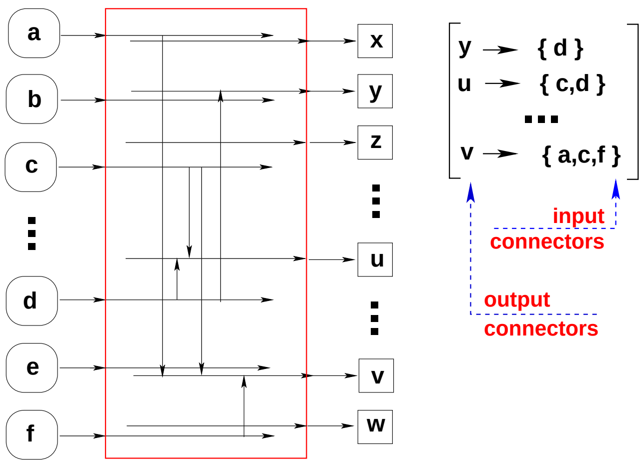

Fig. G.1. A snapshot state of an "abstract, programmable" switch

slide 841

Narrative

114. Recall that switch input connectors connect to depot (land switch) or load arm (tanker switch) or tank (tanker switch) output connectors, cf. Fig. G.1,
115. and that switch output connectors connect to load arm (land switch) or depot (land switch) or tank (tanker switch) input connectors
116. A dynamic attribute of an abstracted switch is that switch output connectors to sets of switch input connectors — such that
    (a) if an actual switch output connector is not in the definition set of the map, then the actual switch output connector represents a closed valve;

(b) if an actual switch input connector is in a range set of the map, then the actual switch input connector represents an opened valve.

117. It is easy to prove that with this model of a switch any set of input connectors can be connected to any set of output connectors.

slide 842

Formalisation

type
116. SW = C $\overrightarrow{m}$ C-**set**
value
116. obs_SW: Switch → SW
axiom
116. ∀ u:Swi(sw) •
dom obs_SW(u)⊆obs_oCs(u) ∧ ∪ **rng** obs_SW(u)⊆obs_iCs(u)
116(a).
116(b).
117.

Depots

slide 843

Narrative

118.
119.
120.
121.
122.

slide 844

Formalisation

type
118.
119.
120.
121.
122.

Refractories

slide 845

Narrative

123.
124.
125.
126.
127.

slide 846

Formalisation

type
123.
124.
125.
126.
127.

Gas Processing Units slide 847

Narrative

128.
129.
130.
131.
132.

slide 848

Formalisation

type
128.
129.
130.
131.
132.

Load Arms slide 849

An Abstraction

We first refer to Fig. F.16 on page 185. slide 850

Narrative

133.
134.
135.
136.
137.

slide 851

Formalisation

```
type
    133.
    134.
    135.
    136.
    137.
```

Tanks

slide 852

Narrative

138.
139.
140.
141.
142.

slide 853

Formalisation

```
type
    138.
    139.
    140.
    141.
    142.
```

Sinks

slide 854

Narrative

143.
144.
145.
146.
147.

slide 855

Formalisation

```
type
    143.
    144.
    145.
    146.
    147.
```

G.1.5 Paths and Routes

slide 856

The structure of units form a graph. Pipes are the edges of the graph and nodes are the vertices of the graph.

Narrative

148. A path is a concept. A path of the petroleum industry is a triple, (c_i, u_j, c_k), an input connector (or `nil`), a unit identifier, and an output connector (or `nil`),
    (a) such that the unit, u, identified by the unit identifier, u_j, is a unit of the oil/natural gas system, ω, and
    (b) such that the connector pairs, (c_i, c_k), are pairs of distinct input/output connectors of the unit, u, identified by the unit identifier, u_j,
    (c) or either c or c' is `nil` to designate paths of reservoir, depot, loading arm or sink units.
149. A route is a concept. A route of the oil/natural gas system is a sequence of one or more paths such that adjacent paths share non–`nil` 'connector'.

Formalisation

```
type
   148.   Path = mkUP(ci:(C|{nil}),ui:UI,co:(C|{nil}))
   149.   Route = Path*
value
   148(c).  is_Initial: U → Bool, is_Final: U → Bool
   148(c).  is_Initial(u) ≡ is_Res(u) ∨ is_Dep(u) ∨ is_LdArm(u)
   148(c).  is_Final(u) ≡ is_Dep(u) ∨ is_LdArm(u) ∨ Sink(u)
axiom
                  ∀ ω:Ω •
                     ∀ mkUP(c,uj,c′):PP •
   148(a)               ∃ u:U • u ∈ xtr_Us(ω)∧uj=obs_UI(u) ⇒
   148(b)-148(c).          is_Initial(u)→c={nil},_→c ∈ obs_iCs(u) ∧
   148(b)-148(c).          is_Final(u)→c′={nil},_→c′ ∈ obs_oCs(u)
   149.        ∀ r:Route • ∀ i:Nat•{i,i+1}⊆inds r
                        ⇒ let (_,_,c)=r(i),(c′,_,_)=r(i+1) in c=c′≠nil end
```

slide 857

slide 858

Narrative

150. An oil/natural gas system, ω, determines a set of routes as follows:
    (a) The empty list, $\langle\rangle$, is a route.
    (b) A singleton list, $\langle p\rangle$, of a path, p, of ω, is a route (so all such singleton lists of ω form routes).
    (c) If r and r' are routes of ω then $r\widehat{\ }r'$, their concatenation, is a route of ω.
    (d) No other route is a route of ω unless it follows from a finite number of applications of prescriptions 150(a), 150(b) and 150(c).

slide 859

Formalisation

We define a function which calculates all static routes of an oil/natural gas system, ω, in terms of another function which calculates all static paths of an oil/natural gas system, ω:

```
value
    calc_Sta_Paths: $\Omega \to$ Path-set
    calc_Sta_Paths($\omega$) $\equiv$
        $\{\langle\rangle\}\cup\{$mkUP(con,uj,con$'$)|u:U•u $\in$ xtr_Us($\omega$)$\Rightarrow$
            con $\in$ obs_iCs(u)$\cup\{$nil$\}$ $\wedge$ con$'$ $\in$ obs_oCs(u)$\cup\{$nil$\}\wedge$uj=obs_UI(u)$\}$

  150.    calc_Sta_Routes: $\Omega \to$ Route-set
  150(a). calc_Sta_Routes($\omega$) $\equiv$
  150(b).     let rs =
                  $\{\langle$path$\rangle$|path:Path • path $\in$ calc_Sta_Paths($\omega$)$\}$ $\cup$
                  $\{$r$\hat{}$r$'$|r,r$'$:Route•$\{$r,r$'\}\subseteq$rs$\wedge$r$\neq\langle\rangle\wedge$r$'\neq\langle\rangle\wedge$lst_C(r)=frst_C(r$'$)$\neq$nil$\}$
  150(c).     in rs end

  frst_C, lst_C: Route $\overset{\sim}{\to}$ C|$\{$|nil|$\}$
  frst_C($\langle$mkUP(con,_,_)$\rangle\hat{}$r$'$) $\equiv$ con, lst_C(r$'\hat{}\langle$mUP(_,_,con)$\rangle$) $\equiv$ con
```

slide 860

Observations

- The "statics" of connections
  - ⋆ "start" at reservoirs, depots or loading arms and
  - ⋆ end at depots, loading arms, respextively sinks.
- Routes cannot "start earlier" than reservoir, depot or loading arm output connectors.
- Routes cannot go further than depot, loading arm or sink input connectors.
- The "dynamics" of loading arms really means that route calculation depends on the state of the system (i.e., ω). slide 861
- Static routes cover
  - ⋆ pipelines from reservoirs to the input depots
    - ◇ of refineries,
    - ◇ gas processors or
    - ◇ harbours; slide 862
  - ⋆ routes through
    - ◇ refineries
      - ○ from refinery input depots,
      - ○ via depot-to-refractory tower switches,
      - ○ refracfory towers and
      - ○ refracfory tower-to-output depot switches,
      - ○ to refinery output depots;
    - ◇ gas processors — similarly; or

    ◇ harbours
        ○ from harbour input depots
        ○ via depot-to-loading arm switches
        ○ to loading arms;

    and

slide 863

  ⋆ routes from
    ◇ refinery output depots
        ○ to harbour input depots,
        ○ or via pipelines or truck distribution nets[2] to sinks.
    ◇ gas processor output depots
        ○ to harbour input depots
        ○ or via pipelines or truck distribution nets[3] to sinks.

    and

    ◇ harbour output depots
        ○ to refinery or gas processor input depots,
        ○ or via pipelines or truck distribution nets[4] to sinks.

- Dynamic routes extend static routes through the varying
  ⋆ loading-arm to tank or
  ⋆ tank to loading arm

  connections.

Acyclic Networks

slide 864

Narrative

151. The routes of an oil/natural gas system are not cyclic,
152. that is, no node or pipe identifier must occur more than at most once.

Formalisation

```
value
    151. acyclic: Route → Bool
    152. acyclic(r) ≡
             ∀ i,j:Nat • {i,j}⊆ index r ∧ i≠j ⇒
                 let (mkUP(_,ui,_),mkUP(_,uj,_)) = (r(i),r(j)) in ui≠uj end
```

[2] We shall; not cover truck distribution nets in this example.
[3] See Footnote 2.
[4] See Footnote 2.

Uniform Routes

slide 865

Narrative

153. A uniform route is a route all of whose units transport or contain the same kind of oil or gas product.
154. Routes between
    (a) reservoirs and input depots,
    (b) output depots and ...
    (c)
    (d)

slide 866

Formalisation

153.
154.
154(a).
154(b).
154(c).
154(d).

G.1.6 Kirchhoff's Law

slide 867

General

The petroleum and natural gas industry is about (i) *production* of crude oil and natural gas; (ii) *transportation*, through pipelines, ships and truck distribution of oil and natural gas; (iii) the *processing*, in refineries, of crude oil, and gas separators, purifiers, etc., of natural gas; and (iv) the final *disposal* of these end products to consumers (here modelled as sinks). slide 868

In all of this petroleum and natural gas flows through the system: from reservoirs via pipelines and tankers, and via refineries and gas processing units to end consumers. Major breaks in these flows are at refineries, gas processing units, depots, and at tankers. slide 869

But otherwise there is a "more-or-less" uninterrupted, i.e., "constant" flow.

What flows into the system at reservoirs eventually, in uninterrupted movements, flow into depots from "myriads" of: drain pumps, pipes, valves, joins and forks, and flow and fill pumps into depots. slide 870

What flows out of refinery (respectively gas processing plant) input depots, eventually, in uninterrupted movements, flow into refinery (respectively gas processing plant) output depots: via switches feeding into refractory towers (respectively gas separators, purifiers, etc.), and, from there, via switches feeding into output depots. slide 871

What flows out of harbour depots in uninterrupted movements, flow into tanker tanks; and what flows out of tanker tanks in uninterrupted movements, flow into harbour depots.

slide 872

Etcetera.

For each of these "uninterrupted movement" sub-systems a variant of *Kirchhoff's Law* applies: *"What flows in flows out: some is lost through leakage, but hopefully most is preserved throughout."* We shall now make this adherence to *Kirchhoff's Law* precise.

An Approximation

slide 873

Narrative

155.
156.
157.
158.
159.
160.

slide 874

Formalisation

155.
156.
157.
158.
159.
160.

G.2 Domain Operations

slide 875

Most of the units can be operated upon: drain pumps, pipes, flow pumps, joins, forks, valves, switches, depots, refractories, gas processors, berth loading arms, tanker tanks and sinks (i.e., end customers). We have left out operating directly upon reservoirs. Operations upon units either observe static or dynamic attributes, i.e., states, or change these states.

G.2.1 [1] Drain Pumps slide 876

G.2.2 [2] Pipes slide 877

G.2.3 [3] Flow Pumps slide 878

G.2.4 [4–5] Joins and Forks slide 879

G.2.5 [6] Valves slide 880

G.2.6 [7] Switches slide 881

G.2.7 [8] Depots slide 882

G.2.8 [9] Refractories slide 883

G.2.9 [10] Gas Processors slide 884

G.2.10 [11] Loading Arms slide 885

G.2.11 [12] Tanks slide 886

G.2.12 [13] Sinks slide 887

G.2.13 Discussion slide 888

G.3 Domain Events slide 889

The environment of the petroleum industry is a cause for external events. Most of the units can likewise be the originators of events: reservoirs, drain pumps, pipes, flow pumps, joins, forks, valves, switches, depots, refractories, gas processors, berth loading arms, tanker tanks and sinks (i.e., end customers). Events reflect state changes: either of static or dynamic unit attributes, or of the environment.

G.3.1 [0] Reservoirs
slide 890

G.3.2 [1] Drain Pumps
slide 891

G.3.3 [2] Pipes
slide 892

G.3.4 [3] Flow Pumps
slide 893

G.3.5 [4–5] Joins and Forks
slide 894

G.3.6 [6] Valves
slide 895

G.3.7 [7] Switches
slide 896

G.3.8 [8] Depots
slide 897

G.3.9 [9] Refractories
slide 898

G.3.10 [10] Gas Processors
slide 899

G.3.11 [11] Loading Arms
slide 900

G.3.12 [12] Tanks
slide 901

G.3.13 [13] Sinks
slide 902

G.3.14 Discussion
slide 903

G.4 Domain Behaviours
slide 904

All units exhibit behaviours: reservoirs, drain pumps, pipes, flow pumps, joins, forks, valves, switches, depots, refractories, gas processors, berth loading arms, tanker tanks and sinks (i.e., end customers). In addition subsystems, i.e., compositions of units exhibit behaviours: pipelines, refineries, harbours, etc.

G.4.1 “Total” System Behaviour

slide 905

slide 906

slide 907

slide 908

G.4.2 Unit Behaviours

slide 909

[1] Drain Pumps

slide 910

[2] Pipes

slide 911

[3] Flow Pumps

slide 912

[4–5] Joins and Forks

slide 913

[6] Valves

slide 914

[7] Switches

slide 915

[8] Depots

slide 916

[9] Refractories

slide 917

[10] Gas Processors

slide 918

[11] Loading Arms

slide 919

[12] Tanks

slide 920

[13] Sinks

slide 921

G.4.3 Subsystem Behaviours

slide 922

Pipeline

slide 923

Refinery

slide 924

Harbour

slide 925

G.4.4 Discussion

slide 926

G.5 Review, Summary and Discussion

slide 927

slide 928

H

Support Technologies

slide 930

Chapter 11 (Pages 97–98) complements the present appendix. slide 931

slide 932

I

Management and Organisation

slide 933

Chapter 12 (Pages 101–107) complements the present appendix. slide 934

J

Rules and Regulations

slide 936

Chapter 13 (Pages 109–112) complements the present appendix. slide 937

K

Scripts, Licenses and Contracts slide 939

Chapter 14 (Pages 115–117) complements the present appendix.

K.1 Overview slide 940

K.2 Scripts slide 941

K.2.1 Time Tables slide 942

We shall view timetables as scripts.

In this section (that is, Pages 233–243) 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. K.8, on licenses for bus traffic, we shall assume the timetable scripts of this section. slide 943

We all have some image of how a timetable may manifest itself. Figure K.1 on the following page shows some such images.

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

[1] We do not distinguish between a time and a time description. That is, when we say January 24, 2009, 17: 21 we mean it either as a description of the time

BUS TIME
EL PRÁCTICO
Desde el centro de Iguazú a Eterno Reverdecer
Desde Eterno Reverdecer al centro de Iguazú
BUSSRUTER
532 Otta - Høvringen - Smuksjøseter
532 Smuksjøseter - Høvringen - Otta

Fig. K.1. Some bus timetables: Italy, India and Norway

description may cover several bus rides. The former have unique identifications and so has the latter. The times of the latter are the approximate times at which the bus of that bus line and bus identification is supposed to be at respective stops. You may think of the bus line identification to express something like "The Flying Scotsman", and the bus ride identification something like "The 4.50 From Paddington".

The Syntax of Timetable Scripts

slide 944

161. 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.
162. A TimeTable associates to Bus Line Identifiers a set of Journies.
163. Journies are designated by a pair of a BusRoute and a set of BusRides.
164. 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.
165. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
166. 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.
167. 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.
168. A Fraction is a **Real** properly between 0 and 1.
169. The Journies must be well_formed in the context of some net.

slide 945

type
161. T, BLId, BId
162. TT = BLId $\overrightarrow{m}$ Journies
163. Journies′ = BusRoute × BusRides

at which this text that you are now reading was LaTeX compiled, and as "that time !".

