Dines Bjørner

Software Engineering 3

Domains, Requirements, and Software Design

Springer

Texts in Theoretical Computer Science
An EATCS Series

Editors: W. Brauer G. Rozenberg A. Salomaa

On behalf of the European Association
for Theoretical Computer Science (EATCS)

D. Bjørner

Software Engineering 3

Domains, Requirements, and Software Design

With 100 Figures

Springer

Author

Prof. Dr. Dines Bjørner

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark
2800 Kgs. Lyngby, Denmark
bjorner@gmail.com
db@imm.dtu.dk

Series Editors

Prof. Dr. Wilfried Brauer

Institut für Informatik der TUM
Boltzmannstr. 3
85748 Garching, Germany
brauer@informatik.tu-muenchen.de

Prof. Dr. Grzegorz Rozenberg

Leiden Institute of Advanced
Computer Science
University of Leiden
Niels Bohrweg 1
2333 CA Leiden, The Netherlands
rozenber@liacs.nl

Prof. Dr. Arto Salomaa

Turku Centre of
Computer Science
Lemminkäisenkatu 14 A
20520 Turku, Finland
asalomaa@utu.fi

Library of Congress Control Number: 2006921809

ACM Computing Classification (1998): C.2, C.3, C.4, C.5, D.1, D.2, D.3, F.3, F.4, H.1, J.1, K.6.3

ISBN-10 3-540-21151-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-21151-8 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media
springer.com

Printed in Germany

Cover Design: KünkelLopka, Heidelberg
Typesetting: Camera ready by the Author
Production: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig

Printed on acid-free paper 45/3100/YL 5 4 3 2 1 0

Nikolaj, Marianne, Katrine and Jakob

the gleam in my eye

There are no theories
There are no proofs
There may be bold conjectures
And there will be sad refutations

Free interpretation of Imre Lakatos and Sir Karl Popper [208, 278, 279, 281]

Preface

General

The present volume is but the third of three textbooks on the engineering principles and techniques of software engineering. With these three volumes we claim that we show how formal techniques, also known as formal methods, can be exploited to their fullest in industry-scale development projects. We risk our reputation by going further: We can now justifiably claim that there is no longer any excuse for not using formal techniques throughout all phases, stages and steps of development. Usually such excuses are claimed due to *a lack of a fully comprehensive guide on the use of formal methods in even very-large-scale software developments.* Here is a set of books that tells you how to do most of it in minute detail!

Surely not all development facets are today clarified down to the level of formal techniques that we would wish were available. But to refrain from using what there is is — in our perhaps not so humble opinion — outright criminal! As these volumes, and many excellent monographs, show: there is so much already now available that the arrogance of not using these techniques boils down to, yes, criminal neglect.

Some so-called software engineering practitioners "hang on" to the lack of management guidance. To them I say: Once you have understood the principles and techniques of these volumes, and if you are otherwise a sensible person with some management experience, the rest follows. You, as well as I, can "fill in" the management principles and techniques.

Appendix B of Vol. 1 contains an extensive glossary, and Appendix A of Vol. 2 contains an overview of our naming convention.

Brief Guide to Volume 3

This volume can be studied in a number of ways. Any path — through chapters, that is, nodes of the graph of Fig. 2 — from the input node, labelled 1,

to the output node, labelled 32, can form a course. Let us elaborate briefly on Fig. 2:

Base course on SE: A minimum course covers Chaps. 1, 2, 5, 8, 11, 16, 17, 19, 24–26, 30–32. That is, all the left column chapters of Fig. 2.

Domain engineering: A course focusing on domain engineering would additionally cover Chaps. 9, 10 and 12–15.

Requirements engineering: A course focusing, instead, on requirements engineering would in addition to the base course cover Chaps. 18 and 20–23.

Software design: A course focusing on software design would in addition to the base course cover Chaps. 27–29.

Any of the four courses outlined above can be given in either of two ways:

Informal: In this way of studying this volume the reader can skip the formalisation bits and focus just on the informal material. That is, one can study this volume in principle and in reality without first having studied Vol. 1 or Vols. 1 and 2.

Formal: In this way of studying this volume the reader covers all the informal material as well as the formal material – and thus a study of at least Vol. 1 is a prerequisite for studying the present volume.

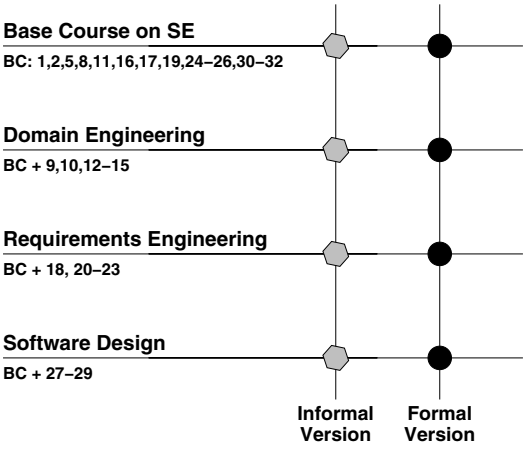


Fig. 1. Course alternatives

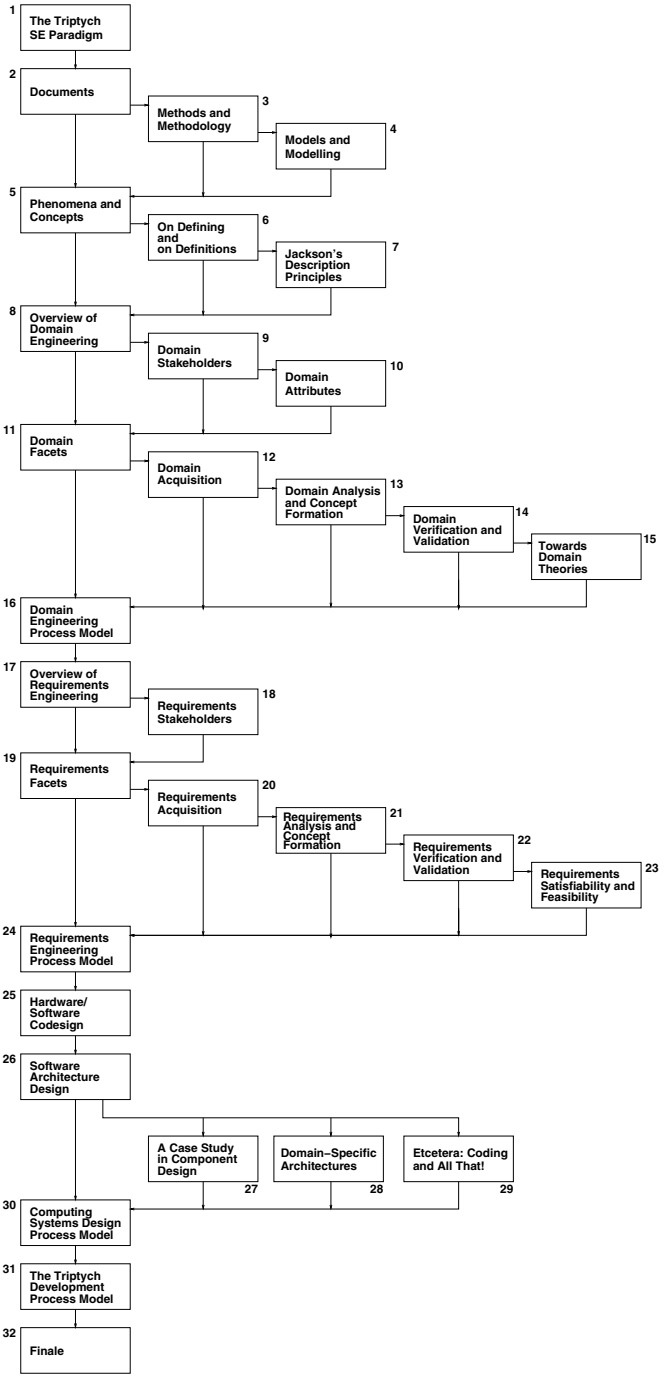
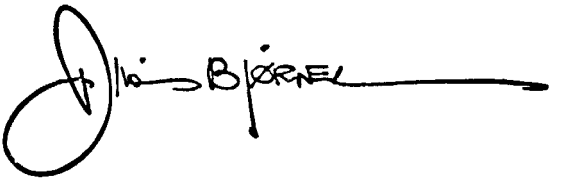


Fig. 2. Chapter precedence graph

Acknowledgments

The acknowledgments of Vols. 1 and 2 carry over to this volume. In addition I wish to acknowledge with gratitude Kirsten Mark Hansen for allowing me to use Chap. 4 of her splendid PhD Thesis [141] in edited form as Sect. 19.6.5. Again I wish to specifically acknowledge the main source of my academic joy over the last almost 30 years, namely my university: the Technical University of Denmark.



Dines Bjørner
The Technical University of Denmark, 2005–2006

Contents

Part II CONCEPTUAL FRAMEWORK

Part III DESCRIPTIONS: THEORY AND PRACTICE

Part IV DOMAIN ENGINEERING

Part VI COMPUTING SYSTEMS DESIGN

Part I

OPENING

Two chapters open this book. Their contents are somehow "orthogonal", that is, the chapters are independent of one another.

1. **The Triptych Dogma:**
   In Chap. 1 we paraphrase the major dogma, not only of this volume, but also of the book (i.e., all three volumes). That major dogma is:
   - Before *software* can be *design*ed, programmed, coded, its *requirements* must first be reasonably well understood.
   - Before requirements can be expressed properly, the *domain* of the application must first be reasonably well understood.

   In order to conduct proper software development:
   - One must first *describe*, informally and formally, the *domain* of the application.
   - Then core parts of the *requirements prescription* must be somehow "derived" from the domain description.
   - Finally the *software design specification* is somehow "derived" from the requirements prescription.

   The present volume outlines how to describe domains, how to prescribe requirements, how to "derive" the latter from the former, how to "derive" software design from requirements prescriptions. Please note our use of the three terms *descriptions*, *prescriptions*, and *specifications*. We use the term description in connection with *domain descriptions*. We use the term prescription in connection with *requirements prescriptions*. And we use the term specification, both in general, and specifically about *software designs*.

2. **Documents:**
   Domain descriptions, requirements prescriptions and software design specifications all amount to textual and graphical documents. In a sense, one could claim that all a software engineer does is to produce (conceive, write and analyse) documents. In Chap. 2 we shall examine the full variety of documents that may, or advisedly must, be produced during software development. We preview this variety:
   - **Informative documents:**
     As the name suggests, these documents inform. They do not describe domains, or prescribe requirements, or specify software design. One may claim, however, that they specify some metafacts, or metaproperties, or metadesigns. To preview the kinds of categories of informative documents we mention, in passing, listing of partners, current situation, needs, ideas (of domain, requirements and/or software), concepts, scope, span, synopsis, contract, and design brief. Section 2.4 discusses principles, techniques and tools for constructing these kinds of documents.
   - **Descriptive/prescriptive/specification documents:**
     The *real meat* of a development, whether of a domain description, of a requirements prescription, of a software design, or of two or all of these, is the descriptive, the prescriptive, respectively the specification, documents. Each of these come in stages and forms. To preview we just mention their category names: rough sketches, terminologies, narratives, and formalisations. Section 2.5 brings in principles, techniques and tools for constructing these kinds of documents – which we, using one term, refer to as descriptive documents.
   - **Analytic documents:**
     A main reason, but not the only one, for using formalisations, is to be able to reason over the texts of domain descriptions, requirements prescriptions or software design specifications. In fact, just plain informal descriptions, prescriptions and specifications do allow the software engineer some "room" for (albeit informal) reasoning. These are better than if no text, other than code, is available. Analytic documents are documents which analyse other — usually description, prescription, or specification — documents. To preview there are basically three kinds of analytic documents: Validation, verification and theory formation documents. Section 2.6 discusses principles, techniques and tools for constructing these kinds of documents.