164. BusRoute = BusStop × BusStop* × BusStop
165. BusRides = BId $\overrightarrow{m}$ BusSched
166. BusSched = T × T* × T
167. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
168. Frac = {|r:**Real**•0<r<1|}
169. Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}

The free n in ∃ n:N • wf_Journies(j)(n) is the net given in the license. slide 946

slide 947

Well-formedness of Journies

170. A set of journies is well-formed
171. if the bus stops are all different[2],
172. if a defined notion of a bus line is embedded in some line of the net, and
173. if all defined bus trips (see below) of a bus line are commensurable.

value
170. wf_Journies: Journies → N → **Bool**
170. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
171.   diff_bus_stops(bs1,bsl,bsn) ∧
172.   is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
173.   commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

slide 948

Well-formedness of Journies

174. The bus stops of a journey are all different
175. if the number of elements in the list of these equals the length of the list.

value
174. diff_bus_stops: BusStop × BusStop* × BusStop → **Bool**
174. diff_bus_stops(bs1,bsl,bsn) ≡
175.   **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 949

176. To explain that a bus line is embedded in a line of the net
177. let us introduce the notion of all lines of the net, lns,
178. and the notion of projecting the bus line on link sector descriptors.
179. 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.

[2] This restriction is, strictly speaking, not a necessary domain property. But it simplifies our subsequent formulations.

```
value
176. is_net_embedded_bus_line: BusStop* → N → Bool
176. is_net_embedded_bus_line(bsl)(hs,ls)
177.   let lns = lines(hs,ls),
178.       cbln = compress(proj_on_links(bsl)(elems bsl)) in
179.   ∃ ln:Line • ln ∈ lns ∧ cbln ∈ projs_on_links(ln) end
```

slide 950

180. Projecting a list (*) of BusStop descriptors (mkBS(hi,li,f,hi′)) onto a list of Sector Descriptors ((hi,li,hi′))
181. we recursively unravel the list from the front:
182. if there is no front, that is, if the whole list is empty, then we get the empty list of sector descriptors,
183. else we obtain a first sector descriptor followed by those of the remaining bus stop descriptors.

```
value
180. proj_on_links: BusStop* → SectDescr*
180. proj_on_links(bsl) ≡
181.   case bsl of
182.     ⟨⟩ → ⟨⟩,
183.     ⟨mkBS(hi,li,f,hi′)⟩⌢bsl′ → ⟨(hi,li,hi′)⟩⌢proj_on_links(bsl′)
183.   end
```

slide 951

184. By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.
185. The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.
186. We express the function recursively.
187. If the argument sector descriptor list an empty result sector descriptor list is yielded;
188. else
189. 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”
190. in front of the compression of the rest of the argument sector descriptor list.

slide 952

```
184. compress: SectDescr* → SectDescr-set → SectDescr*
185. compress(sdl)(sds) ≡
186.   case sdl of
187.     ⟨⟩ → ⟨⟩,
188.     ⟨sd⟩⌢sdl′ →
189.        (if sd ∈ sds then ⟨sd⟩ else ⟨⟩ end)
190.        ⌢compress(sdl′)(sds\{sd}) end
```

In the last recursion iteration (line 190.) the continuation argument sds\{sd} can be shown to be empty: {}. slide 953

191. We recapitulate the definition of lines as sequences of sector descriptions.
192. Projections of a line generate a set of lists of sector descriptors.
193. 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
191. Line′ = (HI×LI×HI)*,
191. Line = {| l:Line′ • wf_Line(l′) |}
value
191. wf_Line: Line′ → Bool
191. wf_Line(l) ≡
191.   ∀ i:Nat • {i,i+1}⊆inds l⇒
191.     let ((_,_,lij),(lik,_,_))=(l(i),l(i+1)) in lij=lik end
192. projs_on_links: Line → Line′-set
192. projs_on_links(ln) ≡
193.   {⟨isl(i)|i:⟨1..len isl⟩⟩|isx:Nat-set•isx⊆inds ln∧isl=sort(isx)}
```

slide 954

194. 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.
195. The result list of (arbitrary) indices must contain all the members of the argument set;
196. and "earlier"elements of the list must precede, in value, those of "later" elements of the list.

```
value
194. sort: Nat-set → Nat*
194. sort(isx) as isl
195.   post card isx = lsn isl ∧ isx = elems isl ∧
196.        ∀ i:Nat • {i,i+1}⊆inds isl ⇒ isl(i)<isl(i+1)
```

slide 955

197. The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:
198. All the intermediate bus stop times must equal in number that of the bus stop list.
199. 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.

200. If the list of intermediate bus stops is empty, then there is only the bus stops of origin and destination, and they must be exist and must fit time-wise.
201. 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.
202. 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.
203. As for Item 202 but now with respect to last, resp. destination bus stop.
204. And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops
205. they must exist and fit time-wise.

slide 956

```
value
197. commensurable_bus_trips: Journies → N → Bool
197. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
198.   ∀ (t1,til,tn):BusSched•(t1,til,tn)∈ rng js∧len til=len bsl∧
199.     case len til of
200.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
201.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧fit((til(1),t2),(bsl(1),bsn))(hs,ls),
202.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
203.           real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
204.           ∀ i:Nat•{i,i+1}⊆inds til ⇒
205.             real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

slide 957

206. 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:
207. All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).
208. There exists links, l, l′, for the identified bus stop links, li, li′,
209. such that these links connect the identified bus stop hubs.
210. Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops
211. The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.
212. 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.
213. The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.
214. So we leave also this as an exercise to the reader to complete.

slide 958

```
206.  real_and_fit: (T×T)×(BusStop×BusStop) → N → Bool
206.  real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
207.    {hi,hi′,hi″,hi‴}⊆his(hs)∧
208.    ∃ l,l′:L•{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
209.      obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
210.    afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

211.  distance: BusStop × BusStop → N → Distance
212.  distance(bs1,bs2)(n) ≡ ... [ left as an exercise ! ] ...

213.  afit: TI → Distance → Bool
214.    [ time interval fits distance between bus stops ]
```

The Pragmatics of Timetable Scripts slide 959

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. slide 960

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.

215. 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.

```
215 have_compatible_BLIds: TT × TT → Bool
215 have_compatible_BLIds(tti,ttj) ≡
215   ∀ blid:BLId • blid ∈ dom tti ∩ dom ttj
215     ⇒ let (bri,_)=tti(blid),(brj,_)=ttj(blid) in bri=brj end
```

slide 961

216. 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.
217. 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
216 are_similar_Js: Journies × Journies → Bool
216 are_similar_Js((bri,_),(brj,_)) ≡ bri=brj

217 is_sub_TT: TT × TT → Bool
217 is_sub_TT(stt,tt) ≡
217   ∀ sblid,blid:BLId•sblid=blid∧sblid ∈ dom stt∧blid ∈ dom tt
217     ⇒ ∀ (sbr,sbrs),(br,brs):Journies•(sbr,sbrs)=stt(sblid)∧(br,brs)=tt(blid)
217       ⇒ sbr=br ∧ ∀ bid:BId•bid ∈ dom sbrs ∩ dom br
217         ⇒ sbrs(bid)=brs(bid)
217   pre have_compatible_BLIds(stt,tt)
```

slide 962

218. We can thus generate all sub-timetables of a timetable.

```
218 all_sub_TTs: TT → TT-set
218 all_sub_TTs(tt) ≡ {stt|stt:TT•is_sub_TT(stt,tt)}
```

219. Two timetables, stt_i and stt_j, are said to be disjoint if they share no same bus line identifier bus rides.

```
219 are_disjoint_TTs: TT × TT → Bool
219 are_disjoint_TTs(tti,ttj) ≡
219   ∀ blidi,blidj:BLId•blidj=blidj∧blidi ∈ dom tti∧blidj ∈ dom ttj
219     ⇒ dom tti(blidi) ∩ dom ttj(blidj) = {}
219   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 963

220. Two timetables can be merged into one timetable provided they are disjoint.
221. Merging two disjoint timetables result in a timetable which has exactly the bus line identified journies of either of the timetables.

```
220 can_be_merged_TTs: TT × TT → Bool
220 can_be_merged_TTs(tti,ttj) ≡ are_disjoint_TTs(tti,ttj)

221 merge_TTs: TT × TT → TT
221 merge_TTs(tti,ttj) as tt
221   pre are_disjoint_TTs(tti,ttj) [ i.e., have_compatible_BLIds(tti,ttj) ]
221   post is_sub_TT(tti,tt′)∧is_sub_TT(ttj,tt′)
221       ∧ dom tt = dom tti ∪ dom ttj
221       ∧ ∀ blid:BLId•blid ∈ dom tt ∧ blid ∈ dom tti ∪ dom ttj
221         ⇒ let ((br,brs),(bri,brsi),(brj,brsj)) = (tt(blid),tti(blid),ttj(blid)) in
221           dom brsi ∩ dom brsj = {} ∧ dom brsi ∪ dom brsj = dom brs
221           ∧ brs = brsi ∪ brsj end
```

slide 964

From a timetable one can construct any number of sub-timetables.

222. 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
222 cons_STT: TT × (BLId →m BId-set) → TT
222 cons_STT(tt,id_map) ≡
222   [ blid ↦ (tt(blid))(bid)
222      | blid:BLId,bid:BId • blid ∈ dom id_map ∧ bid ∈ id_map(blid) ]
222   pre dom id_map ≠ {} ∧ dom id_map ⊆ dom tt ∧
222       ∧ ∀ blid:BLId•blid ∈ dom(tt)
222           ⇒ id_map(blid)≠{}∧id_map(blid)⊆rng tt(blid)
```

slide 965

223. 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.

```
223 cons_compl_STT: TT × (BLId →m BId-set) → TT
223 cons_compl_STT(tt,id_map)
223   let idmap = [ blid ↦ bids | blid:BLId,bids:BId-set
223           • (blid ∈ dom tt \ dom id_map ∧ bids=dom tt(blid))
223             ∨ blid ∈ dom tt ∩ dom id_map ∧ bids=id_map(blid) ]
223   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 966

Some auxiliary functions might come in handy at a later stage.

224. 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.
225. Given a bus line identifier and a bus ride identifier to inquire whether they together identify a proper bus ride of a given timetable.
226. 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.

227. 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 967

```
value
224. is_def: BLId×TT→Bool,
224. is_def(blid,tt) ≡ blid ∈ dom tt ∧ tt(blid)≠[ ]

225. is_def: BLId×BId×TT→Bool,
225. is_def(blid,bid,tt) ≡ dom tt ∧ bid ∈ dom tt(blid)

226. is_def: BId×TT→Bool,
226. is_def(bid,tt) ≡ ∃ blid:BLId • is_def(blid,bid,tt)

227. inquire: BLId×BId×TT ~→(BusRoute×BusSched),
227. inquire(blid,bid,tt) ≡
227.   let (br,brs)=tt(blid) in (br,brs(bid)) end
227.   pre is_def(blid,bid,tt)
```

The Semantics of Timetable Scripts slide 968

One form of timetable denotations is the bus traffic implied by a timetable.

Bus Traffic

228. We postulate a type of Buses.
229. 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.)
230. Bus traffic maps discrete times into the pair of a bus net and the positions of buses.

slide 969

231. A bus positions is either at a hub, on a link or at a bus stop.
232. When a bus is at a hub we can also observe from which link it came and to which link it proceeds.
233. 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.
234. When a bus is at a bus stop — which is like "on a link" — we can observe that bus stop accordingly.
235. Fractions have also be described earlier.

slide 970

```
type
228.  Bus
value
229.  obs_X: Bus → X
```

type
230. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos)))
231. BPos = atHub | onLnk | atBS
232. atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
233. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
234. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
235. 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 971

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)
```

We leave it to the reader to define the on_time predicate. slide 972

K.2.2 Discussion

We have built the foundations for a theory of timetables. We have not yet formulated theorems let alone proven any such.

K.2.3 Aircraft Flight Simulator Script slide 973

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 974
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 975

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 976

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.

K.2.4 Bill of Lading

slide 977

slide 978

We show a template: the … are to be fillled in.

- 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: ...
  - ⋆ Freight Advance: ...
  - ⋆ Time used for loading: ...
    - ⋄ Days: ...
    - ⋄ Hours: ...
- Dated (by charter part: ...): ...
  - ⋆ 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.
    - ⋄ (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. slide 979 slide 980
    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 981

⋆ (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.

K.3 License and Contract Languages

slide 982

By a **domain** $\boxed{\textbf{script}}$ **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** $\boxed{\textbf{contract}}$ **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.

K.4 The Performing Arts: Producers and Consumers slide 983

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).

K.4.1 Operations on Digital Works slide 984

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. slide 985 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.

K.4.2 License Agreement and Obligation slide 986

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.

K.4.3 Two Assumptions slide 987

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 slide 988 the fulfilment of the first). The second assumption is that the consumer is not allowed to, or cannot operate[3] on the works by own means (software, machines). The second assumption implies that acceptance of a license results 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.

[3] render, copy and edit

K.4.4 Protection of the Artistic Electronic Works slide 989

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.

K.4.5 A License Language slide 990

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 991

(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 992

(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 993

(3.) A command is either a render, a copy or an edit or a readme command, or a sub-licensing (sub-license) command. slide 994

(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 995

(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 996

(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 997

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 998

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 999

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 1000

type
5. V
6. Act = Ln $\times$ (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 1001

(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 1002

(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 1003

(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 1004

(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 1005

(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 1006
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 1007

(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. 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. slide 1008

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.

K.5 A Hospital Health Care License Language slide 1009

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.

K.5.1 Patients and Patient Medical Records slide 1010

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.

K.5.2 Medical Staff slide 1011

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.

K.5.3 Professional Health Care

The issue is now, given that we record these licenses, their being issued and being honoured, whether the handling of patients at hospitals follow, or does not follow properly issued licenses.

We refer to the abstract syntax formalised below (that is, formulas 1.–5.). The work on the specific form of the syntax has been facilitated by the work reported in [8].[4]

[4] As this work, [8], has yet to be completed the syntax and annotations given here may change.

K.5.4 A Notion of License Execution State

slide 1012

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. K.2 for an idealised hospitalisation plan. slide 1013

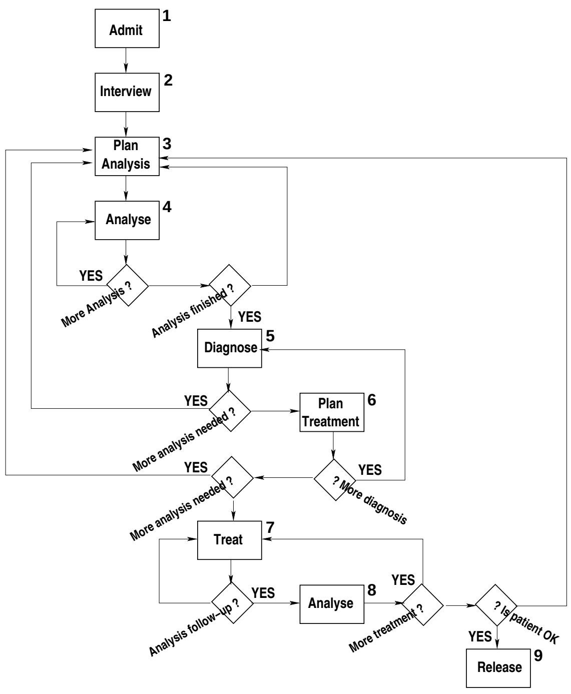

Fig. K.2. An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

slide 1014

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!

K.5.5 The License Language

slide 1015

The syntax has two parts. One for licenses being issued by licensors. And one for the actions that licensees may wish to perform. slide 1016

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**

The above syntax is correct RSL. But it is decorated! The subtypes {|**boldface keyword**|} are inserted for readability. slide 1017

(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.

(5.) A command (Cmd) is a state-labelled statement. slide 1018

(3.) A sub-license just states the command set that the sub-license licenses. As for the Artistic License Language the licensee chooses an appropriate subset of commands. The context "inherits" the name of the patient. But the sub-licensee is explicitly mandated in the license!

(6.) An alternative is also just a set of commands. The meaning is that either the licensee choose to perform the designated actions or, as for ML, but now freely choosing the sub-licensee, the licensee (now new licensor) chooses to confer actions to other staff. slide 1019

(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 1020

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

slide 1021

(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. slide 1022 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.

(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.

K.6 Public Government and the Citizens slide 1023

K.6.1 The Three Branches of Government

By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)[5], understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public government. The Three Branches of Government slide 1024 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.).

K.6.2 Documents slide 1025

A crucial means of expressing public administration is through *documents*.[6] 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 1026

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

K.6.3 Document Attributes slide 1027

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 1028

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

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

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

K.6.4 Actor Attributes and Licenses

slide 1029

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*.

K.6.5 Document Tracing

slide 1030

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.

K.6.6 A Document License Language

slide 1031

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

```
type
0. Ln, An, Cfn
1. L          == Grant | Extend | Restrict | Withdraw
2. Grant      == mkG(license:Ln,licensor:An,granted_ops:Op-set,licensee:An)
3. Extend     ==  mkE(licensor:An,licensee:An,license:Ln,with_ops:Op-set)
4. Restrict   == mkR(licensor:An,licensee:An,license:Ln,to_ops:Op-set)
5. Withdraw == mkW(licensor:An,licensee:An,license:Ln)
6. Op         == Crea|Edit|Read|Copy|Licn|Shar|Rvok|Rlea|Rtur|Calc|Shrd
```

slide 1032

```
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 1033

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

slide 1034

(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 1035

(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[8]) 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.[9] The "based on" documents are moved from licensor to licensee.

slide 1036

(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 1037

(10.) **Reading** a document only changes its "having been read" status (etc.) — as per [24]. 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 1038

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

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

[9] They can therefore be traced (etc.) — as per [24].

(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 [24]. 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 1039

(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 1040

(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 1041

(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 1042

(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 1043

(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 1044

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 1045

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 1046

(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 1047

(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 1048

(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 1049

(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 1050

(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 1051

(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.

K.7 Discussion: License Language Comparisons slide 1052

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. slide 1053

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.

K.7.1 Work Items slide 1054

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.

K.7.2 Operations slide 1055

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).

K.7.3 Permissions and Obligations slide 1056

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).

K.8 A Bus Transport Contract Language slide 1057

K.8.1 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 1058

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[10]. The cancellation rights are spelled out in the contract[11]. A subcontractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor[12]. Etcetera. slide 1059

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. K.2.1. 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

[10] We do not treat this aspect further in this book.
[11] See Footnote 10.
[12] See Footnote 10.

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 1060

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[13] We assume that such insertions must also be reported back to the contractor. slide 1061

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 1062

Contracted Operations, An Overview

So these are the operations that are allowed by a contractor according to a contract: (i) *start:* to perform, i.e., to start, a bus ride (obligated); (ii) *cancel:* to cancel a bus ride (allowed, with restrictions); (iii) *insert:* to insert a bus ride; and (iv) *subcontract:* to sub-contract part or all of a contract.

K.8.2 The Final Narrative slide 1063

We leave, as an exercise, the expression of a complete narrative.

Instead we proceed directly to a formalisation.

K.8.3 A Formal Syntax

We treat separately, the syntax of contracts (for a schematised example see Page 261) and the syntax of the actions implied by contracts (for schematised examples see Page 262).

Contracts slide 1064

An example contract can be 'schematised':

[13] Special events: breakdown (that is, cancellations) of other bus rides, sports event (soccer matches), etc.

```
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 contracting) takes place and in which the implicit net is defined. slide 1065

236. contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.
237. 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.
238. A contract body stipulates a timetable and the set of operations that are mandated or allowed by the contractor.
239. 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 1066

type
236. CId, CNm
237. Contract = CId $\times$ CNm $\times$ CNm $\times$ Body
238. Body = Op-**set** $\times$ TT
239. Op == ″start″ | ″cancel″ | ″insert″ | ″subcontract″

An abstract example contract:
(cid,cnm$_i$,cnm$_j$,({″start″,″cancel″,″insert″,″sublicense″},tt))

Actions

slide 1067

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 261) 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 1068

All actions are 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 261. slide 1069

```
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 242
```

slide 1070

Actions

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.

K.8.4 Uniqueness and Traceability of Contract Identifications slide 1071

240. There is a "root" contract name, rcid.
241. There is a "root" contractor name, rcnm.

```
value
240  rcid:CId
241  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 1072

242. 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.
243. 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
242  gen_CId: CId × CNm × Time → CId
```

```
243 obs_CId: CId $\tilde{\rightarrow}$ CIdL [ pre obs_CId(cid):cid≠rcid ]
243 obs_CNm: CId $\tilde{\rightarrow}$ CNm [ pre obs_CNm(cid):cid≠rcid ]
243 obs_Time: CId $\tilde{\rightarrow}$ Time [ pre obs_Time(cid):cid≠rcid ]
```

slide 1073

244. All contract names are unique.

```
axiom
244 ∀ cid,cid′:CId•cid≠cid′⇒
244   obs_CId(cid)≠obs_CId(cid′) ∨ obs_CNm(cid)≠obs_CNm(cid′)
244   ∨ obs_LicNm(cid)=obs_CId(cid′)∧obs_CNm(cid)=obs_CNm(cid′)
244     ⇒ obs_Time(cid)≠obs_Time(cid′)
```

slide 1074

245. 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
245 contract_trace: CId → LCIdCNmTTrace
245 contract_trace(cid) ≡
245   case cid of
245     rcid → ⟨⟩,
245     _ → contract_trace(obs_LicNm(cid))^⟨obs_TraceTriple(cid)⟩
245   end

245 obs_TraceTriple: CId → TraceTriple
245 obs_TraceTriple(cid) ≡
245   mkTrTr(obs_CId(cid),obs_CNm(cid),obs_Time(cid))
```

The trace is generated in the chronological order: most recent contract name generation times last.

slide 1075

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))
```

K.8.5 Execution State

slide 1076

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 1077

Global State

The net is part of the global state (and of bus traffics). We consider just the bus traffic.

type
167. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI) 234

230. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos))) 242
231. BPos = atHub | onLnk | atBS
232. atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
233. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
234. 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.

Local Sub-contractor Contract States

slide 1078

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.

Local Sub-contractor Bus States

slide 1079

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.[14]

slide 1080

type
```
    BusNo
    Bus$\Sigma$ = FreeBuses$\Sigma$ $\times$ ActvBuses$\Sigma$ $\times$ BusHists$\Sigma$
    FreeBuses$\Sigma$ = BusStop $\overrightarrow{m}$ BusNo-set
    ActvBuses$\Sigma$ = BusNo $\overrightarrow{m}$ BusInfo
    BusInfo = BLId$\times$BId$\times$LicNm$\times$LeeNm$\times$BusTrace
    BusHists$\Sigma$ = Bno $\overrightarrow{m}$ BusInfo$^*$
    BusTrace = (Time$\times$BusStop)$^*$
    LorBus$\Sigma$ = LeeNm $\overrightarrow{m}$ (LicNm $\overrightarrow{m}$ ((BLId$\times$BId) $\overrightarrow{m}$ (BNo$\times$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 1081

value
```
    update_Bus$\Sigma$: Bno$\times$(T$\times$BusStop) $\to$ ActBus$\Sigma$ $\to$ ActBus$\Sigma$
    update_Bus$\Sigma$(bno,(t,bs))(act$\sigma$) $\equiv$
        let (blid,bid,licn,leen,trace) = act$\sigma$(bno) in
        act$\sigma\dagger$[ bno$\mapsto$(licn,leen,blid,bid,trace$^\frown\langle$(t,bs)$\rangle$) ] end
        pre bno $\in$ dom act$\sigma$
```

slide 1082

[14] 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 1083

```
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 1084

```
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 ...]
```

Constant State Values

slide 1085

There are a number of constant values, of various types, which characterise the "business of contract holders". We define some of these now.

246. 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.
247. 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.
248. There is an initial bus traffic, tr.
249. There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.
250. To maintain the bus traffic "spelled out", in total, by timetable tt one needs a number of buses.

251. 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.
252. We leave it to the reader to define a function which ascertain the minimum number of buses needed to implement traffic tr.

slide 1086

value
246. net : N,
247. leens : LeeNm-**set**,
248. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
249. t_0 : T • $t_0 \geq$ **min dom** tr,
250. min_no_of_buses : **Nat** • necessary_no_of_buses(itt),
251. busnos : BusNo-**set** • **card** busnos $\geq$ min_no_of_buses

252. necessary_no_of_buses: TT $\rightarrow$ **Nat**

slide 1087

253. 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]).
254. The initial, i.e., the distinguished, contract has a name, root_licn.
255. The initial contract can only perform the `"sublicense"` operation.
256. The initial contract has a timetable, tt.
257. The initial contract can thus be made up from the above.

slide 1088

value
253. root_leen,init_ln : LeeNm • root_leen $\notin$ leens $\wedge$ initi_leen $\in$ leens,
254. root_licn : LicNm
255. iops : Op-**set** = {″`sublicense`″},
256. itt : TT,
257. init_lic:License = (root_licn,root_leen,(iops,itt),init_leen)

Initial Sub-contractor Contract States

slide 1089

type
  InitLicΣs = LeeNm $\overrightarrow{m}$ LicΣ
value
  ilσ:LicΣ=([init_leen $\mapsto$ [root_leen $\mapsto$ [iln $\mapsto$ init_lic]]]
      $\cup$ [leen $\mapsto$ [ ] | leen:LeeNm • leen $\in$ leenms$\setminus${init_leen}],[ ],[ ])

Initial Sub-contractor Bus States

slide 1090

258. Initially each sub-contractor possesses a number of buses.
259. No two sub-contractors share buses.
260. We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.
261. We do not prescribe a "satisfiable and practical" such initial assignment (ibσs).
262. But we can constrain ibσs.
263. The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.
264. Active bus states must be empty.
265. No two free bus states must share buses.
266. All bus histories are void.

slide 1091

```
type
258. AllBuses′ = LeeNm →m BusNo-set
259. AllBuses = {|ab:AllBuses′•∀ {bs,bs′}⊆rng ab∧bns≠bns′⇒bns ∩ bns′={}|}
260. InitBusΣs = LeeNm →m BusΣ
value
259. allbuses:Allbuses • dom allbuses = leenms ∪ {root_leen} ∧ ∪ rng allbuses = busnos

260. ibσs:InitBusΣs
261. wf_InitBusΣs: InitBusΣs → Bool
262. wf_InitBusΣs(iσs) ≡
263.   dom iσs = leenms ∧
264.   ∀ (_,abσ,_):BusΣ•(_,abσ,_) ∈ rng iσs ⇒ abσ=[ ] ∧
265.   ∀ (fbiσ,abiσ),(fbjσ,abjσ):BusΣ •
265.     {(fbiσ,abiσ),(fbjσ,abjσ)}⊆rng iσs
265.       ⇒ (fbiσ,actiσ)≠(fbjσ,actjσ)
265.         ⇒ rng fbiσ ∩ rng fbjσ = {}
266.           ∧ actiσ=[ ]=actjσ
```

Communication Channels

slide 1092

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. slide 1093

Sub-Contractor↔Sub-Contractor Channels Consider each sub-contractor (same as contractor) to be modelled as a behaviour. Each sub-contractor (licensor) behaviour has a unique name, the LeeNm. Each sub-contractor can potentially communicate with every other sub-contractor. We model each such communication potential by a channel. For n sub-contractors there are thus $n \times (n-1)$ channels.

channel { l_to_l[fi,ti] | fi:LeeNm,ti:LeeNm • {fi,ti}⊆leens ∧ fi≠ti } LLMSG
type LLMSG = ...

We explain the declaration: **channel** { l_to_l[fi,ti] | fi:LeeNm, ti:LeeNm • fi≠ti } LLMSG. It prescribes $n \times (n-1)$ channels (where n is the cardinality of the sub-contractor name sets). Each channel is prescribed to be capable of communicating messages of type MSG. The square brackets [...] defines l_to_l (sub-contractor-to-sub-contractor) as an array.

We shall later detail the BusRideNote, CancelNote, InsertNote and FundXfer message types. slide 1094

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 = ...

slide 1095

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 = ...

slide 1096

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)

slide 1097

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 slide 1098

So we shall be modelling the transport contract domain as follows: As for behaviours we have this to say. There will be n sub-contractors. One sub-contractor will be initialised to one given license. You may think of this sub-contractor being the transport authority. Each sub-contractor is modelled, in RSL, as a CSP-like process. With each sub-contractor, l_i, there will be a number, b_i, of buses. That number may vary from sub-contractor to sub-contractor. There will be b_i channels of communication between a sub-contractor and that sub-contractor's buses, for each sub-contractor. There is one global process, the traffic. There is one channel of communication between a sub-contractor and the traffic. Thus there are n such channels. slide 1099

As for operations, including behaviour interactions we assume the following. All operations of all processes are to be thought of as instantaneous, that is, taking nil time ! Most such operations are the result of channel communications either just one-way notifications, or inquiry requests. Both the former (the one-way notifications) and the latter (inquiry requests) must not be indefinitely barred from receipt, otherwise holding up the notifier. The latter (inquiry requests) should lead to rather immediate responses, thus must not lead to dead-locks.

The System Behaviour **slide 1100**

The system behaviour starts by establishing a number of licenseholder and bus_ride behaviours and the single time_clock and bus_traffic behaviours slide 1101

```
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 1102

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.

K.8.6 Semantic Elaboration Functions **slide 1103**

The Licenseholder Behaviour

267. The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.

268. It internally non-deterministically ($\lceil\rceil$) 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.

slide 1104

```
267. licenseholder: LeeNm → (LicΣ×BusΣ) → Unit
268. licenseholder(leen)(licσ,busσ) ≡
268.     licenseholder(leen)((lic_ops⌈⌉bus_mon⌈⌉neg_licenses)(leen)(licσ,busσ))
```

The Bus Behaviour

slide 1105

269. Buses ply the network following a timed bus route description.
     A timed bus route description is a list of timed bus stop visits.
270. A timed bus stop visit is a pair: a time and a bus stop.
271. 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.
272. Bus behaviours start with a "nil" bus route description.

slide 1106

```
type
269. TBR = TBSV*
270. TBSV = Time × BusStop
value
271. conTBR: BusRoute × BusSched → TBR
271. conTBR((dt,til,at),(bs1,bsl,bsn)) ≡
271(a))   ⟨(dt,bs1)⟩
271(b))   ⌢ ⟨(til[i],bsl[i])|i:Nat•i:⟨1..len til⟩⟩
271(c))   ⌢ ⟨(at,bsn)⟩
             pre: len til = len bsl
type
272. BRD == "nil" | TBR
```

slide 1107

273. The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,
274. The bus repeatedly observes the time, `t`, and its position, `po`, in the traffic.
275. There are now four case distinctions to be made.

276. If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.
277. 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.
278. If the bus is 'en route', at a bus stop, then it so informs its owner and continues the journey.
279. In all other cases the bus continues its journey

slide 1108

```
value
273. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
273.     in,out l_to_b[leen,bno], in,out b_to_tr[bno], in b_to_t[bno] Unit
273. bus_ride(leen,bno)(licn,brd) ≡
274.   let t = b_to_t[bno]? in
274.   let (bus,pos) = (b_to_tr[bno]!reqBusAndPos(bno,t) ; b_to_tr[bno]?) in
275.   case (brd,pos) of
276.     (″nil″,mkAtBS(_,_,_,_)) →
276.          let (licn,brd′) = (l_to_b[leen,bno]!reqBusRid(pos);l_to_b[leen,bno]?) in
276.          bus_ride(leen,bno)(licn,brd′) end
277.     (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
277s          l_to_b[l,b]!BusΣMsg(t,pos);
277          l_to_b[l,b]!BusHistΣMsg(licn,bno);
277          l_to_b[l,b]!FreeΣ_ActΣMsg(licn,bno) ;
277          bus_ride(leen,bno)(ilicn,″nil″),
278.     (⟨(t,pos),(t′,bs′)⟩⌢brd′,mkAtBS(_,_,_,_)) →
278s          l_to_b[l,b]!BusΣMsg(t,pos) ;
278          bus_ride(licn,bno)(⟨(t′,bs′)⟩⌢brd′),
279.     _ → bus_ride(leen,bno)(licn,brd) end end end
```

slide 1109

In formula line 274 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 277s and 278s:

```
Simple:  277s  l_to_b[l,b]!BusΣMsg(pos);
Revised: 277r  l_to_b[l,b]!BusΣMsg(pos,bus_info(bus));

Simple:  278s  l_to_b[l,b]!BusΣMsg(pos);
Revised: 278r  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.
-
-
-

The Global Time Behaviour

slide 1110

280. The time_clock is a never ending behaviour — started at some time t_0.
281. The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.
282. At any moment the time_clock behaviour may not be inquired.
283. 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
280. time_clock: T →
280.     in,out {l_to_t[ leen ] | leen:LeeNm • leen ∈ leenms}
280.     in,out {b_to_t[ bno ] | bno:BusNo • bno ∈ busnos}  Unit
280.  time_clock:(t) ≡
282.    (skip ⊓
281.    (⊓{l_to_t[ leen ]? ; l_to_t[ leen ]!t | leen:LeeNm•leen ∈ leens})
281.    ⊓ (⊓{b_to_t[ bno ]? ; b_to_t[ bno ]!t | bno:BusNo•bno ∈ busnos})) ;
283.    (time_clock:(t) ⊓ time_clock(t+δ_t))
```

The Bus Traffic Behaviour

slide 1111

284. There is a single bus_traffic behaviour. It is, "mysteriously", given a constant argument, "the" traffic, tr.
285. 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.
286. The request for a bus position comes from some bus.
287. The bus positions are part of the traffic at time t.
288. The bus_traffic behaviour, after informing of a bus position reverts to "itself".

```
value
284.  bus_traffic: TR → in,out {b_to_tr[ bno ]|bno:BusNo•bno ∈ busnos} Unit
284.  bus_traffic(tr) ≡
286.    ⌷ { let reqBusAndPos(bno,time) = b_to_tr[ b ]? in assert b=bno
285.         if time ∉ dom tr then chaos else
287.         let (_,bps) = tr(t) in
285.         if bno ∉ dom tr(t) then chaos else
285.         b_to_tr[ bno ]!bps(bno) end end end end | b:BusNo•b ∈ busnos} ;
288.     bus_traffic(tr)
```