1

The Triptych Paradigm

- The **prerequisite** for studying this chapter is that you have at least some introductory level programming skills, as, for example, obtained through a year of `Java` or `C#` programming.
- The **aims** are to introduce the basic ideas of *domain engineering*, *requirements engineering* and *software design* as they relate to one another, to introduce the concept of *separation of concerns* as represented here by the concepts of *phases*, *stages* and *steps* of development, and thus to introduce the concept of the *triptych software development process model.*
- The **objective** is to make the reader a professional software engineer with respect to understanding the crucial phases, stages and steps of software development.
- The **treatment** is precise but informal.

1.1 Delineations of Software Engineering

We give two sets of characterisations of the field of software engineering. One set of characterisations is taken from the literature. The other (a singleton) set is our definition.

1.1.1 "Old" Delineations

The term "software engineering" seems to have many meanings. We shall bring in some of the characterisations that are given in previous textbooks as well as from elsewhere.

Friedrich L. Bauer [257], 1968

> Software engineering is the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines.

So we are left to find out what is meant by engineering principles. These "engineering principles" cannot just be those of conventional engineering as we think that the engineering of software is radically different from other engineerings. Conventional engineering builds on the laws of physics. Software engineering builds on mathematics, notably algebra and logic.

Ian Sommerville [338], 1980–2000

> Software engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

In this definition, there are two key phrases:

> (1) *Engineering discipline:* Engineers make things work. They apply theories, methods and tools where these are appropriate but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods to support them. Engineers also recognise that they must work to organisational and financial constraints so they look for solutions within these constraints.
>
> (2) *All aspects of software production:* Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production.

We are getting some engineering principles unveiled, albeit of the conventional kind.

IEEE Std. 610.12–1990 [178]

The IEEE's Standard Glossary of Software Engineering Terminology:

> Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Again, a very conventional engineering characterisation.

David Lorge Parnas

> Software engineering is defined as the multi-person construction of multi-version software.

This is, of course, not all that Parnas has to say about software engineering. As much of his other musings this one is cogent.

Shari Lawrence Pfleeger [275], 2001

Pfleeger [275] (Page 2) has an indirect characterisation:

> As software engineers, we use our knowledge of computers and computing to help solve problems ... identification of problems and of when a computing solution may be appropriate, further analysis of such problems, and synthesis of solutions using method principles, techniques and tools, are ingredients of software engineering.

We are not getting much closer to our claimed difference between conventional engineerings and software engineering. Pfleeger's characterisation is OK, but insufficient.

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli [121], 2002

> Software engineering is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers.

This definition hides the real content in its reference to computer (including computing) science, i.e., the mathematical discipline upon which the software engineers work. But, as for Parnas' characterisation, the Ghezzi/Jazayeri/-Mandrioli characterisation emphasises scale.

Accreditation Board for Engineering and Technology (ABET)

ABET (www.abet.org) gives a definition of 'engineering' which some software engineering authors refer to:

> Engineering is the profession in which a knowledge of the mathematical and natural sciences gained by study, experience and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind.

We take exception, in parts, to the software engineering use of the ABET characterisation: First, to us, software engineering (in almost all its activities) does not rely on laws of the natural sciences (but almost exclusively only on mathematics). Second, *the materials and forces of nature* must therefore be rephrased into *mathematics*. Third, *for the benefit of mankind* is just a (as of the year 2005) politically correct utterance — so we do not include it in our consideration of what software engineering is.

Hans van Vliet [369], 2000

Pages 6–8 of Hans van Vliet's delightful work [369] details the following software engineering facets:

- *Software engineering concerns the construction of large programs.*
- *The central theme is mastering complexity.*
- *Software evolves.*
- *The efficiency with which software is developed is crucial.*
- *Regular co-operation between people is an integral part of programming-in-the-large.*
- *The software has to support its users effectively.*
- *Software engineering is a field in which members of one culture create artifacts on behalf of members of another culture.*

We quite like van Vliet's characterisation.

1.1.2 Our View: What Is Software Engineering?

Many of the above characterisations are relevant. Some characterise software engineering by how it proceeds, others by what it does. We prefer the latter style of characterisation. In order, however, to emphasise a number of new aspects of the software engineering approach that we shall be propagating in these volumes, our delineation combines both the 'how' and the 'what' styles.

We shall therefore characterise the concept of 'software engineering' as follows:

- Software engineering is the establishment and use of sound methods for the efficient construction of efficient, correct, timely and pleasing software that solves the problems such as users identify them.
- Software engineering extends the field of computing science to include also the concerns of building of software systems that are so large or so complex that they necessarily are built by a team or teams of engineers.
- Software engineering is the profession in which a knowledge of mathematics, gained by study, experience and practice, is applied, with judgment to develop ways of exploiting mathematics to (i) understand the problem domain, (ii) the problem and (iii) to develop computing systems, especially software solutions, to such problems as are conveniently solved by computing.
- Software engineering thus consists of (i) domain engineering (in order to understand the problem domain), (ii) requirements engineering (in order to understand the problem and possible frameworks for their solution) and (iii) software design (in order to actually implement desired solutions).

In the next section we shall examine these three concepts: domain engineering, requirements engineering and software design.

1.2 The Triptych of Software Engineering

Before some specific software can be designed and coded, we must understand the requirements that this software must fulfill.

Characterisation. By *design of software* we mean the process, and the documents resulting from the process, of implementing the software. That is, we mean in the narrow sense of conceiving of and expressing (i.e., specifying) these documents — in stages and steps — eventually in some executable programming language. Software design thus specifies *how* executions may proceed. ∎

Characterisation. By *requirements prescription* we mean the process, and the documents resulting from the process, of acquiring, analysing and writing down, in some language, *what* the software — to be designed — is expected to do. ∎

Before requirements can be written down, we must understand the application domain for which the software is to be developed.

Characterisation. By *domain description* we mean the process, and the documents resulting from the process, of acquiring, analysing and writing down, in some language, a model of the application domain *as it is* — void of any reference to requirements to any software for that domain, let alone references to such software. ∎

So, from descriptions of the application domain we construct prescriptions of the requirements; and from prescriptions of the requirements we design the software, i.e., construct specifications of software. Ideally speaking we would wish to proceed from describing the application domain, via prescribing the requirements, to implementing the software. Actual life sometimes forces us, and always permits us, to iterate between these three phases of software development.

1.2.1 On Universes of Discourse and Domains

Above we have used the term "application domain" without explaining what we mean by that term. We shall now explain that term as well as the more general term 'universe of discourse' and the simpler term 'domain'.

Characterisation. By a *universe of discourse* we shall understand anything that can be spoken about. ∎

Example 1.1 *Universes of Discourse:* We shall take the view here that there are basically three classes of universes of discourse:

(i) The definite, but singleton class of software engineering as an intellectual concept, i.e., software development in general and programming in particular. So, domain engineering, requirements engineering and software design could each, or as a whole, be a universe of discourse. These volumes take this intellectual concept of software engineering as its universe of discourse. As an intellectual concept the software engineering universe of discourse is not an application domain.

(ii) The indefinite class of "things" to which computing may be applied, i.e., application domains (see next).

(iii) The infinite class of anything else that does not satisfy the above characterisations. Examples are: philosophy, politics, poetry, etc.[1] ▪

Characterisation. By an *application domain* we shall understand anything to which computing may be applied. ▪

Example 1.2 *Application Domains:* We shall take the view that there are basically three classes of application domains:

(i) The class of applications which can be characterised as supporting the teaching or study of a subject field: educational or training software, respectively experimental software for theorem proving, or the like.

(ii) The class of applications which can be characterised as supporting the development of computing systems themselves: compilers, operating systems, database management systems, data communication systems, etc.

(iii) And the class of applications which can be characterised as not supporting the development of computing systems themselves, but that of business, or industry software.

It is basically the last class of these application domains that are of interest in this volume. We relegate to specific textbooks the treatment of special principles, techniques and tools for the development of software for application domains (i) and (ii). Classes (i) and (ii), in a sense, overlap. Class (i) is perhaps better viewed as a knowledge engineering topic, while class (ii) is conventionally seen as a systems software topic. Viewed these ways, class (iii) is then normally seen as an "end-user", i.e., "customer software" topic. ▪

Characterisation. By a *domain* we mean an application domain. ▪

[1] The reader may observe two things: Our inability to make precise that to which computing cannot be applied, and our belief that philosophy, politics, poetry, etc., belong to that class. When we claim that computing does not apply to philosophy, politics, poetry, etc., we mean that crucial philosophical thoughts, political ideas and poetic utterings cannot, in our mind, be the result of computations.

That is, the two terms "application domain" and "domain" are taken to be synonymous.

Example 1.3 *Domains:* We continue our exemplification of (application) domains — of the third class mentioned just above.

(iii.1) The applications of software within the transportation sector: Railways, airlines, shipping, public and private road transport (buses, taxis, trucks, automobiles in general), etc., individually define application domains, and together "define"[2] *transportation* as a domain.

(iii.2) The applications of software within the financial services sector: banks, insurance companies, securities trading (stock and bond exchanges, traders, brokers), portfolio and investment management, venture capital companies, etc., individually define application domains, and together "define" the *financial service industry* as a domain.

(iii.3) The applications of software within the healthcare sector: hospitals, family doctors (i.e., private, practicing physician), pharmacies, community nurses, retraining and convalescent clinics, the public health authorities, etc., individually define application domains, and together "define" *healthcare* as a domain.

(iii.4) The applications of software within the machining (metal-working) manufacturing sector: marketing, sales and orders department, design research and development, production planning department, the production "floor" with its "input" production parts inventory and its "output" products warehouse, workers, managers, etc., individually define application domains, and together "define" *machining (metal-working) manufacturing* as a domain. ∎

1.2.2 Domain Engineering

We bring in an overview of domain engineering. Part IV, Chaps. 8–16, bring in details!

General

In this section we shall give a brief characterisation of what we mean by domain engineering.

Characterisation. (I) By a *domain description* we shall understand a description of a domain, that is, something which describes observable phenomena of the domain: entities, functions over these, events and behaviours. ∎

Characterisation. (II) By *domain description* we shall also mean the process of domain capture, analysis and synthesis, and the document which results from that process. ∎

[2] Here our use of "define" indicates that we are not formally defining the subject term. We are merely giving a rough characterisation.

Characterisation. By *domain engineering* we mean the engineering of domain descriptions, that is, of their development: (i) from domain capture and analysis (ii) via synthesis, i.e., the domain description document itself, (iii) to its validation with stakeholders and its possible theory development. ▪

So what does it mean to understand the application domain? To us it means that we have described it. That the description is consistent, i.e., does not give rise to contradictions, and that the description is relatively complete, i.e., does describe "all the things" needed to be described.

What Do We Expect from a Domain Description?

What must we expect from a domain description? We expect that it describes the application area *as it is*.

What a Domain Description Does Not Describe

To us "as it is" means that we have described it without any reference to requirements to any new computing system (i.e., software), let alone to any (implementation, etc.) of such a new computing system (i.e., software). The above was expressed in terms of *what a domain description does not contain*.

What a Domain Description Does Describe

So what does a domain description contain? To us a domain description contains: descriptions of the *phenomena* that can be observed, that can be physically sensed, in the domain, and descriptions of the *concepts*, i.e., the abstractions that these phenomena "embody".

Domain Phenomena and Concepts

What are the phenomena and concepts alluded to just above? To us these phenomena and concepts are such as: (i) entities, (ii) functions, (iii) events and (iv) behaviours. We overview these four categories of phenomena and concepts.

Entities

Entities are "things" that one can point to, things that typically become data "inside" a computer, things that are to have a *type* and a *value* (of that type).

Example 1.4 *Entities:* For a domain of harbours some typical entities are: ships, holding area(s) where ships may wait for a buoy or a quay position, buoys, quay positions and cargo storage areas. The harbour can be considered an entity composed from the above.