License Operations

slide 1112

289. 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, Sect. K.8.3 (Page 262).
290. To perform any licensed operation the sub-contractor needs to know the time and
291. 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.
292. Thus there is a case distinction amongst four alternatives.
293. This case distinction is expressed in the four lines identified by: 293.
294. All the auxiliary functions, besides the action arguments, require the same state arguments.

slide 1113

```
value
289. lic_ops: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
289. lic_ops(leen)(licσ,busσ) ≡
290.   let t = (time_channel(leen)!req_Time;time_channel(leen)?) in
291.   let (licn,act) = choice(licσ)(busσ)(t) in
292.   (case act of
293.      mkCon(blid,bid)  → cndct(licn,leenm,t,act),
293.      mkCan(blid,bid)  → cancl(licn,leenm,t,act),
293.      mkIns(blid,bid)  → insrt(licn,leenm,t,act),
293.      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)
```

Bus Monitoring

slide 1114

Like for the **bus_ride** behaviour we decompose the **bus_mon**itoring behaviour into two behaviours. The **local_bus_mon**itoring behaviour monitors the buses that are commissioned by the sub-contractor. The **licensor_bus_mon**itoring behaviour monitors the buses that are commissioned by sub-contractors sub-contractd by the contractor.

```
value
   bus_mon: l:LeeNm → (LicΣ×BusΣ)
              → in {l_to_b[l,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
   bus_mon(l)(licσ,busσ) ≡
      local_bus_mon(l)(licσ,busσ) ⌈⌉ licensor_bus_mon(l)(licσ,busσ)
```

slide 1115

295. The **local_bus_mon**itoring function models all the interaction between a contract holder and its despatched buses.
296. We show only the communications from buses to contract holders.
297.
298.
299.
300.
301.
302.
303.
304.
305.

slide 1116

```
295.  local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
296.         → in {l_to_b[leen,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
295.  local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
297.     let (bno,msg) = ⌈⌉{(b,l_to_b[l,b]?)|b:BNo•b ∈ allbuses(leen)} in
301.     let (blid,bid,licn,lorn,trace) = abσ(bno) in
298.     case msg of
299.       BusΣMsg(t,bs) →
303.         let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
303.         (licσ,(fbσ,abσ′,histσ)) end,
305.       BusHistΣMsg(licn,bno) →
305.         let lbσ′ =
305.            update_LorBusΣ(obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))(lbσ)   in
305.         l_to_l[leen,obs_LorNm(licn)]!Licensor_BusHistΣMsg(licn,leen,bno,blid,bid,tr);
305.         ((rlσ,slσ,lbσ′),busσ) end
304.       FreeΣ_ActΣMsg(licn,bno) →
305.         let (fbσ′,abσ′) = update_FreeΣ_ActΣ(bno,bs)(fbσ,abσ) in
305.         (licσ,(fbσ′,abσ′)) end
305.     end end end
```

slide 1117

306.
307.

308.
309.
310.

slide 1118

```
306. licensor_bus_mon: lorn:LorNm → (LicΣ×BusΣ)
306.      → in {l_to_l[lorn,leen]|leen:LeeNm•leen ∈ leenms\{lorn}} (LicΣ×BusΣ)
306. licensor_bus_mon(lorn)(licσ,busσ) ≡
306.    let (rlσ,slσ,lbhσ) = licσ in
306.    let (leen,Licensor_BusHistΣMsg(licn,leen″,bno,blid,bid,tr))
                              = ⌈⌉{(leen′,l_to_l[lorn,leen′]?)|leen′:LeeNm•leen′ ∈ leenms\{lorn}} in
306.    let lbhσ′ =
306.       update_BusHistΣ(obs_LorNm(licn),licn,leen″,(blid,bid),(bno,trace))(lbhσ) in
306.    l_to_l[leenm,obs_LorNm(licnm)]!Licensor_BusHistΣMsg(b,blid,bid,lin,lee,tr);
306.    ((rlσ,slσ,lbhσ′),busσ)
306.    end end end
```

License Negotiation

slide 1119

311.
312.
313.
314.
315.
316.
317.
318.
319.
320.
321.
322.

slide 1120

311.
312.
313.
314.
315.
316.
317.
318.
319.
320.
321.

The Conduct Bus Ride Action

slide 1121

323. 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 1122

323.
323(a))
323(b))
323(c))
323(d))
323(e))
323(f))
323(g))

The Cancel Bus Ride Action

slide 1123

324. 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 323(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 1124

324.
324(a))
324(b))
324(c))

The Insert Bus Ride Action

slide 1125

325. 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 323(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 323(d))–323(g)).

slide 1126

325.
325(a))
325(b))
325(c))
325(d))

The Contracting Action

slide 1127

326. 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 1128

326.
326(a))
326(b))
326(c))
326(d))
326(e))
326(f))

K.8.7 Discussion

slide 1129

slide 1130

L

Human Behaviour

slide 1132

Chapter 15 (Pages 119–120) complements the present appendix. slide 1133

M

From Domains to Requirements slide 1135

Chapter 20 (Pages 135–137) complements the present appendix.

M.1 Shared Phenomena and Concepts slide 1136

We (backward) refer to methodology Sect. 20.3 on page 135. It provides the methodological background for the present section.

M.2 Domain Requirements slide 1137

We (backward) refer to methodology Sect. 20.4 on page 136. It provides the methodological background for the present section.

M.3 Interface Requirements slide 1138

We (backward) refer to methodology Sect. 20.5 on page 136. It provides the methodological background for the present section.

M.4 Machine Requirements slide 1139

We (backward) refer to methodology Sect. 20.6 on page 136. It provides the methodological background for the present section.

M.5 Discussion slide 1140

slide 1141

N

Wrapping it All Up !

slide 1143

slide 1144

Part VIII

THE SPECIFICATION LANGUAGES

O

An RSL Primer slide 1146

This is an ultra-short introduction to the `RAISE` Specification Language, `RSL`.

O.1 Types and Values slide 1147

Simplifying we consider a type to be a class (a possibly infinite set) of values, i.e., a set characterised by some unifying properties. Values are then instances of "things" that satisfy such properties. Examples of values are the numbers denoted by the numerals: 0, 1, 2, . . . , etc.; the numbers denoted by the numerals: . . . , -2, -1, 0, 1, 2, . . . , etc.; and the numbers denoted by the numerals: . . . , -5.43, -1.0, 0.0, 1.23· · ·, 2,71828183· · ·, 3,14159265· · ·, 4.56 . . . , etc. We shall refer to the types of these three example sets by the type names **Nat**, **Int**, **Real**. As we shall soon see, there are an infinity of types.

O.1.1 Some Distinctions slide 1148

We distinguish between discrete and continuous types and hence between discrete and continuous values. By a discrete value we mean a value which is either atomic discrete or composite discrete: an atomic discrete value is a value which we are not interested in decomposing into components as it makes no sense for us to do so; a composite discrete value is a value which can be decomposed into (one or more) components which are all discrete values. slide 1149

By a continuous value we mean a value which is either atomic continuous or composite continuous: an atomic continuous value is a value which we are not interested in decomposing into components as it makes no sense for us to do so and where the values of the type lie in some dense point space; a composite continuous value is a value which can be decomposed into meaningful (one or more) components some of which (one or more) are continuous values. slide 1150

Oil Industry Example 1: Atomic and Composite Types and Values

A pipe is a composite continuous value which consists ofa pipe structure and some oil; the former being of discrete atomic value while the latter is a continuous atomic value: *"from no oil, via a tiny drop of oil, and more, say a gallon and a third of oil, to several million barrels of oil"*. Similarly for pumps, valves, depots, etc.: all are composite continuous values consisting of corresponding discrete atomic structures and continuous atomic valued (amounts of) oil.

O.1.2 An Aside

slide 1151

You might mistakenly think that continuous atomic values are composed from "subsegments" or "subspaces" or "subvolumes", etc., of continuous atomic values. Consider the following examples: (i) *Crude oil:* one can decompose crude oil into a very large number of molecules (of different hydrocarbons); the most commonly found molecules are alkanes (linear or branched), cycloalkanes, aromatic hydrocarbons, or more complicated chemicals like asphaltenes; but it is not a decomposition of a liter of crude oil to divide it up into ten deciliters. But if our choice of abstraction ignores the molecular structure of oil, then oil has an atomic, continuous value. slide 1152 (ii) *Time:* one can decompose time into years, months, weeks, days, hours, minutes and seconds; but if our choice of abstraction ignores these units, and just considers the time axis to be a linearly ordered dense set of points, then time has an atomic, continuous value. Thus we must consider the operations to be (i.e., that we wish) performed on what may appear as continuous values in order to determine our abstraction: either as atomic continuous values or as composite continuous values. slide 1153 Operations on oil divide a volume of oil into a set of two or more volumes of oil, displaces a volume of oil from one location into a (usually neighbouring) locations, etc. Operations on time calculates the time interval between two time points, adds an interval of time to time to obtain another time, etc. But in all these examples oil, respectively time remain atomic, continuous values.

O.2 Types

slide 1154

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

O.2.1 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 1155

type
[1] **Bool** **true**, **false**
[2] **Int** ... , −2, −2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., −5.43, −1.0, 0.0, 1.23· · ·, 2,7182· · ·, 3,1415· · ·, 4.56, ...
[5] **Char** "a", "b", ..., "0", ...
[6] **Text** "abracadabra"

slide 1156

Oil Industry Example 2: Atomic Discrete Types

type
DrainPumpStruct, PipeStruct, ValveStruct, FlowPumpStruct,
FillPumpStruct, DepotStruct, SwitchStruct, ...

Oil Industry Example 3: Atomic Continuous Types

type
Time
Oil, Gasoline, Gas, Ethanol, ...

Composite Types slide 1157

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully "taken apart". There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types *slide 1158*

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

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

[7] A-**set**
[8] A-**infset**
[9] $A \times B \times ... \times C$
[10] A^*
[11] A^ω
[12] $A \underset{m}{\overrightarrow{}} B$
[13] $A \to B$
[14] $A \overset{\sim}{\to} B$
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., −2, −1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period (“.”), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian $B \times C \times ... \times D$, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., $(A \underset{m}{\overrightarrow{}} B)$, or (A^*)-**set**, or (A-**set**)list, or $(A|B) \underset{m}{\overrightarrow{}} (C|D|(E \underset{m}{\overrightarrow{}} F))$, 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 1159

Sorts and Observer Functions

type
    A, B, C, ..., D
value
    obs_B: A $\rightarrow$ B, obs_C: A $\rightarrow$ C, ..., obs_D: A $\rightarrow$ D

The above expresses that values of type A are composed from at least three values — and these are of type B, C, ..., and D. A concrete type definition corresponding to the above presupposing material of the next section

type
    B, C, ..., D
    A = B $\times$ C $\times$ ... $\times$ D

O.2.2 Type Definitions

slide 1160

Concrete Types

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

type
    A = Type_expr

Some schematic type definitions are:

```
[1] Type_name = Type_expr /* without |s or subtypes */
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
              mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
              ... |
              mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

slide 1161

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

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

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

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

Subtypes **slide 1162**

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

```
type
  A = {| b:B • 𝒫(b) |}
```

Sorts — Abstract Types **slide 1163**

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

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

O.3 The RSL Predicate Calculus **slide 1164**

O.3.1 Propositional Expressions

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

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

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

O.3.2 Simple Predicate Expressions slide 1165

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

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

are simple predicate expressions.

O.3.3 Quantified Expressions slide 1166

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

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

are quantified expressions — also being predicate expressions.

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

O.4 Concrete RSL Types: Values and Operations slide 1167

O.4.1 Arithmetic

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

O.4.2 Set Expressions slide 1168

Set Enumerations

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

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

Set Comprehension slide 1169

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

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

O.4.3 Cartesian Expressions slide 1170

Cartesian Enumerations

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

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

O.4.4 List Expressions slide 1171

List Enumerations

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

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

$\langle$ 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.

List Comprehension slide 1172

The last line below expresses list comprehension.

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

O.4.5 Map Expressions slide 1173

Map Enumerations

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

type
    T1, T2
    M = T1 $\underset{m}{\overrightarrow{\ \ }}$ T2
value
    u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
    [ ], [u$\mapsto$v], ..., [u1$\mapsto$v1,u2$\mapsto$v2,...,un$\mapsto$vn] $\forall \in$ M

Map Comprehension slide 1174

The last line below expresses map comprehension:

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

O.4.6 Set Operations slide 1175

Set Operator Signatures

value
  19 $\in$: A $\times$ A-**infset** $\rightarrow$ **Bool**
  20 $\notin$: A $\times$ A-**infset** $\rightarrow$ **Bool**
  21 $\cup$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
  22 $\cup$: (A-**infset**)-**infset** $\rightarrow$ A-**infset**
  23 $\cap$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
  24 $\cap$: (A-**infset**)-**infset** $\rightarrow$ A-**infset**
  25 $\setminus$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
  26 $\subset$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
  27 $\subseteq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
  28 $=$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
  29 $\neq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
  30 **card**: A-**infset** $\overset{\sim}{\rightarrow}$ **Nat**

Set Examples slide 1176

examples
  $a \in \{a,b,c\}$
  $a \notin \{\}$, $a \notin \{b,c\}$
  $\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
  $\cup\{\{a\},\{a,b\},\{a,d\}\} = \{a,b,d\}$
  $\{a,b,c\} \cap \{c,d,e\} = \{c\}$
  $\cap\{\{a\},\{a,b\},\{a,d\}\} = \{a\}$

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

Informal Explication

slide 1177

19. $\in$: The membership operator expresses that an element is a member of a set.
20. $\notin$: The nonmembership operator expresses that an element is not a member of a set.
21. $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22. $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23. $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24. $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets. slide 1178
25. $\setminus$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28. $=$: The equal operator expresses that the two operand sets are identical.
29. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

slide 1179

The operations can be defined as follows ($\equiv$ is the definition symbol):

value
$s' \cup s'' \equiv \{ a \mid a{:}A \bullet a \in s' \vee a \in s'' \}$
$s' \cap s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \in s'' \}$
$s' \setminus s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \notin s'' \}$

```
    s′ ⊆ 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 ∗/
```

O.4.7 Cartesian Operations

slide 1180

```
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
```

O.4.8 List Operations

slide 1181

List Operator Signatures

```
value
    hd: A^ω ~→ A
    tl: A^ω ~→ A^ω
    len: A^ω ~→ Nat
    inds: A^ω → Nat-infset
    elems: A^ω → A-infset
    .(.): A^ω × Nat ~→ A
    ⌢: A^∗ × A^ω → A^ω
    =: A^ω × A^ω → Bool
    ≠: A^ω × A^ω → Bool
```

List Operation Examples slide 1182

```
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 slide 1183

- **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. slide 1184
- ⌢: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions slide 1185

```
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 }
```

slide 1186

```
    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″)
```

O.4.9 Map Operations

slide 1187

Map Operator Signatures and Map Operation Examples

```
value
    m(a): M → A ~→ B, m(a) = b

    dom: M → A-infset [domain of map]
        dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

    rng: M → B-infset [range of map]
        rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

    †: M × M → M [override extension]
        [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]
```

slide 1188

```
    ∪: M × M → M [merge ∪]
        [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]
```

$\backslash$: 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′]

Map Operation Explication

slide 1189

- $m(a)$: 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 1190
- \: 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.

Map Operation Redefinitions

slide 1191

The map operations can also be defined as follows:

```
value
   rng m ≡ { m(a) | a:A • a ∈ dom m }

   m1 † m2 ≡
      [ a↦b | a:A,b:B •
         a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

   m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
                  a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

   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
```

O.5 λ-Calculus + Functions slide 1192

O.5.1 The λ-Calculus Syntax

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

O.5.2 Free and Bound Variables slide 1193

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

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

O.5.3 Substitution slide 1194

In RSL, the following rules for substitution apply:

- **subst**([N/x]x) $\equiv$ N;
- **subst**([N/x]a) $\equiv$ a,
  for all variables a$\neq$ x;
- **subst**([N/x](P Q)) $\equiv$ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) $\equiv$ λ y•P;
- **subst**([N/x](λ y•P)) $\equiv$ λy• **subst**([N/x]P),
  if x$\neq$y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) $\equiv$ λz•**subst**([N/z]**subst**([z/y]P)),
  if y$\neq$x and y is free in N and x is free in P
  (where z is not free in (N P)).

O.5.4 α-Renaming and β-Reduction slide 1195

- α-renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) $\equiv$ **subst**([N/x]M)

O.5.5 Function Signatures slide 1196

For sorts we may want to postulate some functions:

type
  A, B, C
value
  obs_B: A $\to$ B,
  obs_C: A $\to$ C,
  gen_A: B$\times$C $\to$ A

O.5.6 Function Definitions slide 1197

Functions can be defined explicitly:

value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments $\stackrel{\sim}{\rightarrow}$ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)

slide 1198

Or functions can be defined implicitly:

value
  f: Arguments → Result
  f(args) **as** result
  post P1(args,result)

  g: Arguments $\stackrel{\sim}{\rightarrow}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

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

O.6 Other Applicative Expressions

slide 1199

O.6.1 Simple let Expressions

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

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

is an “expanded” form of:

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

O.6.2 Recursive let Expressions

slide 1200

Recursive **let** expressions are written as:

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)

O.6.3 Predicative let Expressions

slide 1201

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).