Informal Presentation: Entities: Types, Values and Observers

We shall later explain the notation now used:

type
Harbour, Ship, HoldArea, Buoy, Quay, CSA, Position
value
obs_Ships: Harbour → Ship-**set**
obs_HoldAreas: Harbour → HoldArea-**set**
obs_Buoys: Harbour → Buoy-**set**
obs_Quays: Harbour → Quay-**set**
obs_CSAs: Harbour → CSA-**set**
obs_Position: Ship → Position-**set**
obs_Position: Quay → Position-**set**
obs_Position: Buoy → Position-**set**
obs_Position: HoldArea → Position-**set**

From a harbour one can observe all the

- ships in the harbour,
- holding areas of the harbour,
- buoys of the harbour,
- quay positions of the harbour and all the
- container storage areas of the harbour.

Positions are associated with

- ships,
- holding areas,
- buoys and
- quays.

One may rightfully argue, as we shall later do, that ships be modelled as behaviours, not as entities. In fact, all the entities listed above could be so considered. What are we to make of that? That is, why model, as here, these phenomena as entities? The answer is, in this case, simple: if and when we model them, instead, as behaviours, then the entity types and values otherwise alluded to above will become configuration (context and state) attributes. ∎

Our unfolding story of entities, functions, events and behaviours, will be treated in far more detail in Chap. 5.

Entities are values and values are of some type. That is, a type stands for a usually infinite set of values. Some entities may be considered atomic.

They have no proper subentities. Other entities may be considered composite. One may speak of proper subentities. Entities have attributes, that is, are of types. Atomic entities may have composite types, like Cartesians of several attributes. An example is a person. A person has a name, a birth-date and a gender. These are different types, i.e., different attributes and are of constant value — usually. A person also has a current (i.e., variable) weight and height. Composite entities naturally have composite types. Some of these types describe proper subentities. Others describe how the subentities are composed into the overall, the composite entity. One can observe from an entity the values of its attributes. Thus one can observe from a composite entity the set or sequence or Cartesian components, etcetera, of its subentities.

Informal Presentation: Entities: Types, Values and Observers

We sketch and explain the following formal text:

type
  A, B, C, ..., P, Q, R, ...
value
  a:A,

  obs_B: A → B
  obs_C: B → C
  ...
  obs_Ps: B → P-**set**
  obs_Ql: C → Q*

Type names A, B, C, ..., P, Q, R, ..., are here thought of as abstract types, i.e., sorts. The value "declaration" **values** a:A non-deterministically selects an arbitrary value from type A and names this value (a). Such value declarations correspond to the informal uttering, or writing, *let a be an entity of type* A, ... The *observer* function obs_B applies to values of type A and yields a value of type B. The *observer* function obs_C applies to values of type B and yields a value of type C. The *observer* function obs_Ps applies to values of type B and yields a set of values of type P. The *observer* function obs_Ql applies to values of type B and yields a list of values of type Q. These postulated observer functions correspond to the informal uttering, or writing, of, for example, *from an entity of type* B *one can observe a set of (possibly zero) one or more entities of type* P, ... Postulating a few or several observer functions over values of some sort does not prevent such values containing, i.e., being composed from other subentities, or having other attributes.

In Vol. 1, Sect. 5.2 we treat the concepts of phenomenological and conceptual entities, their atomicity and composition, their types and attributes and their values in detail. Vol. 1 is concerned with formal techniques for basic abstractions and models of phenomena and concepts. Vol. 2 is concerned with formal

techniques for the specification of systems of entities, functions, events and behaviours, and of languages over these. The coverage of entities, types and values of the present chapter will be greatly expanded upon in this volume in Chaps. 5–7, 10, 11, and so on.

Functions

Phenomena or concepts could be *functions* that apply to entities and (i) either test for some property, (ii) observe some subentity, i.e., yield a data value that is "computed" from such entities, or (iii) actually change the entity value — in which case we call the function an operation, or an action.

Example 1.5 *Functions:* For a domain of harbours some typical functions are:

(i) An arriving ship asks the harbour whether it can be allocated either a holding area, a buoy or a quay position.

value: inquire: Ship×Harbour → **Bool**

(ii).1 An arriving ship which can be allocated a holding area, a buoy, or a quay position requests the position.

value: request: Ship×Harbour → Position

(ii).2 For a ship destined for a quay position one needs to know how many containers to unload to and how many to load from the harbour:

value: unload_load_quantities: Ship×Harbour → **Nat**×**Nat**

(iii) A ship [un]loading some cargo.

value: [un]load: Ship×Quay → Ship×Quay

The functions just listed were, as we shall call it, roughly sketched, but given a signature. Now we give a more satisfactory narrative. Function (i) takes two arguments: a ship and a harbour. It yields whether or not the ship can be received by the harbour. Function (ii).1 takes two arguments: a ship and a harbour. It yields either a holding area, a buoy or a quay position identification. Function (ii).2 takes two arguments: a ship and a harbour. It yields a pair of natural numbers (u, ℓ) indicating that u containers are to be unloaded and ℓ containers are to be loaded. Function (iii) takes two arguments: a ship and a harbour. It yields the same type doublet, but now the ship is [less] plus some cargo, and the quay now is [plus] less that cargo. ∎

Informal Presentation: Function Signatures

We define three *sorts*, i.e., *abstract types*, and give the *signature* of four functions:

type
(0) A, B, C
value
(1) inv_A: A → **Bool**
(2) obs_B: A → B
(3) gen_C: B → C
(4) chg_B: A × B → B

(0) A, B and C are sorts, i.e., further unspecified abstract types. (1) inv_A is a predicate: it is supposed to yield **true** for well-formed values of A, **false** otherwise. (2) obs_B is intended as an observer function: from values a of sort A it observes, i.e., extracts, values of type B that are somehow "contained" in a. (3) gen_C is intended as a generator function: from values of sort B it computes values of type C. (4) chg_B is intended as an operation (i.e., a generator function): from Cartesian values over A and B it generates values of type B intended to replace the argument of type B. One might thus write any of the below:

[5] **variable** b:B := ...; ... b:=chg_B(a,b) ...
[6] **let** b′=chg_B(a,b) **in** ...; b:=b′; ... **end**
[7] **let** b′=chg_B(a,b) **in** ... gen_C(b′) ... **end**

Example 1.6 *The A, B, Cs of Ships and Harbours:* The reader is asked to complete this example, that is, to relate the types and functions of Example 1.5 to the sorts and functions of the box above! ∎

Events

Events happen, i.e., occur. And when events occur they do so instantaneously. Events may convey information, i.e., have significance other than just occurring. We can speak of external events and of internal events. External events occur in an outside environment, "around" the part of the domain being considered — i.e., interfacing with it — and are being communicated to that part. Or external events occur within the domain being considered, and are being communicated to "somewhere" outside the part of the domain being considered. Internal events occur in one part of the domain being considered and are destined for, i.e., communicated to, another part of the domain being considered — in which case we consider those parts as belonging to different behaviours.

Example 1.7 *Events:* For a domain of harbours some typical events are: a ship arrives at a harbour; a ship declares itself ready to unload or to load; a ship and a quay engage in the events of unloading and loading; a ship declares itself ready to depart a holding area, or a buoy or a quay position. ▪

Informal Presentation: Events

In RSL we may model events in terms of RSL/CSP inputs/outputs:

```
type
  ShipId, ShChar, HAPos, BuoyPos, QuayPos
  MSG == mkArrive(shid:ShipId,shchar:ShChar)
       | mkHoldArea(p:Pos)
       | mkBuoy(b:BuoyPos)
       | mkQuay(q:QuayPos) | ...
  ArrDep == ready | depart
  Cargo

channel
  sh,hs:MSG
  sqr:ArrDep
  sq,qs:Cargo
value
  ship(...) ≡
    ... sh!mkArrive(si,sc) ... let pos = hs? ... end ...
    ... sqr!ready ... sq!c ... let c′ = qs? ... end ...
  harbour(...) ≡
    ... let mkArrive(s,c) = sh? ... hs!mkQuay(q) ... end
  quay(...) ≡
    ... if sqr?=ready then let c = sq? ... qs!c′ ... end else ... end
```

Here we have just sketched some events: the arrival of a ship, the harbour message of (as here, quay) position, the ship signalling readiness, the ship unloading some cargo (c), the ship taking aboard some cargo (c'), and the corresponding events in the harbour and quay behaviours.

Behaviours

Some *phenomena* (or *concepts*) are thought of as *behaviours*. They proceed, typically in time, by performing functions (actions), generating or responding

to *events* and otherwise *interacting* (i.e., *synchronising* and *communicating*) with other behaviours.

Formal Presentation: Behaviour

We sketch and explain the following specification text:

```
type M
channel t_p,t_q : Bool, k : M

value
   P() ≡
   p:while t_p? do
      action_p1;
      k ! v
      action_p3
   end

Q() ≡
variable w:M;
q:while t_q? do
   action_q1;
   v := k ? in
   action_q3
end

R() ≡ P() || Q()
```

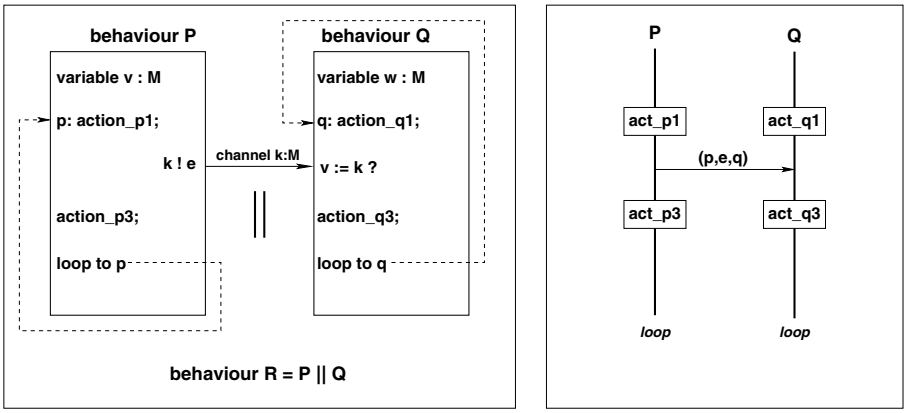


Fig. 1.1. Informal process diagrams

Formal Presentation: Behaviour

We explain the specification text to the left in Fig. 1.1: **type** M is the type of the messages to be sent from **behaviour** P to **behaviour** Q. **Channel**s t_p, t_q are oracles. They determine cyclic behaviour of processes P and Q. **Channel** k is the medium by means of which messages are to be conveyed between behaviours. **Behaviour**s P and Q — in this simple example — take no arguments, hence (). And they are both cyclic: after having honoured their only three **action**s they **loop** back to their first action. The

only really interesting pair is the pair of **input/output** actions. Output `k ! e` prescribes that the value of expression e be communicated on channel k before the computation proceeds to the next action. Input `v := k ?` prescribes that a computation inspects channel k to see whether there is an available message. Once a message has been received it is assigned to the variable v. Two possibilities now exist: either the process named by P awaits that the message it has put on channel k is consumed by something — here the process named by Q. Or the process named by P does not wait for the message it has put on channel k to be consumed. In the former case we say that the channel is a 0-capacity buffer; messages are *not persistent*. In the latter case we say that the channel provides for an infinite-capacity buffer, then messages are *persistent*. When sketching, i.e., presenting programs, like the above, the programmer cum specifier must state whether the channel provides for nonpersistence or persistence. In `RSL/CSP` channels are nonpersistent. If channels are nonpersistent, i.e., are 0-capacity buffers, then we say that the fact that the computation based on behaviour P does not proceed to its next action before the computation based on behaviour Q has consumed the message (sent by Q) constitutes a *synchronisation*. In either case, the message being transferred constitutes a *communication*. We shall usually use the term behaviour in favour of process. However, when a behaviour (or rather, a set of behaviours) is implemented by (i.e., exists inside) the computer, we shall also call it a *process*.

To the right in Fig. 1.1 we have shown an MSC (message sequence chart). The *loop* annotation is strictly speaking outside the proper MSC syntax.

We refer to Vol. 1, Chap. 21 for a thorough coverage of `CSP` [168,301,311] and `RSL/CSP`. And we refer to Vol. 2, Chap. 13 for a thorough coverage of MSC [182–184].