O.6.4 Pattern and "Wild Card" let Expressions

slide 1202

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let $\langle$a$\rangle^\frown\ell$ = list **in** ... **end**
let $\langle$a,_,b$\rangle^\frown\ell$ = list **in** ... **end**

let [a$\mapsto$b] ∪ m = map **in** ... **end**
let [a$\mapsto$b,_] ∪ m = map **in** ... **end**

O.6.5 Conditionals

slide 1203

Various kinds of conditional expressions are offered by `RSL`:

```
if b_expr then c_expr else a_expr
end

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

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

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

O.6.6 Operator/Operand Expressions slide 1204

```
⟨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⟩ ::= !
```

O.7 Imperative Constructs slide 1205

O.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

```
   Unit
value
   stmt: Unit → Unit
   stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

O.7.2 Variables and Assignment

slide 1206

```
0. variable v:Type := expression
1. v := expr
```

O.7.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

```
2. skip
3. stm_1;stm_2;...;stm_n
```

O.7.4 Imperative Conditionals

```
4. if expr then stm_c else stm_a end
5. case e of: p_1→S_1(p_1),...,p_n→S_n(p_n) end
```

O.7.5 Iterative Conditionals

slide 1207

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

O.7.6 Iterative Sequencing

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

O.8 Process Constructs slide 1208

O.8.1 Process Channels

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

```
channel c:A
channel { k[i]:B • i:KIdx }
```

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

O.8.2 Process Composition slide 1209

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

P ‖ Q	Parallel composition
P ⌷ Q	Nondeterministic external choice (either/or)
P ⌈⌉ Q	Nondeterministic internal choice (either/or)
P ⫲ Q	Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external (⌷) or internal (⌈⌉). The interlock (⫲) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

O.8.3 Input/Output Events slide 1210

Let c, k[i] and e designate channels of type A and B, then:

c ?, k[i] ?	Input
c ! e, k[i] ! e	Output

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

O.8.4 Process Definitions slide 1211

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

```
value
    P: Unit → in c out k[i]
    Unit
    Q: i:KIdx →  out c in k[i] Unit

    P() ≡ ... c ? ... k[i] ! e ...
    Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

O.9 Simple RSL Specifications slide 1212

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

```
    type
        ...
    variable
        ...
    channel
        ...
    value
        ...
    axiom
        ...
```

slide 1213

P

Petri Nets

slide 1214

Q

MSC: Message Sequence Charts

R

DC: Duration Caluculus

Part IX

SOLUTIONS

S

Solution to Exercises

S.1 Some Development Paradigms

S.1.1 What is a Domain?

The below suggests an answer to Exercise 1.10.1 on page 33.

S.1.2 Are These Domains?

The below suggests an answer to Exercise 1.10.2 on page 34.

0.1. Programming: Yes. Has simple entities (programmers, programs, compilers, etc.), has operations (writing a program, program syntax and type checking, compiling, debugging a program, etc.), events (program compilation ended successefully, or did not end successfully, etc.), has behaviours (program writing followed by syntax and type checking, compiling, debugging, etc.).
0.2. Compilers: No. A compiler is a simple entity, but it is a inert piece of software; it itself has not operations, no events, no behaviour; but, as a simple entity, it enters into operations ("outside" itself), is implicitly the cause of events and, as a simple entity, enters into behaviours.
0.3. Compiler Writing: Yes. For same reason as 'Programming'.
0.4. Patient Hospitalisation: Yes. Has simple entities (hospitals, wards, beds, medicine, patient medical records, etc.), has operations (annamnese, analysis, diagnoses, treatment planning, treatment, etc.), events (blood pressure goes above acceptable level, blood sugar content goes below acceptable level, enters or leaves coma, death, patient is declared "fresh" (can leave hospital), etc.), behaviours (patients, medical staff, patient hospitalisations, etc.).

S.1.3 The Triptych Paradigm

The below suggests an answer to Exercise 1.10.3 on page 34.

S.1.4 The Three Phases of Software Development

The below suggests an answer to Exercise 1.10.4 on page 34.

S.1.5 Domain Engineering

The below suggests an answer to Exercise 1.10.5 on page 34.

S.1.6 Requirements Engineering

The below suggests an answer to Exercise 1.10.6 on page 34.

S.1.7 Software Design

The below suggests an answer to Exercise 1.10.7 on page 34.

S.1.8 What is a Model

The below suggests an answer to Exercise 1.10.8 on page 34.

S.1.9 Phase of Development

The below suggests an answer to Exercise 1.10.9 on page 34.

S.1.10 Stage of Development

The below suggests an answer to Exercise 1.10.10 on page 34.

S.1.11 Step of Development

The below suggests an answer to Exercise 1.10.11 on page 34.

S.1.12 Development Documents

The below suggests an answer to Exercise 1.10.12 on page 35.

S.1.13 Descriptions, Prescriptions and Specifications

The below suggests an answer to Exercise 1.10.13 on page 35.

S.1.14 Software

The below suggests an answer to Exercise 1.10.14 on page 35.

S.1.15 Informal and Formal Software Development

The below suggests an answer to Exercise 1.10.15 on page 35.

S.1.16 Specification Ontology

The below suggests an answer to Exercise 1.10.16 on page 35.

The four kinds of phenomena and concepts around which our informal (i.e., narrative) and formal descriptions, prescriptions and specifications evolve are:

- simple entities,
- operation (i.e., functions),
- events and
- behaviours.

S.1.17 Discreteness

The below suggests an answer to Exercise 1.10.17 on page 35.

An entity is discrete if it is timewise fixed, i.e., does not change structure with time but could change value of possible sub-entities or of atttributes.

S.1.18 Continuous

The below suggests an answer to Exercise 1.10.18 on page 35.

An entity is continuous if it is timewise variable, i.e., changes structure with time, or if any subpart of it is also an entity of the same structure.

S.1.19 Discrete and Continuous Entities

The below suggests an answer to Exercise 1.10.19 on page 35.

0.1.
0.2.
0.3.
0.4.

S.1.20 Operations on Time and Time Intervals

The below suggests an answer to Exercise 1.10.20 on page 36.

```
**type**
    T, TI [ We exclude the possibility of negative times and time intervals ]
**value**
    $t_0$:T
**axiom**
    ∀ t:T • t≥$t_0$
**value**
    convert_t_to_ti: T → TI
    convert_t_to_ti(t) ≡ t − $t_0$
    +: ((T×TI) → T) | ((TI×TI) → TI)
    −: (((T×T)(TI×TI)) $\overset{\sim}{\rightarrow}$ TI) | ((T×TI) $\overset{\sim}{\rightarrow}$ T)
        **pre** −(t,t′): t≥t′, −(ti,ti′): ti≥ti′, −(t,ti′): convert_t_to_ti(t)≥ti
    /: (TI×TI) → **Real**
    ∗: (TI×**Real**) → TI
    <,≤,=,≥,>: ((T×T)|(TI×TI)) → **Bool**
```

t_0 models an origin of time.

S.1.21 Operations on Oil and Gas

The below suggests an answer to Exercise 1.10.21 on page 36.

```
**type**
    Oil, Gas
**value**
    obs_Amount: (Oil|Gas) → **mol**
    +: ((Oil×Oil)→Oil) | ((Gas×Gas)→Gas)
    −: ((Oil×Oil)$\overset{\sim}{\rightarrow}$Oil) | ((Gas×Gas)$\overset{\sim}{\rightarrow}$Gas)
        **pre** −(og,og′): obs_Amount(og)≥obs_Amount(og′)
    ∗: ((Oil×**Real**)→Oil) | ((Gas×**Real**)→Gas)
    /: ((Oil×Oil)→**Real**) | ((Gas×Gas)→**Real**)
    /: ((Oil×**Real**)→Oil) | ((Gas×**Real**)→Gas)
**axiom**
    ∀ g,g′:Gas, o,o′:Oil •
            obs_Amount(g+g′)=obs_Amount(g)+obs_Amount(g′) ∧
            obs_Amount(o+o′)=obs_Amount(o)+obs_Amount(o′) ∧
            obs_Amount(g−g′)=obs_Amount(g)−obs_Amount(g′) ∧
            obs_Amount(o−o′)=obs_Amount(o)−obs_Amount(o′)
    ∀ g,g′:Gas, g,g′:Gas, r:**Real** • g′∗(g/g′)=g ∧ o′∗(o/o′)=o
```

S.1.22 Simple Entities

The below suggests an answer to Exercise 1.10.22 on page 36.

S.1.23 Operations

The below suggests an answer to Exercise 1.10.23 on page 36.

S.1.24 Events

The below suggests an answer to Exercise 1.10.24 on page 36.

S.1.25 Behaviours

The below suggests an answer to Exercise 1.10.25 on page 36.

S.1.26 Atomic and Composite Entities

The below suggests an answer to Exercise 1.10.26 on page 36.

S.1.27 Mereology

The below suggests an answer to Exercise 1.10.27 on page 36.

See characterisation 31 on page 20: 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.

S.1.28 Operations Research (OR)

The below suggests an answer to Exercise 1.10.28 on page 37.

See Sect. 1.8.1 on page 32.

S.1.29 OR versus Domain Modelling

The below suggests an answer to Exercise 1.10.29 on page 37.

'OR Modelling' usually use classical applied mathematics: calculus ([partial] differential equations), statistics, probability theory, graph theory, combinatorics, signal analysis, theory of flows in networks, etcetera.

'Domain Modelling' use formal specifications and emphasises applied mathematical logic and modern algebra.

See Characterisation. 5 on page 5 and Sect. 1.8.3 on page 32.

S.2 Informative Documents

S.2.1 1.a

The below suggests an answer to Exercise 3.18.1 on page 65.

S.2.2 1.b

The below suggests an answer to Exercise 3.18.2 on page 65.

S.2.3 1.c

The below suggests an answer to Exercise 3.18.3 on page 65.

S.2.4 1.d

The below suggests an answer to Exercise 3.18.4 on page 65.

S.3 Stakeholder Identification and Liaison

S.3.1 2.a

The below suggests an answer to Exercise 4.7.1 on page 68.

S.3.2 2.b

The below suggests an answer to Exercise 4.7.2 on page 68.

S.3.3 2.c

The below suggests an answer to Exercise 4.7.3 on page 68.

S.3.4 2.d

The below suggests an answer to Exercise 4.7.4 on page 68.

S.4 Domain Acquisition

S.4.1 3.a

The below suggests an answer to Exercise 5.8.1 on page 73.

S.4.2 3.b

The below suggests an answer to Exercise 5.8.2 on page 73.

S.4.3 3.c

The below suggests an answer to Exercise 5.8.3 on page 73.

S.4.4 3.d

The below suggests an answer to Exercise 5.8.4 on page 73.

S.5 Business Processes

S.5.1 4.a

The below suggests an answer to Exercise 6.7.1 on page 76.

S.5.2 4.b

The below suggests an answer to Exercise 6.7.2 on page 76.

S.5.3 4.c

The below suggests an answer to Exercise 6.7.3 on page 76.

S.5.4 4.d

The below suggests an answer to Exercise 6.7.4 on page 76.

S.6 Domain Analysis and Concept Formation

S.6.1 5.a

The below suggests an answer to Exercise 7.6.1 on page 81.

S.6.2 5.b

The below suggests an answer to Exercise 7.6.2 on page 81.

S.6.3 5.c

The below suggests an answer to Exercise 7.6.3 on page 81.

S.6.4 5.d

The below suggests an answer to Exercise 7.6.4 on page 81.

S.7 Terminology

S.7.1 6.a

The below suggests an answer to Exercise 8.8.1 on page 84.

S.7.2 6.b

The below suggests an answer to Exercise 8.8.2 on page 84.

S.7.3 6.c

The below suggests an answer to Exercise 8.8.3 on page 85.

S.7.4 6.d

The below suggests an answer to Exercise 8.8.4 on page 85.

S.8 Domain Intrinsics

S.8.1 7.a

The below suggests an answer to Exercise 10.5.1 on page 94.

S.8.2 7.b

The below suggests an answer to Exercise 10.5.2 on page 94.

S.8.3 7.c

The below suggests an answer to Exercise 10.5.3 on page 94.

S.8.4 7.d

The below suggests an answer to Exercise 10.5.4 on page 94.

S.9 Domain Support Technologies

S.9.1 8.a

The below suggests an answer to Exercise 11.6.1 on page 98.

S.9.2 8.b

The below suggests an answer to Exercise 11.6.2 on page 98.

S.9.3 8.c

The below suggests an answer to Exercise 11.6.3 on page 98.

S.9.4 8.d

The below suggests an answer to Exercise 11.6.4 on page 98.

S.10 Domain Management and Organisation

S.10.1 9.a

The below suggests an answer to Exercise 12.5.1 on page 107.

S.10.2 9.b

The below suggests an answer to Exercise 12.5.2 on page 107.

S.10.3 9.c

The below suggests an answer to Exercise 12.5.3 on page 107.

S.10.4 9.d

The below suggests an answer to Exercise 12.5.4 on page 107.

S.11 Domain Rules and Regulations

S.11.1 10.a

The below suggests an answer to Exercise 13.7.1 on page 112.

S.11.2 10.b

The below suggests an answer to Exercise 13.7.2 on page 112.

S.11.3 10.c

The below suggests an answer to Exercise 13.7.3 on page 112.

S.11.4 10.d

The below suggests an answer to Exercise 13.7.4 on page 112.

S.12 Domain Scripts and Contracts

S.12.1 11.a

The below suggests an answer to Exercise 14.5.1 on page 117.

S.12.2 11.b

The below suggests an answer to Exercise 14.5.2 on page 117.

S.12.3 11.c

The below suggests an answer to Exercise 14.5.3 on page 117.

S.12.4 11.d

The below suggests an answer to Exercise 14.5.4 on page 117.

S.13 Domain Human Behaviour

S.13.1 12.a

The below suggests an answer to Exercise 15.6.1 on page 120.

S.13.2 12.b

The below suggests an answer to Exercise 15.6.2 on page 120.

S.13.3 12.c

The below suggests an answer to Exercise 15.6.3 on page 120.

S.13.4 12.d

The below suggests an answer to Exercise 15.6.4 on page 120.

S.14 Domain Verification

S.14.1 13.a

The below suggests an answer to Exercise 16.7.1 on page 123.

S.14.2 13.b

The below suggests an answer to Exercise 16.7.2 on page 123.

S.14.3 13.c

The below suggests an answer to Exercise 16.7.3 on page 123.

S.14.4 13.d

The below suggests an answer to Exercise 16.7.4 on page 123.

S.15 Domain Validation

S.15.1 14.a

The below suggests an answer to Exercise 17.3.1 on page 125.

S.15.2 14.b

The below suggests an answer to Exercise 17.3.2 on page 125.

S.15.3 14.c

The below suggests an answer to Exercise 17.3.3 on page 125.

S.15.4 14.d

The below suggests an answer to Exercise 17.3.4 on page 125.

S.16 Domain Theory Formation

S.16.1 15.a

The below suggests an answer to Exercise 18.3.1 on page 127.

S.16.2 15.b

The below suggests an answer to Exercise 18.3.2 on page 127.

S.16.3 15.c

The below suggests an answer to Exercise 18.3.3 on page 127.

S.16.4 15.d

The below suggests an answer to Exercise 18.3.4 on page 127.

S.17 Domain Engineering: A Postludium

S.17.1 15.a.x

The below suggests an answer to Exercise 19.6.1 on page 132.

S.17.2 15.b.x

The below suggests an answer to Exercise 19.6.2 on page 132.

S.17.3 15.c.x

The below suggests an answer to Exercise 19.6.3 on page 132.

S.17.4 15.d.x

The below suggests an answer to Exercise 19.6.4 on page 132.

S.18 From Domains to Requirements

S.18.1 16.a

The below suggests an answer to Exercise 20.9.1 on page 137.

S.18.2 16.b

The below suggests an answer to Exercise 20.9.2 on page 137.

S.18.3 16.c

The below suggests an answer to Exercise 20.9.3 on page 137.

S.18.4 16.d

The below suggests an answer to Exercise 20.9.4 on page 137.

S.19 Summary and Conclusion

S.19.1 17.a

The below suggests an answer to Exercise 21.6.1 on page 141.

S.19.2 17.b

The below suggests an answer to Exercise 21.6.2 on page 141.

S.19.3 17.c

The below suggests an answer to Exercise 21.6.3 on page 141.

S.19.4 17.d

The below suggests an answer to Exercise 21.6.4 on page 141.

Part X

ADMINISTRATIVE APPENDICES

T
Bibliographical Notes

Specification languages, techniques and tools, that cover the spectrum of domain and requirements specification, refinement and verification, are dealt with in Alloy: [106], ASM: [161, 162], B/event B: [2, 45], CSP [95, 167, 169, 96], DC [196, 197] (Duration Calculus), Live Sequence Charts [55, 87, 110], Message Sequence Charts [102, 103, 104], RAISE [72, 74, 25, 26, 27, 71, 33] (RSL), Petri nets [108, 152, 158, 157, 159], Statecharts [83, 84, 86, 88, 85], Temporal Logic of Reactive Systems [128, 129, 144, 155], TLA+ [115, 116, 134, 135] (Temporal Logic of Actions), VDM [36, 37, 69, 68], and Z [173, 174, 194, 93, 92]. Techniques for integrating "different" formal techniques are covered in [7, 77, 43, 41, 166]. The recent book on Logics of Specification Languages [35] 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. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings.* Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
3. Jean-Raymond Abrial and L. Mussat. *Event B Reference Manual (Editor: Thierry Lecomte)*, June 2001. Report of EU IST Project Matisse IST-1999-11435.
4. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS. Springer–Verlag, 1998.
5. Krzysztof R. Apt. *From Logic Programming to Prolog.* Prentice Hall, International Series in Computer Science, 1997. viii + 328 pages.
6. Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, August 2003. ISBN 0521825830.
7. 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.

8. 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.
9. Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, US, 1996.
10. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Heidelberg, Germany, 1998.
11. John W. Backus and Peter Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–1, 1963.
12. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version I. Technical report, IBM Laboratory, Vienna, 1966.
13. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version II. Technical report, IBM Laboratory, Vienna, 1968.
14. Hans Bekič, Peter Lucas, Kurt Walk, and Many Others. Formal Definition of PL/I, ULD Version III. IBM Laboratory, Vienna, 1969.
15. 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.
16. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, September 1996.
17. G.M. Birtwistle, O.-J.Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA* `begin`. Studentlitteratur, Lund, Sweden, 1974.
18. Dines Bjørner. Programming in the Meta-Language: A Tutorial. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language, [36]*, LNCS, pages 24–217. Springer–Verlag, 1978.
19. 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, [36]*, LNCS, pages 337–374. Springer–Verlag, 1978.
20. 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.
21. Dines Bjørner, editor. *Abstract Software Specifications*, volume 86 of *LNCS*. Springer–Verlag, 1980.
22. Dines Bjørner. Application of Formal Models. In *Data Bases*. INFOTECH Proceedings, October 1980.
23. Dines Bjørner. Formalization of Data Base Models. In Dines Bjørner, editor, *Abstract Software Specification, [21]*, volume 86 of *LNCS*, pages 144–215. Springer–Verlag, 1980.
24. Dines Bjørner. Documents: A Domain Analysis. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
25. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
26. 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.

27. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
28. 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.
29. 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.
30. 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).
31. 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.
32. 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.
33. 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.
34. 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.
35. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [162, 45, 62, 141, 81, 71, 135, 68, 92]*. EATCS Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
36. 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*. [18, 19, 20].
37. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
38. 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.
39. Dines Bjørner and Hans Henrik Løvengreen. Formalization of Data Models. In *Formal Specification and Software Development, [37]*, chapter 12, pages 379–442. Prentice-Hall, 1982.
40. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
41. 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.

42. J. P. Bowen. Glossary of Z notation. *Information and Software Technology*, 37(5–6):333–334, May–June 1995.
43. 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.
44. 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: [161, 61, 142, 73, 134, 93] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
45. Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages 47–152 in [35]. Springer, 2008.
46. 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.
47. 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 [48].
48. 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.
49. 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).
50. David Crystal. *The Cambridge Encyclopedia of Language*. Cambridge University Press, 1987, 1988.
51. CVS. Concurrent Versions System Home Page. Electronically, on the Web: `www.cvshome.org`, 2005.
52. O.-J. Dahl, Edsger Wybe Dijkstra, and Charles Anthony Richard Hoare. *Structured Programming*. Academic Press, 1972.
53. O.-J. Dahl and Charles Anthony Richard Hoare. Hierarchical program structures. In *[52]*, pages 197–220. Academic Press, 1972.
54. O.-J. Dahl and K. Nygaard. SIMULA – an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.
55. 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.
56. C.J. Date. *An Introduction to Database Systems, I*. The Systems Programming Series. Addison Wesley, 1981.
57. C.J. Date. *An Introduction to Database Systems, II*. The Systems Programming Series. Addison Wesley, 1983.

58. Jim Davies. Announcement: Electronic version of Communicating Sequential Processes (CSP). Published electronically: http://www.usingcsp.com/, 2004. Announcing revised edition of [95].
59. J.W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
60. 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.
61. 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: [161, 44, 142, 73, 134, 93] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
62. Răzvan Diaconescu. *Logics of Specification Languages*, chapter A Methodological Guide to the CafeOBJ Logic, pages 153–240 in [35]. Springer, 2008.
63. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
64. 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.
65. 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.
66. R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, Cambridge, Mass., USA, 2003. 3rd Edition.
67. 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.
68. John S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages 453–487 in [35]. Springer, 2008.
69. John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using VDM-SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.
70. FOLDOC: The free online dictionary of computing. Electronically, on the Web: http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?ISWIM, 2004.
71. Chris George and Anne E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages 349–399 in [35]. Springer, 2008.
72. 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.
73. 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: [161, 44, 61, 142, 134, 93] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

74. 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.
75. James Gosling and Frank Yellin. *The Java Language Specification*. Addison--Wesley & Sun Microsystems. ACM Press Books, 1996. 864 pp, ISBN 0-10-63451-1.
76. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
77. 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.
78. Oliver Grillmeyer. *Exploring Computer Science with Scheme*. Springer-Verlag, New York, USA, 1998.
79. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
80. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.
81. Michael R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages 299–347 in [35]. Springer, 2008.
82. Michael Reichhardt Hansen and Hans Rischel. *Functional Programming in Standard ML*. Addison Wesley, 1997.
83. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
84. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
85. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
86. 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.
87. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
88. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
89. E.C.R. Hehner. *The Logic of Programming*. Prentice-Hall, 1984.
90. E.C.R. Hehner. *a Practical Theory of Programming*. Springer-Verlag, 2nd edition, 1993. On the net: http://www.cs.toronto.edu/~hehner/aPToP/.
91. 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.
92. Martin C. Henson, Moshe Deutsch, and Steve Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages 489–596 in [35]. Springer, 2008.
93. 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: [161, 44, 61, 142, 73, 134] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

94. 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.
95. Tony Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
96. Tony Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [95]. See also `http://www.usingcsp.com/`.
97. Christopher John Hogger. *Essentials of Logic Programming*. Graduate Texts in Computer Science, no.1, 310 pages. Clarendon Press, December 1990. .
98. IEEE CS. IEEE Standard Glossay of Software Engineering Terminology, 1990. IEEE Std.610.12.
99. IEEE: The Institute for Electrical and Electronics Engineers. The IEEE Home Page. Electronically, on the Web: `http://www.ieee.org`, 2005.
100. 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.
101. ISO: The International Standards Organisation. The ISO Home Page. Electronically, on the Web: `http://www.iso.org`, 2005.
102. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
103. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
104. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
105. ITU: The International Telecommunications Union. The ITU Home Page. Electronically, on the Web: `http://www.itu.org`, 2005.
106. Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
107. 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.
108. 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.
109. Brian Kernighan and Dennis Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1989.
110. 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.
111. D.E. Knuth. *The Art of Computer Programming, Vol.1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., USA, 1968.
112. D.E. Knuth. *The Art of Computer Programming, Vol.2.: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., USA, 1969.
113. D.E. Knuth. *The Art of Computer Programming, Vol.3: Searching & Sorting*. Addison-Wesley, Reading, Mass., USA, 1973.
114. 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.).

115. Leslie Lamport. The Temporal Logic of Actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995.
116. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
117. J.C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *15th. Int. Symp. on Fault-tolerant computing*. IEEE, 1985.
118. J.A.N. Lee. *Computer Semantics*. Van Nostrand Reinhold Co., 1972.
119. 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.
120. H.S. Leonard and N. Goodman. The Calculus of Individuals and Its Uses. *Journal of Symbolic Logic*, 5:45–55, 1940.
121. Xavier Leroy and Pierre Weis. *Manuel de Référence du langage Caml*. InterEditions, Paris, France, 1993. ISBN 2-7296-0492-8.
122. 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.
123. W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1987.
124. J.W. Lloyd. *Foundation of Logic Programming*. Springer-Verlag, 1984.
125. P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Devt. and Res.*, 25(5):549–561, 1981.
126. P. Lucas and K. Walk. On the formal description of PL/I. *Annual Review Automatic Programming Part 3*, 6(3), 1969.
127. E.C. Luschei. *The Logical Systems of Leśniewksi*. North Holland, Amsterdam, The Netherlands, 1962.
128. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.
129. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.
130. ANSI X3.23-1974. The Cobol programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1974.
131. ANSI X3.53-1976. The PL/I programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1976.
132. ANSI X3.9-1966. The Fortran programming language. Technical report, American National Standards Institute, Standards on Computers and Information Processing, 1966.
133. J. McCarthy and et al. *LISP 1.5, Programmer's Manual*. The MIT Press, Cambridge, Mass., USA, 1962.
134. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [161, 44, 61, 142, 73, 93] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
135. Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages 401–451 in [35]. Springer, 2008.

136. 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.
137. 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.
138. D. Miéville and D. Vernant. *Stanisław Leśniewksi aujourd'hui*. Grenoble, October 8-10, 1992.
139. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., USA and London, England, 1990.
140. C. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.
141. T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. *Logics of Specification Languages*, chapter CASL – the Common Algebraic Specification Language, pages 241–298 in [35]. Springer, 2008.
142. 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: [161, 44, 61, 73, 134, 93] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
143. 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.
144. Ben C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.
145. 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.
146. 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.
147. 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.
148. 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.
149. 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.
150. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
151. David Lorge Parnas. A technique for software module specification with examples. *Communications of the ACM*, 14(5), May 1972.
152. Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

153. Charles Petzold. *Programming Windows with C# (Core Reference)* . Microsoft Corporation, Redmond, WA, USA, 2001. 1200 pages.
154. 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.
155. 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. .
156. 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.
157. Wolfang Reisig. *A Primer in Petri Net Design*. Springer Verlag, March 1992. 120 pages.
158. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.
159. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.
160. Wolfgang Reisig. On Gurevich's Theorem for Sequential Algorithms. *Acta Informatica*, 2003.
161. Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [44, 61, 142, 73, 134, 93] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
162. Wolfgang Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [35]. Springer, 2008.
163. John C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
164. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
165. P.M. Roget. *Roget's Thesaurus*. Collins, London and Glasgow, 1852, 1974.
166. 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.
167. 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.
168. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
169. 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.
170. Peter Sestoft. *Java Precisely*. The MIT Press, 25 July 2002. 100 pages (sic !), ISBN 0262692767.

171. 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.).
172. 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.
173. 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.
174. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
175. J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.
176. J.T.J. Srzednicki and Z. Stachniak, editors. *Leśniewksi's Lecture Notes in Logic*. Dordrecht, 1988.
177. J.T.J. Srzednicki and Z. Stachniak. *Leśniewksi's Systems Protothetic*. . Dordrecht, 1998.
178. Staff of Encyclopœdia Brittanica. Encyclopœdia Brittanica. Merriam Webster/Brittanica: Access over the Web: http://www.eb.com:180/, 1999.
179. 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.
180. Staff of Oxford University Press. *The Oxford Dictionary of Quotations*. Oxford University Press, London, 1941, 1974.
181. Jess Stein, editor. *The Random House American Everyday Disctionary*. Random House, New York, N.Y., USA, 1949, 1961.
182. B. Stroustrup. *C++ Programming Language*. Addison-Wesley Publishing Company, 1986.
183. 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.
184. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
185. Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 2nd edition, 29 March 1999. 512 pages, ISBN 0201342758.
186. 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.
187. 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.
188. 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.
189. 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.

190. Peter van Roy and Seif Haridi. *Concepts, Techniques and Models of Computer Programming*. The MIT Press, Cambridge, Mass., USA, March 2004.

191. Bill Venners. *Inside the Java 2.0 Virtual Machine (Enterprise Computing)*. McGraw-Hill; ISBN: 0071350934, October 1999.

192. Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, Paris, France, 1999. ISBN 2-10-004383-8, Second edition.

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

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

195. Edward N. Zalta. Logic. In *The Stanford Encyclopedia of Philosophy*. Published: http://plato.stanford.edu/, Winter 2003.

196. 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.

197. Chao Chen Zhou, Charles Anthony Richard Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

U

Glossary

U.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 [179]

U.1.1 Glossary

According to [123] 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 [123] a *glossary* is therefore "a collection of glosses, a list with explanations of abstruse, antiquated, dialictical, or technical terms; a partial dictionary." [42] provides a *Glossary of Z Notation.*

U.1.2 Dictionary

According to [123] 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 [179, 180, 123].

U.1.3 Encyclopædia

According to [123], 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." [178] is, perhaps, the most "famous" encyclopædia.

U.1.4 Ontology

By *ontology* is meant [123]: "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.

U.1.5 Taxonomy

By *taxonomy* is meant [123]: "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."

U.1.6 Terminology

By a *term* is here meant [123]: "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 [123]: "the doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature." [117] provides a terminology of *Dependable Computing and Fault Tolerance: Concepts and Terminology*.

U.1.7 Thesaurus

By *thesaurus* is, in general, meant [123]: "a 'treasury' or 'storehouse' of knowledge, as a dictionary, encyclopædia or the like. (1736)" The thesaurus [165] has set a unique standard for and "the" meaning, now, of the term 'thesaurus'.

U.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 ser italicized font* stand for defined terms.
- Definiens (second part) text ending with [179] (or [123]) 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.