Example 1.8 *Railway Entities, Functions, Events and Behaviours:* Our example derives from railways. Example railway entities are: (i) the railway net (N), (ii) its lines (L), (iii) its stations (S), (iv) the units (U) of the net into which it can be decomposed (linear, switches, crossovers, etc.), etc.

Formal Presentation: Railway Entities

```
type
   N, L, S, U
value
   is_Linear, is_Switch, is_Crossover: U → Bool
```

An example railway function is: (v) the issuance of a ticket in return for the monies it costs. The function issue takes monies (Mo), from-station (Sn), to-station (Sn), date (Da), train number (Tn) and the state of all train reservations

(TnRes) as arguments and delivers a ticket (Ticket) and an updated state of all train reservations as results.

Formal Presentation: Railway Functions

```
type
   Mo, Sn, Da, Tn, TnRes, Ticket
value
   issue: Mo × Sn × Sn × Da × Tn → TnRes → TnRes × Ticket
```

(vi) An example railway behaviour is: passengers getting on a train, at a station platform; followed by the departure of the train from the station platform; the ride of the train down the line towards the next station, including the acceleration and deceleration of the train along the line; the arrival of the train at the next station, and subsequently its stopping at a platform; and the alighting of passengers at that platform.

Formal Presentation: Railway Behaviours

```
type
   Sn, Train
value
   train_ride: Sn* → N → Train → N × Train
   train_ride(snl)(net)(trn) ≡
      if len snl ≤ 1
         then
            (net,trn)
         else
            let (net′,trn′) = get_on_train(hd snl)(net)(trn);
            let (net″,trn″) = train_dept(hd snl,hd tl snl)(net′)(trn′);
            let (net‴,trn‴) = ride(hd snl,hd tl snl)(net″)(trn″);
            let (net⁗,trn⁗) = arriv_and_stop(hd tl snl)(net‴)(trn‴);
            let (net⁗⁗,trn⁗⁗) = get_off_train(hd tl snl)(net⁗)(trn⁗);
            train_ride(tl snl)(net⁗⁗)(train⁗⁗)
      end end end end end end
```

We explain the formula above: Sn* denotes lists of station names. We assume that the net also registers passengers at stations. train_ride is based on a list of two or more station names, a railway net (with lines and stations) and a train. If the list of station names is of length less than two, the train ride is finished. getting_on_the_train is assumed to accept passengers from the first station of the station list, whereby both the net and the train states change. departure_of_train models the train ride inside the station named first in the station name list. Again, the departure, from the platform to the

beginning of the line going to the next station, changes both the net and the train states. ride models the line part of the journey between neighbouring stations, including acceleration, etc. arrival_and_stop models the train ride inside the station named second in the station name list. Again, the arrival, from the end of the line into this next station to its platform, changes both the net and the train states. getting_off_the_train is assumed to model the alightening of passengers at the second station of the station list, whereby both the net and the train states change. Finally, the train_ride resumes as from this station onwards.

▪

Formal Presentation: Some Comments

The above behaviour was expressed purely functionally, with references only to simple mathematical functions. That is, these functions are all to be thought of as executing instantaneously. So what is their temporal behaviour, one may very well ask? It is the set of sequences of actions and events denoted by the function definitions. In Vol. 1, Chap. 21, Sect. 21.2.3 we explain what we mean by a trace semantics of behavioural specifications. Temporality is exhibited by orderings of these actions and events. One may, however, read the above formula as if each function took some not-further-specified time to execute, i.e., to be applied. Thus you may trick yourself into believing that the formulas prescribe a timed behaviour.

Example 1.9 *Railway Functions:* We continue Example 1.8.

Formal Presentation: Railway Functions and Behaviours

Next we give a set of definitions of functions. These evolve around channels and function definitions with synchronisation and communication between functions. We may then claim that this formalisation more properly describes a behaviour. We have tried to make the two formalisations, the above and the below, as similar as possible.

```
type
  P, SIdx, Tn, Σ, Train
  mTn == mkTn(t_n:Tn)
  mPs == mkPs(p_s:P-set)
channel
  { c[s]:(mTn|mPs) | s:SIdx }
value
  obs_Tn: Train → Tn

  passengers: Tn → Σ → Σ × P-set
  passengers: Train → SIdx → P-set
```

```
passengers: Train → P-set


station(s)(σ) ≡
  let tn_or_ps = c[s] ? in
  case tn_or_ps of
    mkTn(tn) →
      let (σ′,ps′) = passengers(tn)(σ) in
      c[s] ! mkPs(ps′);
      station(s)(σ′) end,
    mkPs(ps) →
      station(s)(merge(ps)(σ))
  end end

merge: P-set → Σ → Σ


train(sl)(τ) ≡
  if len snl ≤ 1
    then
      skip /* assert: passengers(τ) = {} */
    else
      let s = hd sl in
      c[s] ! mkTn(obs_Tn(τ));
      let mkPs(ps) = c[s] ? in
      let τ′ = seat(τ)(ps) in
      let τ″ = leave(τ′)(s) in
      let τ‴ = ride(τ″)(s,hd tl sl) in
      let τⁱᵛ = arrive(τ‴)(hd tl sl) in
      let (τᵛ,ps′) = passengers(τⁱᵛ)(s);
      c[s] ! mkPs(ps′);
      train(τᵛ)(tl sl)
  end end end end end end end end
  assert: tn = obs_Tn(τ) = ... = obs_Tn(τᵛ)


seat: Simple function
leave: Behaviour − communicates with station rail net
ride: Behaviour − communicates with line rail net
arrive: Behaviour − communicates with station rail net
```

■

We shall treat the concepts of phenomena and concepts in Chapter 5. Suffice it for now to justify the above remarks as follows. Entities typically are manifest;

that is, they exist in time and space. Functions can be conceived through their effects, but cannot, in and by themselves, be observed. *Nobody*[3] *has ever seen the number which we may represent by any of the numerals* 7, vii, seven, |||||||, III, *etc.* Even more so for behaviours: we may observe a progression of changing entities, effects of function applications and events; but we cannot "see" the behaviour, only conceive of it! The same is true for events.

Further Expectations from Domain Descriptions

What else must we expect from a domain description? Although we shall review domain engineering in Section 1.3.3, and treat domain engineering in detail in Chaps. 8–16, we shall just mention a few things.

(i) We expect a domain description to be readable and understandable by all stakeholders of the domain, i.e., by all those people who "populate" the domain.

(ii) We expect a domain description to be the basis for learning about the domain, that is, for education about and training in the domain — say, for such people as are being hired into a job in the domain, or for such people that need services offered by the domain.

(iii) We expect a domain description to be the basis for constructing a major part of the requirements, namely that part which we shall call the *domain requirements*.

(iv) And we expect a domain description to be a basis for what — in other contexts than software engineering — is known as *business process reengineering*. We shall cover issues of business process reengineering in Chap. 11's Sect. 11.2.1.

Domain Descriptions as Bases for Domain Theories

Physicists have spent the last 400 years studying nature. Traditional engineering disciplines, such as civil engineering, mechanical engineering, chemical engineering, electrical engineering, and electronics engineering, all build on various theories of physics and chemistry. The engineering artifacts, that such engineers build, embody, so-to-speak, fragments of these theories.

For the class of application domains that was characterised as being end-user, public administration and institution oriented, as well as business and industry oriented, for that class of human-made universes of discourse we cannot refer to any such similar theories!

Isn't it about time that we develop theories, such as physicists have done, for respective application domains? This author thinks so.

With the principles and techniques of domain engineering the reader will be well prepared to help contribute research and development on such a theory.

[3] Although the analogy should, more properly, be to a function, it is here to another mathematical thing, namely a number.

But to do it properly, the reader needs to learn additional principles, formal techniques and related tools.

More on Domain Engineering

We have briefly previewed some domain concepts, there are many more. For domain engineers to know how to proceed, what to do, and how to do it, and do it professionally, with assurance, it is important that they know what domain engineering entails. In particular they must know what should be, and not be, in a domain description document, that is, its parts and structure. We shall cover these and other domain engineering concepts a little more in Section 1.3.3, and in detail in Chaps. 8–16.

1.2.3 Requirements Engineering

In this section we shall give a brief characterisation of what we mean by requirements engineering.

The Machine

We introduce the term machine.

Characterisation. By *machine* we shall understand a combination of hardware and software that is the target for, or result of, computing systems development. ▪

Discussion. Although the title of these volumes is *Software Engineering* we cannot avoid also dealing with the engineering of the hardware aspects of the computing system for which a domain description is first established, for which requirements are then developed, and for which subsequent software design is finally sought — or completed. That is, although the main development actions may have to do with software, there will necessarily be a hardware design component in that development. The software "resides" on, or "in" some hardware (computer); the software thus relies on that computer and its peripherals having certain minimal properties, etc. So the requirements shall stipulate properties, not only of the software, but possibly also of the hardware. ▪

The objective of writing down requirements is to prescribe desired properties of a machine: the software and the hardware on which the software resides.

The Machine Environment

We introduce the concept of the environment of a machine.

Characterisation. By *machine environment* we shall understand the rest of the world. More specifically, we mean those parts of the world which interface to the machine: its users, whether humans or technology. ∎

Discussion. The concept of machine environment is a fuzzy one. Ideally the machine environment includes all the stakeholders of the machine, that is, of the new services (functions and facilities) offered by the machine, as well as all the nonhuman interfaces to the machine: monitored and controlled phenomena of the world "surrounding" the machine. But, to predict, in advance of establishing the requirements to the software to be designed, and in advance of the actual installation of that software, which are to be "all" these stakeholders and "all" those affected phenomena, is an art; it is not easy. ∎

So the objective of writing down requirements is also to delineate, to decide upon and distinguish between what is to "belong" to the machine, and what is to "belong" to the environment.

General

Characterisation. By *requirements* we mean a document which prescribes desired properties of a machine: *what* the machine shall (must, not should) offer of functions and behaviours, and what entities it shall maintain. ∎

Characterisation. By a *requirements prescription* we mean the process — and the document which results from the process — of requirements capture, analysis and synthesis. ∎

Characterisation. By *requirements engineering* we understand the engineering, that is, we understand the development of requirements prescriptions: from requirements prescription via the analysis of the requirements document itself, its validation with stakeholders and its possible theory development. ∎

Different Kinds of Requirements

We see four different kinds of requirements: (i) *business process reengineering*, (ii) *domain requirements*, (iii) *interface requirements*, and (iv) *machine requirements*.

Conventionally the following terms are in circulation: *systems requirements*, which approximately covers our overall requirements, *user requirements*, which approximately covers our domain and interface requirements, *functional requirements*, which approximately covers our domain and interface requirements and *non-functional requirements:* approximately covers our machine requirements.

More on Requirements Engineering

We have briefly previewed some requirements concepts. There are many more. For the requirements engineer to know how to proceed, what to do, and how to do it, and do it professionally, with assurance, it is important that that engineer knows what requirements engineering entails, in particular: what should be, and not be, in a requirements prescription document: its parts and structure.

We shall cover these and other requirements engineering concepts a little more in Section 1.3.4, and in detail in Chaps. 17–24.

1.2.4 Software

Characterisation. By *software* we understand (i) not only *code* that may be the basis for executions by a computer, (ii) but also its full *development documentation*: (ii.1) the stages and steps of *application domain description*, (ii.2) the stages and steps of *requirements prescription*, (ii.3) and the stages and steps of *software design* prior to code, with all of the above including all *validation* and *verification* (including *test*) *documents*. In addition, as part of our wider concept of software, we also include (iii) a comprehensive collection of *supporting documents:* (iii.1) *training manuals*, (iii.2) *installation manuals*, (iii.3) *user manuals*, (iii.4) *maintenance manuals*, and (iii.5, iii.6) *development* and *maintenance logbooks*. So, software, as documentation, comprises many parts. ▪

1.2.5 Software Design

From an understanding of what software syntactically, i.e., as documents, "is", we can go on to characterise pragmatic and semantic aspects related to software.