U.3 The SI Units and Wikipedia

The definition of a number of terms related to physics: *acceleration, Ampere, candela, Celcius, direction, distance, Hertz, Joule, Kelvin, kilogram, mass, meter, mole, motion, Newton, Ohm, position, second, speed, steridian, velocity, Volt, Watt,* etc., have been "lifted" from the web pages of The International System of Units (abbreviated SI from the French Le Système International d'Unités) and The Wikipedia.

U.4 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. **Acceleration:** 1. In Physics, acceleration is the *change in velocity over time*. In other words acceleration is the rate at which something speeds up or slows down. 2. In kinematics, acceleration is defined as the first derivative of velocity with respect to time (that is, the rate of change of *velocity*), or equivalently as the second derivative of *position*. It is a vector quantity with dimension $\mathsf{Length}\cdots\mathsf{Time}^{-2}$. In SI units, acceleration is measured in metres per second squared (m/s^2).
8. **Acquirer:** The legal entity, a person, an institution or a firm which orders some *development* to take place. (Synonymous terms are *client* and *customer*.)
9. **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*.)
10. **Action:** By an action we shall understand something who potentially changes a *state*.
11. **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*.)
12. **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*.)
13. **Adaptive:** By adaptive we mean some thing that can adapt or arrange itself to a changing *context*, a changing *environment*.
14. **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*.)
15. **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.)
16. **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 $f : S \rightarrow S$. (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 [111]. 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 [80, 160, 161]. As such this definition goes beyond Turing machine and Lambda-calculus "compatibility". That is, captures more!)

17. **Algorithmic:** Adjective form of *algorithm*.
18. **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*.
19. **Ampere:** The Ampere is that constant current which, if maintained in two straight parallel conductors of infinite length, of negligible circular cross-section, and placed 1 metre apart in vacuum, would produce between these conductors a force equal to $2 * 10^{-7}$ Newton per meter of length.
20. **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 [123]. (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.)
21. **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).)
22. **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*.)

23. **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.)
24. **Applicative programming:** See the term *applicative* just above. (Thus the terms applicative programming and *functional programming* are here used synonymously.)
25. **Applicative programming language:** Same as *functional programming language*.
26. **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.)
27. **Artefact:** An artificial product [123]. (Anything designed or constructed by humans or machines, which is made by humans.)
28. **Artifact:** Same term as *artefact*.
29. **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.)
30. **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.)
31. **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"!)
32. **Axiom:** An established rule or principle or a self-evident truth.

33. **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.)
34. **Axiom system:** Same as *axiomatic specification*.

$\mathcal{B}$

35. **B:** B stands for Bourbaki, pseudonym for a group of mostly French mathematicians which began meeting in the 1930s, aiming to write a thorough unified set-theoretic account of all mathematics. They had tremendous influence on the way mathematics has been done since. (The founding of the Bourbaki group is described in André Weil's autobiography, titled something like "memoir of an apprenticeship" (orig. Souvenirs D'apprentissage). There is a usable book on Bourbaki by J. Fang. Liliane Beaulieu has a book forthcoming, which you can sample in "A Parisian Cafe and Ten Proto-Bourbaki Meetings 1934–1935" in the Mathematical Intelligencer 15 no. 1 (1993) 27–35. From `http://www.faqs.org/faqs/-sci-math-faq/bourbaki/` (2004). Founding members were: Henri Cartan, Claude Chevalley, Jean Coulomb, Jean Delsarte, Jean Dieudonné, Charles Ehresmann, René de Possel, Szolem Mandelbrojt, André Weil. From: `http://www.bourbaki.ens.fr/` (2004). B also stands for a model-oriented specification language [2].)
36. **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* [319], in particular manifest *entities* [154]. And then behaviour is that which can be observed about the *value* of the *entity* and its *interaction* with an *environment*.)
37. **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.)
38. **Boolean connective:** By a *Boolean connective* we mean either of the Boolean operators: $\wedge$, $\vee$, $\Rightarrow$ (or $\supset$), $\sim$ (or $\neg$).
39. **BPR:** See *business process reengineering*
40. **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*.)
41. **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*.)
42. **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.)

43. **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*.)

$\mathcal{C}$

44. **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.)
45. **Candela:** The candela is the luminous intensity, in a given direction, of a source that emits monochromatic radiation of frequency $540 * 10^{12}$ *Hertz* and that has a radiant intensity in that direction of 1/683 *Watt* per *steradian*.
46. **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.
47. **Cartesian:** By a Cartesian is understood an ordered product, a fixed grouping, a fixed composition, of *entities* [154]. (Cartesian derives from the name of the French mathematician René Descartes.)
48. **Celcius:** See *Kelvin*, Item 237.
49. **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.)
50. **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*.)
51. **Class:** By a class we mean either of two things: a **class** *clause*, as in `RSL`, or a set of *entities* [154] defined by some *specification*, typically a *predicate*.
52. **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.)
53. **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).)

54. **Code:** By code we mean a *program* which is expressed in the machine language of a computer.
55. **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).)
56. **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.
57. **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.
58. **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.)
59. **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.
60. **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*.
61. **Compositional:** We say that two or more *phenomena* [319] or *concepts* [69] are compositional if it is meaningful to *compose* [60] 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.)
62. **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*.)
63. **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.
64. **Computation:** See *calculation*.
65. **Compute:** Given an expression and an applicable *rule* of a *calculus*, to change the former expression into a resulting expression. (Same as *calculate*.)

66. **Computer Science:** The study and knowledge of the phenomena that can exist inside computers.
67. **Computing Science:** The study and knowledge of how to construct those phenomena that can exist inside computers.
68. **Computing system:** A combination of *hardware* and *software* that together make meaningful *computation*s possible.
69. **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"!)
70. **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).)
71. **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]).)
72. **Concrete syntax:** A *concrete syntax* is a syntax which prescribes actual, computer representable *data structure*s. (Typically a *BNF Grammar* is a concrete syntax.)
73. **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.)
74. **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.)
75. **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!)
76. **Correct:** See next entry: *correctness*.
77. **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*.)

78. **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*.)
79. **CSP:** Abbreviation for Communicating Sequential Processes. (See [95, 167] and Chap. 21. Also, but not in this book, a term that covers constraint satisfaction problem (or programming).)
80. **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).)

$\mathcal{D}$

81. **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.)
82. **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 [56, 57] for seminal coverage, and to [23, 22, 38, 39] for formalisation, of these database forms.)
83. **Database schema:** By a database schema we understand a *type definition* of the structure of the data kept in a database.
84. **Data abstraction:** Data abstraction takes place when we abstract from the particular formal representation of data.
85. **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*.)
86. **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.)
87. **Data reification:** Same as *data refinement*. (To reify is to render something abstract as a material or concrete thing.)

88. **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.)
89. **Data transformation:** Same as *data refinement* and, hence, *data reification*.
90. **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*.
91. **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.
92. **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.
93. **Decomposition:** By a decomposition is meant the presentation of the parts of a *composite* "thing".
94. **Definiendum:** The left-hand side of a *definition*, that which is to be defined.
95. **Definiens:** The right-hand side of a *definition*, that which is defining "something".
96. **Definite:** Something which has specified limits. (Watch out for the four terms: *finite*, *infinite*, *definite* and *indefinite*.)
97. **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.
98. **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*.)
99. **Denotation:** A direct specific meaning as distinct from an implied or associated idea [179]. (By a denotation we shall, in our context, associate the idea of mathematical functions: That is, of the *denotational semantics* standing for functions.)
100. **Denotational:** Being a *denotation*.
101. **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.
102. **Denote:** Designates a mathematical meaning according to the principles of *denotational semantics*. (Sometimes we use the looser term designate.)
103. **Dependability:** Dependability is defined as the property of a *machine* such that reliance can justifiably be placed on the service it delivers [156].

(See definition of the related terms: *error*, *failure*, *fault* and *machine service*.)

104. **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.
105. **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*.)
106. **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 description is a description of the *entities* [154], *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*.)
107. **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.
108. **Designate:** To designate is to present a reference to, to point out, something. (See also *denote* and *designation*.)
109. **Designation:** The relation between a *syntactic* marker and the semantic thing signified. (See also *denote* and *designate*.)
110. **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*.)
111. **Developer:** The person, or the company, which constructs an *artefact*, as here, a *domain description*, or a *requirements prescription*, or a *software design*.
112. **Development:** The set of actions that are carried out in order to construct an *artefact*.

113. **Diagram:** A usually two-dimensional drawing, a figure. (Sometimes a diagram is annotated with informal and *formal* text.)
114. **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).)
115. **Dictionary:** See Sect. U.1.2
116. **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.)
117. **Directed graph:** A directed graph is a *graph* all of whose *edge*s are directed, i.e., are *arrow*s.
118. **Direction:** Direction is the information contained in the relative *position* of one point with respect to another point without the *distance* information.
119. **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.)
120. **Discrete:** As opposed to *continuous*: consisting of distinct or unconnected elements [179].
121. **Disjunction:** Being separated, being disjoined, decomposed. (We shall mostly think of disjunction as the (meaning of the) logical connective "or": $\vee$.)
122. **Distance:** Distance is a numerical description of how far apart objects are. In physics or everyday discussion, distance may refer to a physical length, a period of time, or an estimation based on other criteria (e.g. "two counties over"). In mathematics, distance must meet more rigorous criteria.
123. **Document:** By a document is meant any text, whether informal or *formal*, whether *informative* 214, *descriptive* 106 (or *prescriptive* 332) or *analytic*. (Descriptive documents may be *rough sketch*es, *terminologies* 472, *narrative*s, or *formal*. Informative documents are not *descriptive* 106. Analytic documents "describe" relations between documents, *verification* and *validation*, or describe properties of a document.)
124. **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.
125. **Domain:** Same as *application domain*; hence see that term for a characterisation. (The term domain is the preferred term.)
126. **Domain acquisition:** The act of acquiring, of gathering, *domain knowledge*, and of analysing and recording this knowledge.
127. **Domain analysis:** The act of analysing recorded *domain knowledge* in search of (common) properties of phenomena, or relating what may be considered separate phenomena.

128. **Domain capture:** The act of gathering *domain knowledge*, of collecting it — usually from domain *stakeholder*s.
129. **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.)
130. **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.)
131. **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*.)
132. **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*.)
133. **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).)
134. **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*.
135. **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*.)
136. **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*.)

137. **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*.)
138. **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*.)
139. **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*.)
140. **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) [94, 67].)
141. **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 definitions 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*.)
142. **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* [154], *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.)

143. **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.)
144. **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*.)
145. **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*.

$\mathcal{E}$

146. **Elaborate:** See next: *elaboration*.
147. **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.
148. **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.)
149. **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.)
150. **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*.)
151. **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.

152. **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*.)
153. **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.)
154. **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.)
155. **Enumerable:** By enumerable we mean that a set of elements satisfies a *proposition*, i.e., can be logically characterised.
156. **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*.)
157. **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* [147], *evaluated* [161], or *interpreted* [232]. (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.)
158. **Epistemology:** The study of knowledge. (Contrast, please, to *ontology*.)
159. **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 [156]. (An error is caused by a *fault*.)
160. **Evaluate:** See next: *evaluation*.
161. **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.

162. **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.)
163. **Expression:** An expression, in our context (i.e., that of software engineering), is a syntactical entity which, through *evaluation*, designates a *value*.
164. **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 [145].)
165. **Extensional:** Concerned with objective reality [179]. (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*.)

$\mathcal{F}$

166. **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.)
167. **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 [156]. (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*.)
168. **Fault:** The adjudged (i.e., the 'so judged') or hypothesised cause of an *error* [156]. (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.)
169. **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].)
170. **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.)

171. **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*.)
172. **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.
173. **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).
174. **Formal definition:** Same as *formal description*, *formal prescription* or *formal specification*.
175. **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).)
176. **Formal description:** A *formal description* of something. (Usually we use the term formal description only in connection with *formalisation* of *domain*s.)
177. **Formalisation:** The act of making a formal specification of something elsewhere informally specified; or the document which results therefrom.
178. **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.

[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.).

UML is a case in point — much is presently being done to formalise subsets of UML [147].)

179. **Formal prescription:** Same as *formal definition* or *formal specification*. (Usually we use the term formal prescription only in connection with *formalisation* of *requirements*.)
180. **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 designs*.)
181. **Function:** By a function we understand something which when *applied* [21] 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.)
182. **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 [91, 170, 82]), 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.)
183. **Functional:** A function whose arguments are allowed themselves to be functions is called a functional. (The *fix point* (finding) function is a functional.)
184. **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 *values*, 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 [185, 186, 139].)
185. **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 [49, 47, 48, 192, 121], Haskell [185], Miranda [186],

`Scheme` [1, 78, 66] and `SML` (Standard ML) [139, 82]. `LISP 1.5` was a first functional programming language [133].)

186. **Function application:** The act of applying a function to an argument is called a function application. (See 'comment' field of *function activation* just above.)
187. **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.)
188. **Function invocation:** Same as *function application*. (See parenthesized remark of entry 182 (*function activation*).)
189. **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).

$\mathcal{G}$

190. **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.).)
191. **Glossary:** See Sect. U.1.1.
192. **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.)
193. **Grouping:** By grouping we mean the ordered, finite collection, into a *Cartesian*, of mathematical structures (i.e., *value*s).

$\mathcal{H}$

194. **Hardware:** By hardware is meant the physical embodiment of a computer: its electronics, its boards, the racks, cables, button, lamps, etc.
195. **HCI:** Abbreviation for human computer interface. (Same as *CHI*, and same as *man-machine* interface.)
196. **Hertz:** The Hertz (symbol: **Hz**) is a measure of frequency per unit of time, or the number of cycles per *second*.
197. **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*.)

$\mathcal{I}$

198. **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.
199. **Identifier:** A name. (Usually represented by a string of alphanumeric characters, sometimes with properly infixed "-"s or "_"s.)
200. **Imperative:** Expressive of a command [179]. (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*.)
201. **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.)
202. **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. [132, 130, 11, 131, 11, 109]. Today programming languages like `C++, Java, C#,` etc. [182, 170, 91] additionally offer *module* cum *object* "features".)
203. **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*.)
204. **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*).
205. **Incomplete:** We say that a *proof system* is incomplete if not all true sentences are provable.
206. **Incompleteness:** Noun form of the *incomplete* adjective.
207. **Inconsistent:** A set of *axiom*s is said to be inconsistent if, by means of these, and some *deduction rule*s, one can *prove* [345] a property and its negation.
208. **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*.)

209. **Indicative:** Stating an objective fact. (See also *imperative*, *optative* and *putative*.)
210. **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*.)
211. **Infinite:** As you would think of it: not finite! (Watch out for the four terms: *finite*, *infinite*, *definite* and *indefinite*.)
212. **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 [147].)
213. **Informatics:** The confluence of (i) *application*s, (ii) *computer science*, (iii) *computing science* [i.e., the art [111, 112, 113] (1968–1973), craft [163] (1981), discipline [63] (1976), logic [89] (1984), practice [90] (1993–2004), and science [76] (1981) of programming], (iv) *software engineering* and (v) *mathematics*.
214. **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.)
215. **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.)
216. **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*.)
217. **Infrastructure:** According to the World Bank: *'Infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spillovers from users to nonusers).* We shall use the term as follows: Infrastructures are concerned with supporting other systems or activities. Computing systems for infrastructures are thus likely to be distributed and concerned in particular with supporting communication of information, control, people and materials. Issues of (for example) openness, timeliness, security, lack of corruption, and resilience are often important. (Winston Churchill is

quoted to have said, during a debate in the House of Commons, in 1946: *... The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms as 'infra-structures'.*)

218. **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*)).)
219. **Instance:** An individual, a thing, an *entity*. (We shall usually think of an 'instance' as a *value*.)
220. **Instantiation:** 'To represent (an abstraction) by a concrete *instance*' [179]. (We shall sometimes be using the term 'instantiation' in lieu of a *function invocation* on an *activation stack*.)
221. **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*.)
222. **Intangible:** Not *tangible*.
223. **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*.)
224. **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 [145]. The intension determines the *extension* [145].)
225. **Intensional:** Adjective form of *intension*.
226. **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*.)
227. **Interaction:** Two-way reciprocal action.
228. **Interface:** Boundary between two disjoint sets of communicating phenomena or concepts. (We shall think of the systems as *behaviour*s or *process*es, the boundary as being *channel*s, and the communications as *input*s and *output*s.)
229. **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*.)

230. **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*.
231. **Interpret:** See next: *interpretation*.
232. **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.
233. **Interpreter:** An interpreter is an *agent*, a *machine*, which performs *interpretation*s.
234. **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*.)
235. **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*!)

$\mathcal{J}$

236. **Joule:** The Joule, J, is the derived unit of energy in the *International System of Units*. It is defined as: $1\,\mathrm{J} = 1\,\mathrm{kg}\cdot\frac{\mathrm{m}^2}{\mathrm{s}^2}$ (where kg is *kilogram*, m is *meter*, s is *second*).

$\mathcal{K}$

237. **Kelvin:** The Kelvin, unit of thermodynamic temperature, is the fraction 1/273.16 of the thermodynamic temperature of the triple point of water.

The thermodynamic temperature, symbol T, in terms of its difference from the reference temperature $T_0 = 273.15K$, the ice point. This temperature difference is called a Celsius temperature, symbol t, and is defined by the quantity equation $t = T - T_0$.

238. **Keyword:** A significant word from a title or document. (See *KWIC*.)
239. **Kilogram:** The kilogram is the unit of mass; it is equal to the mass of the international prototype of the kilogram.
240. **Knowledge:** What is, or what can be known. The body of truth, information, and principles acquired by mankind [179]. (See *epistemology* and *ontology*. *A priori knowledge:* Knowledge that is independent of all particular experiences. *A posteriori knowledge:* Knowledge, which derives from experience alone.)

$\mathcal{L}$

241. **Label:** Same as named *program point*.
242. **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*.
243. **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)).)
244. **Lemma:** An auxiliary *proposition* used in the demonstration of another proposition. (Instead of proposition we could use the term *theorem*.)
245. **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).
246. **Linguistics:** The study and knowledge of the *syntax*, *semantics* and *pragmatics* of *language*(s).
247. **List:** A list is an ordered sequence of zero, one or more not necessarily distinct entities.
248. **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.)
249. **Live Sequence Chart:** The Live Sequence Chart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [55, 87, 110].)
250. **Location:** By a location is meant an area of *storage*.

251. **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.)
252. **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.)
253. **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` [124, 97].)
254. **Loose specification:** By a loose specification is understood a specification which either *underspecifies* a problem, or specifies this problem *nondeterministically*.