Characterisation. By *software design* we understand the process, as well as all the documents resulting from the process, of turning requirements into executable code (and appropriate hardware). ▪

We make a distinction between two kinds of abstract software specifications, and hence their designs: the software architecture, and the component structure. After a brief presentation of these we shall comment on their nature.

Software Architecture and Software Architecture Design

Characterisation. By a *software architecture* we mean a first specification of software, after requirements, that indicates *how* the software is to handle the given requirements in terms of components and their interconnection — though without detailing, i.e., designing these components. ▪

Characterisation. By a *software architecture design* we mean the development process of going from existing requirements and possibly some already designed components to the software architecture — producing all appropriate architecture documentation. ▪

The term *component design* is used here in a perhaps confusing sense: In architecture design we may certainly identify components, delineating their "boundaries"[4], but leaving their "internals" undefined.[5]

Component Structure and Component Design

Characterisation. By a *component structure* we mean a second kind of specification of software — after requirements and software architecture — one which indicates *how* the software is to implement individual components and modules. ▪

Characterisation. By *component design* (I) we mean the development process of going from existing requirements and a software architecture design to the detailed component modularisation — producing all appropriate component and module documentation. ▪

Software Architecture Versus Component + Module Structure

We are not saying that one must first design the software architecture, and thereafter the component plus module structure. We are presently leaving their order of development — and one of the two, or even a "mix" of them — unexplained!

Modules, Components and Systems

The principle of grouping programming text into modules and collections of modules into components is both old (for modules since the late 1960s, e.g., Simula 67 [30]), and new (for components since the early 1990s).

Characterisation. By a *module specification* we shall understand a syntactic construct, i.e., a structure of program text, which, as a unit of program text, defines what we shall otherwise also call an *abstract data type:* namely a collection of data values and a collection of functions (i.e., operations) over these. ▪

[4] When reading this book in the formal version: Defining their types only as sorts and defining their functions only by giving signatures.

[5] When reading this book in the formal version: That is, their concrete type counterparts, respectively the function definition bodies, undefined. Thus architecture design may "design" part of the component structure.

Discussion. So, by an *abstract data type*, i.e., a *module* we mean a set of data values and a set of procedures (routines) that apply to such data values and yield such data values. Typically *module specifications* are of the following schematic form:

module m:
  types
    t1 = te1, t2 = te2, ..., tt = tet;
  variables
    v1 **type** ta := ea, v2 **type** tb := eb, ..., vv **type** tc := ec
  functions
    f1: ti → tj, f1(ai) ≡ $\mathcal{C}_1$(ai)
    f2: tk → tℓ, f2(ak) ≡ $\mathcal{C}_2$(ak)
    ...
    fn: tp → tq, fn(ap) ≡ $\mathcal{C}_1$(ap)
  hide: fi, fj, ..., fk
end module

(1) *Syntactically* the idea of the above is roughly as follows: m is the name of the module. The module defines t types: $t1, t2, \ldots, tt$, local to that module. A type definition has a left-hand side name ti, and a right-hand side type expression tei. These right-hand side type expressions could be such things as `integer`, `real`, `Boolean`, `character`, `record` structures over types, `vector` structures over a type, and so on. The module also declares v variables: $v1, v2, \ldots, vv$, local to that module. A variable declaration consists of three parts: the variable name, the type of the data values that the variable is allowed to contain and an initialising expression e (something). The module then defines n functions (procedures, routines, operations, methods, or whichever name you wish to call them). Each function definition has two parts: a *function signature,* and a *function definition body.* The function signature defines the name of the function, f, and the type of argument and result values — those types, t, t', that are mentioned on either side of the function space symbol →. The function definition body consists of three parts: the function name followed by a *formal parameter list*, a *function definition symbol*, say, as here, ≡, and the *function body* — here schematised in the form of the abstract clause $\mathcal{C}$(a). This clause can stand for an expression, or a list of statements, or whatever your favourite programming language allows. The module finally lists those function names which are local, i.e., which cannot be referred to by program texts outside the defining module.

(2) The above described, to some — albeit incomplete — extent, the syntax of a typical kind of module. We shall explain the *semantics,* i.e., the meaning, of a module, by just saying: You are assumed to know the meaning of type definitions, of variable declarations, and of function (procedure, method, etc.) definitions. So what's new? The new "thing" is the "encapsulation", the structuring, the "putting together" of these program text structures in a

module structure — one beginning with the keyword **module** and one ending with the keyword **end module.** The meaning of this encapsulation is — again roughly speaking — as it changes "slightly" from actual programming language to actual programming language — that all types, all variables, all function definition bodies, and some function names and signatures are hidden, that is: Their definition cannot be known by program texts outside the defining module. Thus one can substitute one module text by another as long as the function signatures that were visible from outside remain the same in the new, the replacement module. This typically means that the signature types of visible functions are not locally defined types.

(3) The *pragmatics* of a module — perhaps its most important distinguishing feature — can be summarised as follows: The programmer has decided, for whatever reasons, to "lump together" some data structures, in the form of typed variable declarations, and some functions over these, to form an abstract data type, while preserving the right to replace the implementation details of this abstract data type with any other that is believed to yield the same abstract data type. Two things are involved here. First, we have the *hiding* of *implementation details*, that is, (i) the local types, the local variables, the auxiliary functions (those which are hidden) and the bodies of all function definitions. (ii) Second, we have the decomposition of larger program texts into a collection, a usually unordered set, of module definitions. We shall have more to say about this in later sections and chapters. ∎

Characterisation. By a *component specification* we shall usually understand a set of type definitions, a set of component local variable declarations, together defining a component local state, and a set of modules. ∎

The above is just a rough, generic characterisation of components.

Discussion. The idea is that a *component specification* to some surrounding text offers functions of some of its modules. The surrounding text may consist of modules, what we call *initial modules* of a *software system.* ∎

We may suggest a syntax for components:

component
  types: $\mathcal{T}_{i_1}, \mathcal{T}_{i_2}, \ldots, \mathcal{T}_{i_t}$
  variables: $\mathcal{V}_{j_1}, \mathcal{V}_{j_2}, \ldots, \mathcal{V}_{j_v}$
  modules: $\mathcal{M}_{k_1}, \mathcal{M}_{k_1}, \ldots, \mathcal{M}_{k_m}$
  hide: $\mathcal{H}_{\ell_1}, \mathcal{H}_{\ell_1}, \ldots, \mathcal{H}_{\ell_h}$
end component

$\mathcal{T}_i$ suggests some form of type definitions, $\mathcal{V}_j$ suggests some form of variable declarations, $\mathcal{M}_k$ suggests some form of module specifications, and $\mathcal{H}_\ell$ suggests some form of export or hiding of module visible functions (etc.).

Systems, Design and Refinement

Characterisation. By a *software system specification* we shall understand a set of what we shall call *initial* modules together with a set of components — and such that functions of the set of initial modules together invoke functions of modules in the set of components. Systems are what we are developing. ▪

Discussion. The idea is that a *system* is a completely self-contained "item" of software, and that it is composed from components and the *core*, that is, the initial modules. The idea is also that a most abstract level system may be the same as a software architecture, or a component plus initial module structure. ▪

Characterisation. By *software system design* we understand (i) the determination, (i.1) from domain requirements and from some interface requirements, of the software architecture, or (i.2) from machine requirements and from other interface requirements, of the component structuring plus initial modules. Since software architecture design also entails determination of component structuring plus initial modules, we get, more generally, that software system design, in its first stage, i.e., where only the domain description and the requirements prescription exists, entails (ii.1) the determination of the main (system) types of values, (ii.2) the determination of the basic structuring of and facilities (i.e., functions) offered by components, and (ii.3) the determination of such initial modules as are necessary to get the system executing once it is committed. ▪

Usually a first stage of systems design is expressed abstractly, i.e., in a form not suited as a prescription for execution. Hence we need stages and steps of what is called *refinement*:

Characterisation. By *software system refinement* we understand (i) the stagewise and stepwise transformation of (i.1) an abstract specification (i.2) into increasingly more concretely specified modules and components. ▪

Characterisation. By *abstract specification* we mean one that indicates how requirements are to be implemented, but does it by using specification cum programming constructs that are not necessarily efficiently executable. ▪

Characterisation. By *concrete specification* we mean one that uses specification cum programming constructs that prescribe efficient executions. ▪

Components and Modules, Design and Refinement

Characterisation. By *component design* (II) we shall additionally understand the determination of which facilities, that is, which functions (defined

locally "within" the component), and which types (defined globally, i.e., "outside"), the component shall offer. We shall also, by component design roughly speaking mean, the decomposition of the component into modules, and hence, the functions offered by these modules. ▪

One cannot expect a first attempt at component design to succeed in finalising all aspects of an efficient implementation. As will be argued in the next section, separation of concerns makes it easier to tackle many diverse issues. Hence our development needs to proceed in stages and steps of refinement.

Characterisation. By *component refinement* we shall usually understand: (i) a concretisation of the usually initially abstractly defined component types, (ii) a concretisation of the usually initially abstractly specified initialisations of component variables, and, possibly, (iii) the refinement of the component modules. ▪

Characterisation. By *module refinement* we understand: (i) a concretisation of the usually initially abstractly defined module types, (ii) a concretisation of the usually initially abstractly specified initialisations of component variables and (iii) a concretisation of the usually initially abstract module function definitions — (iv) with the latter often entailing the introduction of additional auxiliary (i.e., hidden) function definitions. ▪

Code Design

Finally, we reach the development stage where such program specifications are constructed that can be the basis for efficient execution. We call this kind of program specification 'code'. Since we shall assume the reader to have a necessary background in programming we shall not cover this topic in these volumes.

More on Software Design

We have briefly previewed some software concepts. There are many more. We shall cover these and other software design concepts a little more in Section 1.3.1, and in some detail in Chaps. 25–30. But, these volumes will not present *anywhere near* a fully satisfactory treatment of the software design problem. That is neither the aim nor the objectives of these volumes. First, we have assumed some knowledge, education and training of the reader. Second we have to refer to special topic texts for detailed software design principles, techniques and tools.

1.2.6 Discussion

We have introduced the three main phases of software development:

- Domain engineering in which we describe "what there is"
- Requirements engineering in which we prescribe "what there shall be!"
- Software design in which we specify "how it will be!"

We have indicated, assuming some programming maturity of the readers, some software design structuring — such as revolving around components and modules (with locality and hiding of names), types and variables (abstract and concrete), and program statements and expressions — summarised as clauses ($\mathcal{C}$).

We have not — so far — suggested similar structuring mechanisms for domain descriptions, or for requirements prescriptions. The software design cum programming language structuring constructs of components and modules, of types and variables, etc., aid the developer in knowing "what to do next!" by providing documentation "standards". In the next section we shall preview such textual structuring (decomposition, composition) mechanisms for domain descriptions and requirements prescriptions.

Also, we have intimated, rather loosely, notions of abstract versus concrete software design specifications, and hence we have intimated the entailed notion of refinement. We have not mentioned such stagewise and stepwise mechanisms, for domain descriptions and requirements prescriptions in general, other than for the business process reengineering, and domain, interface and machine requirements stages. Such development stage and step principles are mentioned already in the next section.

1.3 Phases, Stages and Steps of Development

Three Terms

The terms phase, stage, and step, are just that: terms. They are meant to designate basically the same idea: the decomposition of something occurring in time into adjacent, repeated or concurrent intervals. The "something" here is the development of software. The adjacent, concurrent or overlapping intervals are logically and otherwise distinguishable development activities.

A Principle of "Separation of Concerns"

The main reason for decomposing the *software development process* into clearly distinguishable development activities is to tackle separate development issues at separate times, hopefully scheduling these in adjacent, concurrent or overlapping intervals in a fruitful, beneficial way.

In the next several numbered sections we shall briefly review possible decompositions. Each represent a concern; together they represent *separation of concerns*.

Linear, Cyclic and Parallel Development Activities

In the next sections we shall present a view of the software development process as proceeding in strict linear order. Given human nature, such is rarely the case. At the end of this section we shall therefore present two additional views on the software development process: one in which iterations, backwards and forwards, are discussed; and one in which the concurrent tackling of logically separate stages or steps is discussed. By a repeated, or cyclic, interval we mean two intervals, occurring in non-overlapping time periods, in which basically the same item of work is done, i.e., repeated, for example, because a first iteration was not good enough. By concurrent or overlapping intervals we mean two (or more) intervals, in which clearly unrelated work items can be done, independently of one another, hence in parallel.

1.3.1 Phases of Software Development

We have already introduced the main three phases of software development: (i) domain development, (ii) requirements development, and (iii) software design. We have earlier argued for their distinctness, i.e., their focus on truly separate concerns, but we have also emphasised their desirable order, namely as listed.

1.3.2 Stages and Steps of Development

In order to capture a notion of development stage it is important to first capture a notion of the complete documentation — as here — of a phase of development. The documentation for a phase of development is complete if all there is to be documented — at a certain level of abstraction — has been so documented! The idea of "all there is to be documented" is explained in Chap. 2.

It is thus important to also capture a notion of abstract versus concrete documentation — as here — of a phase of development. The phase documentation can be more or less abstract, i.e., more or less concrete. The phase documentation is abstract if primarily properties have been described. The phase documentation is concrete if primarily a model in terms of either a computable program, or a model in terms of such discrete mathematical notions as sets, Cartesians, lists, maps, etc., has been presented. The above distinction allows one to speak about grades of abstractness, versus grades of concreteness.

The distinction between stages and steps is basically a pragmatic distinction. That is, there is no "hard" theoretical basis for making that distinction, but there are good, sensible, practical reasons for doing so.

Characterisation. 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. ▪

Characterisation. By a *stage kind* we shall loosely understand a way of characterising a set of development documents as being comprehensive (i.e., relatively complete) and, at the same time, as specifying (describing, prescribing) a set of properties of what is being specified in such a way that other such sets of documents can be said to describe the same stage kind, or a different stage kind. ▪

Characterisation. Thus a *stage kind* imposes an equivalence relation on a set of sets of related documents: some sets, $s_k, s'_k, \ldots, s''_k$ as belonging to the same kind (k), other sets, $s_k, s'_{k'}, \ldots, s''_{k''}$ as belonging to different kinds $(k, k', \ldots, k'')$. ▪

Discussion. The notion of *stage kinds* is deliberately vague. As for philosophical notions, it therefore needs to be discussed and exemplified. Here we shall discuss that notion. The basic problem is really that, in actual development practice, we need operate with a spectrum of "phases", "stages" and "steps". That is, the simple tripartite decomposition into phases (domain, requirements and software design) may be OK, whereas the likewise simple quadruple decomposition into, for example, business process reengineering requirements, domain requirements, interface requirements and machine requirements stages may, in several development cases, not be entirely satisfactory. The borders between these stages are not that sharp. Human ingenuity allows us to break molds, and to discover new principles and techniques. So why do we then suggest the phases, stages and steps that we do indeed name and describe? We do this so that the reader can be looking vigilant for additional stage and step concepts. In the best case the reader will discover additional, usefully nameable stage concepts. In the worst case the reader may finally decide that our stage and step conjecture is all wrong, and must be refuted. Such, sometimes, happens — as is amply illustrated in works by Imre Lakatos [208] and Sir Karl Popper [277–279]. ▪

The examples given next presuppose that you have read the previous material carefully. We are basically referring to concepts that were just briefly mentioned, i.e., to concepts that will only be further mentioned below, to be finally "disposed of" in later chapters. Some examples, and the discussion text following, refer to concepts that are introduced only a few pages further on!

Example 1.10 *Stage Kinds:* Examples of domain stage kinds are: (d_1) business processes, (d_2) intrinsics, (d_3) support technologies, (d_4) management and organisation, (d_5) rules and regulations and (d_6) human behaviour.

Examples of requirements stage kinds are: (r_1) business process reengineering, (r_2) domain requirements, (r_3) interface requirements and (r_4) machine requirements.

Examples of software design stage kinds are: (s_1) software architecture, (s_2) component design (in which the entire component structuring of a software architecture is decided), (s_3) module design (in which all modules of all components are designed) and (s_4) code. ∎

Discussion. One may properly argue whether the following are not also *stage kinds* rather than steps: domain requirements (r_{2_1}) projection, (r_{2_2}) determination, (r_{2_3}) instantiation, (r_{2_4}) extension and (r_{2_5}) fitting. It really is just a matter of convenience, hence pragmatics. ∎

To properly describe what we shall wish to call a step of development it seems necessary to further elaborate on two concepts. First the concept of a *module of description*. We already covered a version of this notion in an earlier section, where it was "tied" to the concept of program specification (text). We now enlarge upon the module concept and speak of similar, contained domain description and requirements prescription parts. Second we refer to the concept of *refinement*. We also covered a version of this notion in an earlier section, where it was likewise "tied" to the concept of relation between pairs of program specifications (texts). We now enlarge upon the refinement concept and speak of similar refinements of domain description modules as well as requirements prescription modules.

Characterisation. By a *development step* we mean a refinement of a description module, from a more abstract to a more concrete description. ∎

It may now be necessary to improve upon the characterisation of the concept of stage, so as to make the distinction between stages and steps more practical.

Characterisation. By a *development stage* we mean 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. ∎

1.3.3 Domain Development

Stages of Domain Development

Within domain development we can distinguish the following major stages — on which work can beneficially be pursued basically in the order now listed: identification and classification of *domain stakeholders*, and identification and modelling, relative to identified domain stakeholder classes, of a number of *domain facets*. (i) These include: modelling *business process* facets of a domain, (ii) modelling *intrinsics* facets of a domain, (iii) modelling possible *support technology* facets of a domain, (iv) modelling possible *management and organisation* facets of a domain, (v) modelling possible *rules and regulations*

facets of a domain, (vi) modelling possible *script* facets of a domain, and (vii) modelling possible *human behaviour* facets of a domain. We shall briefly characterise these stages shortly, and we cover them in detail in Chap. 11.

Characterisation. By a *business process* domain we shall understand one or more behavioural descriptions in which strategic, tactical and operational sequences of transactions of a business,[6] an enterprise,[7] a public administration,[8] an infrastructure component,[9] are given — each from possibly a number of stakeholder perspectives[10]. ▪

The business process facet overlaps with the next facets. So be it!

Example 1.11 *Railway Business Processes:* A simple business process is that of a passenger inquiring, with a travel agent, about train travel possibilities; being offered some alternatives; settling for one; reserving appropriate tickets; paying and collecting these; starting the travel (i.e., train journey); being ticketed and finishing the journey. ▪

Characterisation. By domain *intrinsics* we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed below), with such a domain intrinsics initially covering at least one specific, hence named, stakeholder view. ▪

Example 1.12 *Rail Intrinsics:* Examples of rail intrinsics are: the rail net, the lines, and the stations — as seen from the passenger perspective — and the above plus the rail units (whether linear [including curved], points [switches], crossover, etc.), connectors (that allow units to be put together), etc. — as seen from the rail net signalling staff; and so on. ▪

Characterisation. By domain *support technology* we shall understand ways and means of implementing certain observed phenomena. ▪

Example 1.13 *Railway Support Technologies:* A rail unit switch can be implemented in either of a number of support technologies: as operated purely

[6] By a business we mean such things as a retail store, a wholesaler, a hotel, a restaurant, etc.

[7] By an enterprise we mean such things as a manufacturing plant, like a distribution company, like a logistics firm, etc.

[8] By a public administration we mean such things as taxes and excises, social services etc.

[9] By an infrastructure component we mean such things as a nation's healthcare system (whether public and/or private), a rail infrastructure owner, a railway, etc.

[10] The above examples, i.e., footnotes 6–9, overlap, and are only suggestive.

by human power, as operated, from afar by mechanical wires, as operated electromechanically, or as operated electronically and electromechanically (say, in interlocking mode). ▪

Characterisation. By domain *management* we shall understand such people who determine, formulate and thus set standards (rules and regulations, see next) concerning strategic, tactical, and operational decisions. Domain management (i) ensures that these decisions are passed on to the ("lower") levels of management, and to "floor" staff, (ii) makes sure that such orders, as they were, are indeed carried out, (iii) handles undesirable deviations in the carrying out of these orders cum decisions, and (iv) "backstops" complaints from lower management levels and from floor staff. ▪

Example 1.14 *Railway Management:* An aspect of train operator management is that some functions, being of strategic nature, are considered on a yearly basis (whether to offer new train services). Other functions, being of a tactical nature, are considered more regularly, although not daily (whether prices should be lowered or raised due to lower, or higher costs, or due to competition or lack thereof). Yet other functions being of an operational nature, and are considered, and decided upon, "from hour to hour" (rescheduling trains due to delays, etc.). ▪

Characterisation. By domain *organisation* we shall understand the structuring of management and nonmanagement staff levels, and the allocation of strategic, tactical and operational concerns to within management and nonmanagement staff levels. Hence we mean the "lines of command": who does what and who reports to whom, both administratively, and functionally. ▪

Example 1.15 *Railway Organisation:* Example 1.14 considered management functions. The number and specialised nature of these usually warrants corresponding organisational structures: executive management entrusted with strategic issues, mid-level management with tactical issues and "floor" (or operational) management with operational issues. ▪

Characterisation. By a domain *rule* we shall understand some text which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their functions. ▪

Example 1.16 *Railway Rules:* In China: At railway stations, no two (or more) trains are allowed to enter and/or leave, including basically move around, simultaneously. In fact, train arrivals and departures must be scheduled to occur with at least 2-minute intervals.

Elsewhere: A line between neighbouring stations is usually segmented into blocks with the rule that at most one train may occupy any one block, or even, in cases, with at least one "empty" (i.e., no train block) between two trains. ▪

Characterisation. By a domain *regulation* we shall understand 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. ▪

Example 1.17 *Railway Regulations:* Regulations may thus prescribe properties that must hold when rescheduling trains, for instance, negotiating with neighbouring stations, etc. Or regulations may prescribe punitive staff actions when a train driver disobeys a train signal. ▪

Characterisation. By domain *human behaviour* we shall understand any of a quality spectrum of humans carrying out assigned work: From careful, diligent and accurate, via sloppy despatch and delinquent work, to outright criminal pursuit. ▪

Example 1.18 *Railway Staff Behaviour:* A railway ticket collector may check and double-check that all passengers have been duly ticketed, or may fail to do so, or may deliberately skip checking a whole carriage, etc. ▪

Steps of Domain Development

Steps of domain development are now to be seen as such activities which do not materially, i.e., in substance, change the properties of what is being described, but which refine, from more abstract to more concrete, their way of description. Other than the above, we shall not cover the issue of domain development steps in the present chapter, but refer to Chaps. 8–16 for more details.

1.3.4 Requirements Development

From Sect. 1.2.3 we repeat some of the below characterisations.

Characterisation. By *requirements* we shall understand a document which prescribes the desired properties of a machine: *what* the machine shall (must, not should) offer of functions and behaviours, and what entities it shall "maintain". ▪

Characterisation. By *requirements prescription* we mean the process — and the document which results from the process — of requirements capture, analysis and synthesis. ▪

Characterisation. By *requirements engineering* we understand the development of requirements prescriptions: from requirements prescription via the analysis of the requirements document itself, its validation with stakeholders and its possible theory development. ▪

Stages of Requirements Development

We see four different kinds of requirements: (i) *business process reengineering*, (ii) *domain requirements*, (iii) *interface requirements*, and (iv) *machine requirements*. Conventionally the following terms are in circulation:

- *functional requirements* which approximately cover our domain requirements;
- *user requirements* which approximately cover our interface requirements;
- *non-functional requirements* which approximately cover some of our machine requirements; and
- *system requirements* which approximately cover some other of our machine requirements.