$\mathcal{M}$

255. **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 [107].)
256. **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*.)
257. **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 [156].
258. **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*.)
259. **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*.)
260. **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 oper-

ational 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*.)

261. **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. (
262. **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*.))
263. **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*.)
264. **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.)
265. **Mass:** In physical science, mass refers to the degree of *acceleration* a body acquires when subject to a *force*: bodies with greater mass are accelerated less by the same force. One says the body of greater mass has greater inertia. (In everyday usage, mass is commonly confused with weight. But, in physics and engineering, weight means the strength of the gravitational pull on the object; that is, how heavy it is, measured in units of newtons. In everyday situations, the weight of an object is proportional to its mass, which usually makes it unproblematic to use the same word for both concepts. However, the distinction between mass and weight becomes important for measurements with a precision better than a few percent (due to slight differences in the strength of the Earth’s gravitational field at

different places), and for places far from the surface of the Earth, such as in space or on other planets.)

266. **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 [171, 127, 138, 176, 177, 183] and Leonard and Goodman [120].)
267. **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.)
268. **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.)
269. **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`).)
270. **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.)
271. **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.))
272. **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.)

273. **Meter:** The meter is the length of the path travelled by light in vacuum during a time interval of 1/299 792 458 of a *second*.
274. **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.)
275. **Model-oriented:** A specification (description, prescription) is said to be model-oriented if the specification (etc.) *denote*s a *model*. (Contrast to *property-oriented*.)
276. **Model-oriented type:** A type is said to be model-oriented if its specification *designate*s a *model*. (Contrast to *property-oriented type*.)
277. **Modularisation:** The act of structuring a text using *module*s.
278. **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 [54, 53, 17] and to [151, 150] for original, early papers on modules.)
279. **Module design:** By module design we shall understand the *design* of (one or more) *module*s.
280. **Mole:** The mole is the amount of substance of a system which contains as many elementary entities as there are atoms in 0.012 *kilogram* of carbon 12; its symbol is **mol**. (When the *mole* is used, the elementary entities must be specified and may be atoms, molecules, ions, electrons, other particles, or specified groups of such particles.)
281. **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 $f(a) \sqsubseteq f(a')$.
282. **Motion:** The constant change in the location of a body.
283. **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.)
284. **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”).

$\boxed{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\mathcal{N}}$

285. **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.”)
286. **Naming:** The action of allocating a unique name to a value.
287. **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.)
288. **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.)
289. **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*.)
290. **Newton:** The Wewton is the unit of force derived in the SI system; it is equal to the amount of force required to give a *mass* of one *kilogram* an *acceleration* of one *meter* per *second* squared. Algebraically: $1\,N = 1\,\frac{kg \cdot m}{s^2}$.
291. **Node:** A point in some *graph* or *tree*.
292. **Nondeterminate:** Same as *nondeterministic*.
293. **Nondeterministic:** A property of a specification: May, on purpose, i.e., deliberately have more than one meaning. (A specification which is ambiguous also has more than one meaning, but its ambiguity is of overriding concern: It is not ‘nondeterministic’ (and certainly not ‘deterministic’!).)
294. **Nondeterminism:** A *nondeterministic* specification models nondeterminism.
295. **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.)
296. **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”!)

$\mathcal{O}$

297. **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` [46, 64, 65], `RSL`, etc., each have their own, further refined, meaning for the term 'object', and so do *object-oriented programming language* (viz., `C++` [182], `Java` [9, 75, 122, 191, 4, 170], `C#` [153, 137, 136, 91] and so on).)
298. **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` [17].)
299. **Observer:** By an observer we mean basically the same as an *observer function*.
300. **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.)
301. **Ohm:** By definition in Ohm's Law, 1 Ohm equals 1 *Volt* divided by 1 *Ampere*. In other words, a device has a resistance of 1 Ohm if a voltage of 1 *Volt* will cause a current of 1 *Ampere* to flow.
302. **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. U.1.4.)
303. **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.)
304. **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*).)

305. **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.)
306. **Operational semantics:** A *definition* of a *language semantics* that is *operational*. (See also *structural operational semantics*.)
307. **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: [140, 16, 10].)
308. **Optative:** Expressive of wish or desire. (See also *imperative*, *indicative*, and *putative*.)
309. **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.
310. **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*.)
311. **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*).)
312. **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*.)

$\mathcal{P}$

313. **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.)

314. **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` [100], and of a specificational 'programming' language is `CSP` [95, 167, 169]. Most recent *imperative programming language*s (Java, C#, etc.) provide for programming constructs (e.g., threads) that somehow mimic parallel programming.)
315. **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*.)
316. **Performance:** By performance we, here, in the context of computing, mean quantitative figures for the use of computing resources: time, storage space, equipment.
317. **Performance requirements:** By performance requirements we mean *requirements* which express *performance* properties (desiderata).
318. **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.
319. **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.)
320. **Phenomenology:** Phenomenology is the study of structures of consciousness as experienced from the first-person point of view [195].
321. **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` operating system with an `Oracle Database` $10g$ running on a `Sun Fire E25K Server.`)
322. **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.)

323. **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.
324. **Position:** The (x, y, z) point in a three-dimensional co-ordinate system, or in a spherical co-ordinate system: the $r$adial distance of a point from a fixed origin, the $z$enith angle from the positive z-axis to the point, and the azimuth angle from the positive x-axis to the orthogonal projection of the point in the (x, y) plane.
325. **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*.)
326. **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!.)
327. **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.)
328. **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*.)
329. **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.
330. **Predicate logic:** A predicate logic is a language of *predicate*s (given by some *formal syntax*) and a *proof system*.
331. **Presentation:** By presentation we mean the syntactic *document*ation of the results of some *development*.
332. **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.)

333. **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*.)
334. **Principle:** An accepted or professed rule of action or conduct, . . . , a fundamental doctrine, right rules of conduct, . . . [181]. (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.)
335. **Procedure:** By a procedure we mean the same as a *function*. (Same as *routine* or *subroutine*.)
336. **Process:** By a process we understand a sequence of actions and events. The events designate interaction with some environment of the process.
337. **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.)
338. **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*.)
339. **Programmer:** A person who does *software design*.
340. **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.
341. **Programming:** The act of constructing *program*s. From [70]:

    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).
342. **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*.)
343. **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*.)

344. **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.)
345. **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, $\phi_2, \ldots, \phi_n$, 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 $\mathcal{L}$.
346. **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.)
347. **Proof rule:** Same as *inference rule* or *axiom*.
348. **Proof system:** A *consistent* and (relative) *complete* set of *proof rule*s.
349. **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".)
350. **Property-oriented:** A specification (description, prescription) is said to be property-oriented if the specification (etc.) expresses *attribute*s. (Contrast to *model oriented*.)
351. **Proposition:** An expression in language which has a truth value.
352. **Pure functional programming language:** A *functional programming language* is said to be pure if none of its constructs designates *side-effects*.
353. **Putative:** Commonly accepted or supposed, that is, assumed to exist or to have existed. (See also *imperative*, *indicative* and *optative*.)

$\mathcal{Q}$

354. **Quality:** Specific and essential character. (Quality is an *attribute*, a *property*, a characteristic (something has character).)
355. **Quantification:** The operation of quantifying. (See *quantifier*. The x (the y) is quantifying expression $\forall x{:}X \cdot P(x)$ (respectively $\exists y{:}Y \cdot Q(y)$).)
356. **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.)
357. **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.)

$\mathcal{R}$

358. **RAISE:** `RAISE` stands for `R`igorous `A`pproach to `I`ndustrial `S`oftware `E`ngineering. (`RAISE` refers to a method, `The RAISE Method` [74], a specification language, `RSL` [72], and "comes" with a set of tools.)
359. **Range:** The concept of range is here used in connection with functions. Same as *range set*. See next entry.
360. **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*.
361. **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*.)
362. **Reactive system:** A *system* whose main phenomena are chiefly *reactive*. (See the *reactive* entry just above.)
363. **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*.)
364. **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.)
365. **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*.)

366. **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).
367. **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.)
368. **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.)
369. **Refutation:** A refutation is a statement that (convincingly) refutes an assertion. (Lakatos [114] 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.)
370. **Reification:** The result of a *reify* action. (See also *data reification*, *operation reification* and *refinement*.)
371. **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.)
372. **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.)
373. **Renaming:** By renaming we mean *Alpha-renaming*. (Renaming, in this sense, is a concept of the *Lambda-calculus*.)
374. **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.)
375. **Requirements:** A condition or capability needed by a user to solve a problem or achieve an objective [98].
376. **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).)
377. **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.

378. **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*.)
379. **Requirements definition:** Proper *definition*al part of a *requirements prescription*.
380. **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*.)
381. **Requirements elicitation:** By requirements elicitation we mean the actual extraction of *requirements* from *stakeholder*s.
382. **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*).)
383. **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*.
384. **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*.
385. **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*.)
386. **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.)
387. **Requirements specification:** Same as *requirements prescription* — the preferred term.
388. **Requirements validation:** By requirements validation we rather mean the *validation* of a *requirements prescription*.

389. **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.)
390. **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*.)
391. **Rigorous:** Favoring rigor, i.e., being precise.
392. **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).)
393. **Risk:** The Concise Oxford Dictionary [123] 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.
394. **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*.)
395. **Root:** A root is a *node* of a *tree* which is not a sub*tree* of a larger, *embedding* (*embedded*) tree.
396. **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.
397. **Route:** Same as *path*.
398. **Routine:** Same as *procedure*.
399. **RSL:** RSL stands for the RAISE [74] Specification Language [72]. ()
400. **Rule:** A regulating principle. (We use the concept of rules in several different contexts: *rewrite rule*, *rule of grammar* and *rules and regulations*.)

$\mathcal{S}$

401. **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*.)

402. **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.
403. **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.
404. **Second:** The second is the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium 133 atom.
405. **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.)
406. **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.)
407. **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*.
408. **Semantics:** Semantics is the study and knowledge [incl. specification] of meaning in language [50]. (We make the distinction between the *pragmatics*, the semantics and the *syntax* of languages. Leading textbooks on semantics of programming languages are [59, 79, 164, 168, 184, 193].)
409. **Semantic function:** A semantics function is a function which when applied to *syntactic values* yields their *semantic values*.
410. **Semantic type:** By a semantic type we mean a *type* that defines *semantic values*.
411. **Semiotics:** Semiotics, as used by us, is the study and knowledge of *pragmatics*, *semantics* and *syntax* of language(s).

412. **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*.)
413. **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 [179].
414. **Sequential:** Arranged in a sequence, following a linear order, one after another.
415. **Sequential process:** A process is sequential if all its observable actions can be, or are, ordered in sequence.
416. **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.)
417. **Set theoretic:** We say that something is set theoretically understood or explained if its understanding or explanation is based on *sets*.
418. **Shared data:** See *shared phenomenon*.
419. **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*.)
420. **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*.)
421. **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*.)
422. **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*.)
423. **Shared information:** See *shared phenomenon*.
424. **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*.)

425. **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*.)
426. **Sign:** Same as *symbol*.
427. **Signature:** See *function signature*.
428. **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*.)
429. **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*).
430. **Software component:** Same as *component*.
431. **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.
432. **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.)
433. **Software design specification:** The *specification* of a *software design*.
434. **Software development:** To us, software development includes all three phases of *software development*: *domain development*, *requirements development* and *software design*.
435. **Software development project:** A *software* development project is a planning, research and development project whose aim is to construct *software*.
436. **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*).)
437. **Software engineering:** The confluence of the science, logic, discipline, craft and art of *domain engineering*, *requirements engineering* and *software design*.
438. **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.)

439. **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*.)
440. **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*.)
441. **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.
442. **Speed:** Speed, v, is the rate of *motion*, or equivalently the rate of change in *position*: the *distance*, x, traveled per unit of *time*, t, i.e., $v = x/t$.
443. **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.
444. **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` [161], `B & eventB` [2, 3, 44], `CASL` [15, 143, 142], `CafeOBJ` [60, 61], `RSL` [72, 73], `VDM-SL` [36, 69] and `Z` [173, 175, 194, 93].)
445. **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.)
446. **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,

[2] These three areas of concern form three *universes of discourse*.

(again) characterised (and, again, loosely) by their common interest in, or dependency on the domain (requirements, software design). (The three stakeholder groups usually overlap.)

447. **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.
448. **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*.
449. **Statechart:** The Statechart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [83, 84, 86, 88, 85].)
450. **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.)
451. **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).
452. **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*.
453. **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.)
454. **Steradian:** The steradian (symbol: **sr**) is the SI unit of solid angle. It is used to describe two-dimensional angular spans in three-dimensional space, analogous to the way in which the radian describes angles in a plane. ( )
455. **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.)
456. **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).)

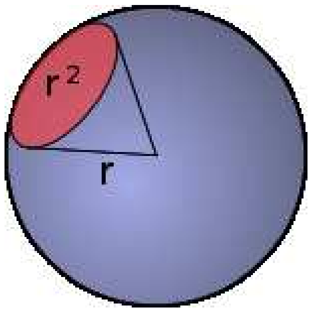

Fig. U.1. A graphical representation of 1 steradian

457. **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: $\{a \mid a : T \cdot p(a)\}$.)
458. **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*.)
459. **Synopsis:** By a synopsis we shall understand a composition of *informative documentation* and *rough-sketch description* of some project.
460. **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.)
461. **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.)
462. **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.)

463. **Systems engineering:** By systems engineering we shall here understand computing systems engineering: The confluence of developing *hardware* and *software* solutions to *requirements*.

$\mathcal{T}$

464. **Taxonomy:** See Sect. U.1.5.
465. **Technique:** A procedure, an approach, to accomplish something.
466. **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.
467. **Temporal:** Of or relating to time, including sequence of time, or to time intervals (i.e., durations).
468. **Temporal logic:** A(ny) *logic* over *temporal phenomena*. (We refer to Vol. 2, Chap. 15 for our survey treatment of some temporal logics.)
469. **Term:** From [123]: 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).)
470. **Terminal:** By a terminal we shall mean a terminal *symbol* which (in contrast to a *nonterminal* symbol) designates something specific.
471. **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*.)
472. **Terminology:** By terminology is meant ([123]): The doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature.
473. **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.)
474. **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.)

475. **Theorem:** A theorem is a *sentence* that is *provable* without assumptions, that is "purely" from *axioms* and *inference rules*.
476. **Theorem prover:** A mechanical, i.e., a computerised means for *theorem proving*. (Well-known theorem provers are: `PVS` [148, 149] and `HOL/Isabelle` [146].)
477. **Theorem proving:** The act of *proving theorems*.
478. **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.
479. **Thesaurus:** See Sect. U.1.7.
480. **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 [189]. We refer to our treatment of time and space in Vol. 2 Chap. 5, to Johan van Benthem's book *The Logic of Time* [187], and to Wayne D. Blizard's paper *A Formal Theory of Objects, Space and Time* [40].)
481. **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.)
482. **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.)
483. **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*.)
484. **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.)
485. **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.)

486. **Transition:** Passage from one state, stage, subject or place to another; a movement, development, or evolution from one form, stage or style to another [179].
487. **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 [154], set a de facto standard for formulating transition rules (exploring their theoretical properties and uses).)
488. **Translate:** See *translation*.
489. **Translation:** An act, process or instance of translating, i.e., of rendering from one language into another.
490. **Translator:** Same as a *compiler*.
491. **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 [179]. (The trilogy of the *phases* of *software development*, *domain engineering*, *requirements engineering* and *software design* as promulgated by this trilogy of volumes!)
492. **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).)
493. **Type:** Generally a certain kind of set of *values*. (See *algebraic type*, *model-oriented type*, *programming language type* and *sort*.)
494. **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*).)
495. **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.)
496. **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*.
497. **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*.
498. **Type name:** A type name is usually just a simple *identifier*.

499. **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*.)

$\boxed{\mathcal{U}}$

500. **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.)
501. **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*.)
502. **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.
503. **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*.)
504. **Update:** By an update we shall understand a change of value of a variable, including also the parts, or all, of a *database*.
505. **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.
506. **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*.)
507. **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.

508. **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*.)

$\mathcal{V}$

509. **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*.)
510. **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 [127] [377] 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.)
511. **Valuation:** Same as *evaluation*.
512. **Value:** From (assumed) Vulgar Latin *valuta,* from feminine of *valutus,* past participle of Latin *valere* to be of worth, be strong [179]. (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.))
513. **Variable:** (i) From Latin *variabilis,* from *variare* to vary; (ii) able or apt to vary; (iii) subject to variation or changes [179]. (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.)
514. **Velocity:** In physics, velocity is defined as the *rate of change of position*. It is a vector physical quantity; both *speed* and *direction* are required to define it. In the SI (metric) system, it is measured in meters per second: (m/s) or ms^-1.
515. **VDM:** `VDM` stands for the `Vienna Development Method` [36, 37]. (`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 [12, 13, 14, 126] of `PL/I`, a programming language of that time. "JAN" (John A.N.) Lee [118] is believed to have coined the name `VDL` [119, 125] for the notation (the Vienna Definition Language) used in those semantics

definitions. So the letter M follows, lexicographically, the letter L, hence `VDM`.)

516. **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.)
517. **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.)
518. **Verify:** Same, for all practical purposes, as *verification*.
519. **Volt:** The volt is defined as the potential difference across a conductor when a current of one *Ampere* dissipates one *Watt* of power.

$$\mathrm{V} = \frac{\mathrm{W}}{\mathrm{A}} = \frac{\mathrm{J}}{\mathrm{C}} = \frac{\mathrm{N}\cdot\mathrm{m}}{\mathrm{A}\cdot\mathrm{s}} = \frac{\mathrm{kg}\cdot\mathrm{m}^2}{\mathrm{A}\cdot\mathrm{s}^3}$$

$\mathcal{W}$

520. **Watt:** The Watt (symbol: **W**) is the SI derived unit of power, equal to one *Joule* of energy per *second*. It measures a rate of energy conversion.
521. **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*!)
522. **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).)
523. **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 [179].

$\mathcal{Z}$

524. **Z:** Z stands for Zermelo (Frankel), a set theoretician. (`Z` also stands for a model-oriented specification language [173, 174, 194, 93, 92].)

V

Indexes

V.1 Index of Methodology Concepts

V.2 Index of Definitions

V.3 Index of Principles

V.4 Index of Techniques

V.5 Index of Tools

V.6 Index of Symbols

V.7 Index of Examples

V.8 Index of Domain Phenomena and Concepts