Business Process Reengineering Requirements

Characterisation. By *business process reengineering requirements* we understand such requirements which express assumptions about the future, usually changed, business process behaviour of the environment of the machine as brought about by the introduction of computing. ▪

We suggest five domain-to-business process reengineering operations — which will be covered in Sect. 19.3: (i) introduction of some new and removal of some old *support technologies*, (ii) introduction of some new and removal of some old *management and organisation structures*, (iii) introduction of some new and removal of some old *rules and regulations*, (iv) introduction of some new and removal of some old work practices (relating to *human behaviours*), and related *access rights* (i.e., password authentication, authorisation), and (v) related *scripts*.

Domain Requirements

Characterisation. By *domain requirements* we understand such requirements, to software, which are expressed solely in terms of domain phenomena and concepts. ▪

We suggest five domain to requirements operations that will be covered in Sect. 19.4: domain projection, domain determination, domain instantiation, domain extension and domain fitting.

Business Process Reengineering and Domain Requirements

So in setting out, initially, acquiring (eliciting, "extracting") requirements, the requirements engineer naturally starts "in" or "with" the domain. That is, asks questions, of or to the stakeholders, that eventually should lead to the formulation of business process reengineering and domain requirements.

Interface Requirements

Characterisation. By *interface requirements* we understand such requirements, to software, which are expressed in terms of domain phenomena shared between the environment and the machine. ∎

We consider five kinds of interface requirements which will be covered in Sect. 19.5: *shared data initialisation requirements, shared data refreshment requirements, man-machine dialogue requirements, man-machine physiological interface requirements*, and *machine-machine dialogue requirements*.

Machine Requirements

Characterisation. By *machine requirements* we understand those requirements of software that are expressed primarily in terms of concepts of the machine. ∎

We shall, in particular, consider the following kinds of machine requirements — to be covered in Sect. 19.6: *performance requirements*, *dependability requirements*, *maintenance requirements*, *platform requirements* and *documentation requirements*.

Steps of Requirements Development

Steps of requirements development are now to be seen as such activities which do not materially, i.e., in substance, change the properties of what is being prescribed, but which refine, from more abstract to more concrete, their way of prescription. Other than the above, we shall not cover the issue of requirements development steps in the present chapter, but refer to Chaps. 17–24 for more details.

1.3.5 Computing Systems Design

Given a comprehensive set of requirements, including, notably, machine requirements, one is then ready to tackle, systematically, the issue of implementing these requirements. Usually these requirements not only, as their main implication, direct us to design software, but also in many instances imply hardware design. In other words: computing systems design derives from requirements.

Stages and Steps of Hardware Design

Performance, dependability and platform requirements typically imply a need for rather direct considerations of hardware — whether computers, computer peripherals, or sensory and actuator technologies. By stages and steps of hardware design we thus mean such which determine the overall composition of hardware: information technology units, buses, etc., and their interfaces, and the specific design of information technology units and buses, and so on.

Sometimes trade-off decisions have to be made as to whether a required function or behaviour is to be implemented in hardware or in software. These are called *codesign* decisions. Other than just mentioning these facts here, we shall not cover the subject till Chap. 25.

Stages of Software Design

We have briefly mentioned the problem before: sometimes a set of requirements and a set of (domain) assumptions on the stability of the environment and the execution platform allow us to first develop a high-level, i.e., abstract, software design from domain (and possibly some interface) requirements. At other times these assumptions are such (i.e., imply instability, such) that we must first, defensively, develop a less high-level, i.e., a less abstract, software design from machine requirements.

In the former case we say that we are first designing a *software architecture:* something that very directly reflects what the user most directly expects. In the latter case we say that we are first designing a *software component and module structure:* something that very directly reflects what some machine requirements imply. The boundary between the two design choices is not sharp.

It is possible to identify other software design stages. Some may involve "conversion" from informal (or formal, abstract specification) language specifications to the identification and (hence) reuse of existing, "ready-made" and/or instantiatable (i.e., parameterisable) "off-the-shelf" (OTS) modules and components. Others involve conversion from informal (or formal, abstract specification) language specifications to formal (say, programming) language specifications, without the use of OTS software. This stage includes the final coding stage. Also here the boundaries are usually fuzzy.

Steps of Software Design

We have, for this book, assumed that the reader already has some knowledge of programming, i.e., of software design, albeit at a perhaps rather concrete, i.e., coding, level. In line with this assumption we shall not treat the important concept of software refinement. Instead we shall assume that the reader has studied, or will study, such textbooks as: Dijkstra's *Discipline of Programming* [86], Gries' *Science of Programming* [130], Reynolds' *Craft of Programming* [296], Hehner's *Logic and Practical Theory of Programming* [158,159], Jones'

Systematic Software Development [197, 198], Morgan's *Refinement Calculus* [249] or Back and von Wright's (earlier) *Refinement Calculus* [19]. In Chap. 29 we shall, however, briefly illustrate notions of software design refinement.

1.3.6 Discussion: Phases, Stages and Steps

The *phase, stage* and *step* concepts, i.e., the concept of *separation of concerns*, are — pragmatically as well as semantically — important. Hence we shall further clarify these concepts.

Iterations of Phases, Stages and Steps

Ideally it would be nice if a software development could proceed linearly, from domain development, via requirements development, to software design. But reality seldom permits linear thinking and development. Instead one often encounters, in software developments which span the three phases, that they iterate: forwards and backwards between temporally neighbouring, even further temporally spaced phases. Figure 1.2 attempts to illustrate this iteration.

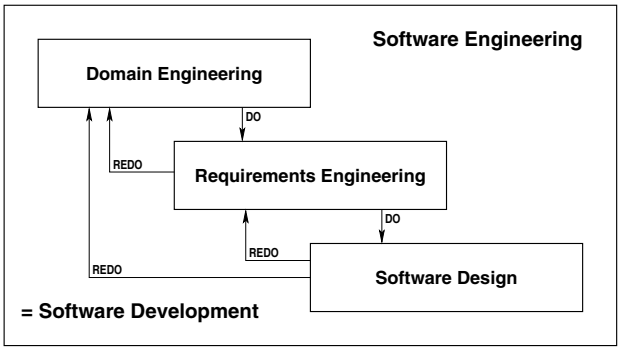


Fig. 1.2. A diagramming of the iterative triptych phase development

The forward, i.e., the temporally linear progression, is in Fig. 1.2 shown by arrows labelled DO. The backward, i.e., the temporally iterative regression, is shown by arrows labelled REDO. Thus iteration is afforded by traversing, in one's development, a sequence of one or more DO and one or more REDO labelled arrows. We shall later explain, more carefully, what it may mean to do iterative and evolutionary development. Similar remarks can be made concerning iterative stagewise and iterative stepwise developments.

Concurrency of Phases, Stages and Steps

Usually phases are developed, one at a time. First the domain phase is developed, then the requirements phase, and finally the software design phase. For

stages of a phase and steps of different stages one may sometimes be able to carry out their development concurrently, that is, by different teams of developers at the same, or at least in partially overlapping time intervals. Stage of domain modelling usually follow the sequential order shown in Fig. 1.3.

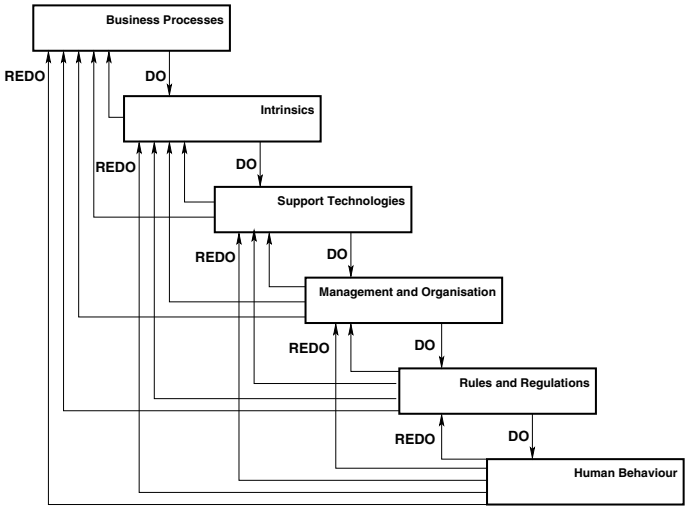


Fig. 1.3. A diagramming of iteration of domain stages

Typically the domain requirements, the interface requirements and the machine requirements stages can be developed independently, i.e., concurrently. Independent development is also appropriate for the individual "steps" within machine requirements: performance, dependability, maintainability, platform, and documentation requirements steps. Figure 1.4 illustrates the possibly independent development of machine requirements stages.

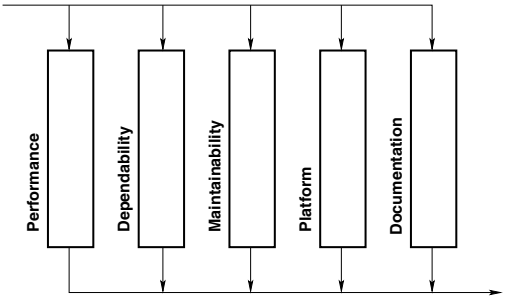


Fig. 1.4. A diagramming of stage concurrency of machine requirements

The diagrams shown in this section lead us into the subject of process models.

1.4 The Triptych Process Model — A First View

The term process model has had, and continues to have, some currency and is somewhat fashionable among practitioners and researchers of software engineering. We shall therefore very briefly respond to that notion.

1.4.1 The Concept of a Process Model

In software development many teams of many people each may have to collaborate over long periods of time and over geographically widely distributed locations. It is therefore of utmost importance that clear guidelines, principles, techniques and tools are established and are agreed upon by all teams and people involved. The concept of a *software development process model* and its enunciation serves this role.

Characterisation. By a *software development process model* we shall understand a set of guidelines for how to start, conduct and end a software development project, a set of principles and techniques for decomposing these parts (start, conduct and end) into smaller, more manageable parts, and a set of principles, techniques and tools for what to do in, and how to do, these smaller parts. ∎

In this section we shall thus very briefly summarise the basic ideas of the software development process model that these volumes are based upon.

1.4.2 The Triptych Process Model

Figure 1.2 highlighted what we shall refer to as the triptych phase process model. Figure 1.3 diagrammed an iterative process model for part of domain development. Figure 1.4 illustrates a concurrent process model for part of requirements development.

In the next sections we shall summarise the triptych process model. Throughout it is useful to keep in mind our remarks (Sect. 1.3.6) on iterative and concurrent phases, stages and steps of development.

1.5 Conclusion to Chapter 1

It is time to complete this long introductory survey chapter.

1.5.1 Summary

We have introduced crucial aspects of our approach to software development.

- *Definitions of software engineering:* First, in Sect. 1.1 we brought in "old" and "new" definitions and characterisations of "what is software engineering".
- *The triptych of software engineering:* Then, in Sect. 1.2 we surveyed the three key phases of our unique approach to software development: domain engineering, requirements engineering and software design.
- *Phases, stages and steps of software development:* In Sect. 1.3 we reviewed these three phases and further suggested stages and steps of development within these phases and stages. We invite the reader to recapitulate the stages.
- *Software development process models:* And in Sect. 1.4.1 we very briefly broached the topic of process models, in particular the one brought forward by this book. This process model will be enlarged upon in subsequent chapters, notably Chap. 16, Chaps. 24, 30 and 31.

1.5.2 What Will Be Covered Later?

Naturally, this chapter has only provided a glimpse of things to come.

- *Domain Engineering:* Part IV will cover domain engineering in "excruciating" detail.
- *Requirements Engineering:* Part V will cover requirements engineering in "painstaking" detail.
- *Software Design:* And Part VI will cover software design in a somewhat more superficial manner!

1.6 Bibliographical Notes

Section 1.1 referred to several leading textbooks on software engineering. Those and others are by the following authors:

- Ian Sommerville [338]
- Roger S. Pressman [284]
- Shari Lawrence Pfleeger [275]
- Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli [121]
- Watts S. Humphrey [175]
- Hans van Vliet [369]

If you do not have access to Vols. 1 and 2 of this series on Software Engineering, then we recommend the Ghezzi, Jazayeri and Mandrioli textbook [121]. In particular, Software Engineering, Vol. 2 makes our otherwise recommended access to [121] unnecessary.

To complement the present three volumes we strongly recommend Hans van Vliet's fine work [369]. Also, Humphrey's work, [175], is a good supplement to the present three volumes.

1.7 Exercises

1.7.1 On a Series of Software Developments

In this volume we will relate to a number of complete software developments. Some will be covered, here and there, in the body of chapters in this volume, or have already been partially touched upon in previous volumes (railways, etc.). Other aspects of this claimed complete development will be covered in the exercise sections of most chapters.

Common to this coverage is a number of rough sketches of specific application domains. Expressed briefly and phrased as problem questions to be solved these are:

1. *What is administrative forms processing?*
2. *What is an airport?*
3. *What is air traffic?*
4. *What is a container harbour?*
5. *What is a document system?*
6. *What is a financial services system?*
7. *What is freight logistics?*
8. *What is a hospital?*
9. *What is a manufacturing company?*
10. *What is the market?*
11. *What is a metropolitan area*[11] *tourism industry?*
12. *What is a railway system?*
13. *What is a university?*
14. *What is public administration?*
15. *What is a ministry of finance?*

Next we briefly give very rough sketches of each of these individual domains.

1. **What is administrative forms processing?**

Typically enterprises base part of their day-to-day operations (especially administration) on a small set of forms. These include *employment forms:* application, employment offer, offer acceptance or rejection, work exercise form, form(s) for reporting sick leave, leave with, or without pay, termination or notification forms, and so on; and *procurement forms:* product or service inquiry, product or service offers, requisition, receipt form, inspection (acceptance or

[11] Such cities as Singapore, Macau, Hong Kong, London, New York, Tokyo, Paris, etc. can be said to be 'Metropolitan Areas'.

rejection) form, payment form, etcetera. Each form basically contains preformatted fields, to be filled in, partially or fully. Each such partially filled in form may undergo several rounds of filling in and possibly, where needed, approvals (i.e., signatures).

2. **What is an airport?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of people (passengers), material (fuel, catering, luggage), aircraft, information (passenger, luggage, catering, fuel, servicing, etc., information), and control in an airport.

3. **What is air traffic?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the movements (start-up, take-off, flight, preparation for landing, possible holding (in holding areas), touch-down and taxiing) of aircraft — under the monitoring and control by ground, terminal, area and continental air traffic control towers.

4. **What is a container harbour?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of ships and cargo, into and out from a container harbour: ships arriving at a container harbour, ships having, possibly, to anchor for container quay place, ships unloading and loading containers, ships being detained for customs, illegal cargo, or lack of sea-worthiness reasons in a harbour, ships cleaning their fuel tanks in a harbour, and ships leaving harbour.

5. **What is a document system?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with documents: their creation as originals, at a certain time and location, their placement with (allocation to) people or file cabinets, their copying (whereby unique, distinct copies are made, with no two copies of the same document being the same due to their necessarily being copied at different times), their editing (whereby the document which is being edited — whether an original, a copy, or a version — becomes a version of the document it was "edited from"), their movement (i.e., transfer from persons or file cabinets to (other) persons or (other) file cabinets, all necessarily having different locations — or their movement because the person with whom a document is associated is carrying that document "around"), or their shredding.

6. **What is a financial services system?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with people, customers, using banks, insurance companies, stockbrokers and portfolio managers. Thus also the entities, functions, etc., of these phenomena need be described. Of special interest are transfers of securities instruments between banks, insurance companies, stockbrokers, the (assumed one) stock exchange, and portfolio managers.

7. **What is freight logistics?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with (1) people (senders) inquiring with logistics firms about and actually sending or receiving freight transports; (2) with logistics firms arranging such transportation with trucking companies, with freight train operators, with ship owners, and with air cargo companies — as well as logistics firms interacting with trucking and freight train depots, harbours and airports; with (3) trucks, trains, ships and aircraft unloading and loading freight at depots, harbours and airports, etc. A central concept is that of a *way bill* (or a *bill of lading*), which directs freight from its point of origin via intermediate hubs (depots, harbours, airports), to its final destination.

8. **What is a hospital?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of patients, visitors and healthcare workers, of materials (beds, medicine, etc.), information (patient medical records with update information on clinical tests, X-rays, ECGs, MR scans, CT scans, etc.) and control in a hospital. Thus patient treatment, as a process, and its interaction with other hospital processes needs to be narrated.

9. **What is a manufacturing company?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of orders into and deliveries from a manufacturing company, as well as the flow of materials (parts), equipment (trucks, conveyor belts, etc.), information (sales orders, production orders, etc.), and control among and within the various departments of a manufacturing enterprise: marketing, sales and service, design, production floor (machines [lathes, saws, mills, planers, etc.] and their in and out trays, delivery trucks, etc.), parts and products warehouses, etc.

10. **What is the market?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with customers inquiring about, ordering, taking delivery, returning (rejecting), accepting and paying

for, merchandise — with, from, and to retailers, who again perform similar actions with wholesalers, who again perform similar actions with producers, and where distribution companies may be involved in deliveries from producers to wholesalers to retailers to consumers.

11. **What is a metropolitan area tourism industry?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the inquiry, arrival, flow and departure of people (tourists, conference-goers, business people) about, to, within and from a metropolitan area: between airports and hotels, and between hotels, restaurants, shops, museums, theatres, parks, and historic sights. Inquiry about and reservations of hotel rooms, restaurants (tables), theatres (tickets), the inquiry and buying of transport cards, what to buy, planning of shopping (itinerary), etc. — all are part of what a visitor to a metropolitan area undergoes, including possible visits to the dentist, medical doctor or hospital emergency room.

12. **What is a railway system?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the rail net (lines and stations), timetables, train traffic, passengers inquiring, buying tickets, cancelling or using tickets, etc. The lines and stations consists of rail units, signals, etc. Thus railway system personnel despatch and reschedule, maintain (clean, repair, etc.) trains, and personnel are rostered (i.e., assigned to train duties), and so on.

13. **What is a university?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with students, lecturers and administrators, students' and lecturers' courses cum classes, course descriptions, lecture plans, lecture rooms (exercise and occupancy), examinations, etc. Included is students applying for admission to a university and registering for courses; of lecturers preparing courses, posting information, lecture notes, etc., and actually giving lectures. You are to "add", i.e., join to the previous, all those "other things" that you associate with being a university.

14. **What is public administration?**

The basis for this group of exercises is the creation and administration of such laws which concern the daily life of citizens and whose effects they daily, weekly, monthly, yearly or just occasionally "suffer" or benefit from — as the case may be. For example, such laws as govern social welfare, healthcare benefits, taxes and excise, building codes and land zoning (regulations), and so on. Creation of these laws takes place in parliament. Ministries prepare drafts of

laws to be put before parliament. Parliamentary committees discusses these laws and may recommend changes or adoption by the parliament. Parliament discuss these laws and eventually adopts the laws with which this group of exercises are concerned. Ministries formulate and ministers sign rules and regulations on how civil servants are to administrate these laws. Public administration divisions may further amend these rules and regulations with respect to particular interpretation. And so on.

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the creation, all the way through parliament, ministries, public administration divisions down to the individual offices with which the citizens interact. Citizens may wish to know about the history of the law: what motives there were for creating the law in the first place, what discussions took place in parliamentary subcommittees, in parliament (i.e., how it was enacted), what rules and regulations were formulated by ministerial offices and which administrative procedures were formulated, and the practice of these by public administration divisions, and so on. Finally, and, as concerns the intention of the law, citizens need to interact with the law by requesting, for example, benefits from it, by submitting, for example, reports, and so forth.

15. **What is a ministry of finance?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with a ministry of finance's taxation and budget departments and the treasury.

In contrast to the above 14 domain outlines — where we relied on your own prior, albeit only superficial, knowledge of those domains — we shall elucidate the "workings" of the ministry of finance somewhat more.

A ministry of finance's perception of the nation in which it serves is that it is hierarchically organised: the state (s), the (nonoverlapping) provinces (p_i), the (nonoverlapping) districts (within provinces, d_{i_j}), and the (nonoverlapping) communes (cities, townships, villages, etc., $c_{i_{j_k}}$) within provinces — such that all provinces "make up" the state ($\{p_1, p_2, \ldots, p_i, \ldots, p_p\} = s$), all districts of a province "make up" that province ($\{d_{i_1}, d_{i_2}, \ldots, d_{i_\iota}, \ldots, d_{i_d}\} = p_i$), and all communes of a district "make up" the district ($\{c_{i_{\iota_1}}, c_{i_{\iota_2}}, \ldots, c_{i_{\iota_i}}, \ldots, c_{i_{\iota_c}}\} = d_{i_\iota}$).

Now the main functions of a ministry of finance wrt. the taxation and budget departments and the treasury are as follows:

(1) Annually an order is issued — by the ministry of finance taxation department — whereby the corresponding taxation departments of each province (of the state), each district within each province (of the state), and each commune within each district (etc., etc.) are to assemble, gather, obtain, by census or otherwise, statistical data, that is "the assessment data". These data represent "best guesses" of the basis for tax revenue (such as personal income, sales (for sales tax purposes), fees (for services rendered by province, district or commune authorities), etc.). From the communes this kind of data is

communicated (perhaps in simplified, summary form) to the district of that commune, and likewise from district to province, and to state. These communications must take place before certain dates ($D_{a_{c\to d}}, D_{a_{d\to p}}, D_{a_{p\to s}}$).

(2) More or less simultaneously an order is issued — by the budget department of the ministry of finance — whereby each ministry (m_μ, incl. the ministry of finance, m_f) is to set up a budget, B_{m_μ}, for next year's activities (i.e., expenditures) E_{m_μ}. The ministry of finance sets an initial ceiling I_{m_μ} (of so many millions of, say, dollars) for respective ministries' expected incomes. The various ministries contribute their (possibly negotiated) budgets for next year to the ministry of finance by a certain date $D_{\to m}$. A twist to this budgeting process may occur if the ministry of finance judges, well before $D_{\to m}$, but after $D_{a_{p\to s}}$, that the assessment data warrants either a downward (pessimistic), or an upward (optimistic), adjustment of the income I_{m_μ}. The submitted budget B_{m_μ} must balance within the possibly adjusted set income ceiling I_{m_μ}. The various ministries also have "shadow" budget departments in each province, district and commune.

(3) The parliament then negotiates and eventually, in time for the next year, passes the national budget, B_s, as assembled from all ministries' individual budgets B_{m_μ}.

(4) The budget B_s is subdivided into province, district and commune expenditures.

(5) Finally, the next fiscal year arrives, and the ministry of finance taxation department requests the taxation departments (of provinces, districts and communes) to regularly gather all relevant taxes and regularly send appropriate proportions of these taxes to the corresponding commune, district, province and state treasuries. Thus some proportion of a commune tax revenue goes to that commune's treasury, and the rest to the district treasury. As districts, independently of communes, also gather taxes, their income derives from these taxes and from the communes, and its outlay goes locally, to the district treasury and the treasury of its province, and so on.

1.7.2 Introductory Remarks

Three remarks are in order:

- *Selected Topic:* When in the following we mention a selected topic, we are referring to any one of the 15 problems listed in Sect. 1.7.1.
- *Informal/Formal:* When using the present volume in an informal course on software engineering, answers to the below questions need only be given informally, i.e., in precise natural language text (e.g., English). When using the present volume in a formal course you are to present both a precise natural language text and formulas to support that text.
- *Incremental/Evolutionary Solutions:* For each of the exercises — and, in principle, you need solve all of them — you may try your mind and hand on it. But since it is rather early in this volume and since, in particular,

much material on how to really solve these exercises is given in almost all chapters that follow you are expected to review your solution to the below exercises, one review, ideally, for each of the many coming chapters!

1.7.3 The Exercises

Exercises in subsequent chapters will repeat the first nine exercises, but in more elaborate forms.

Exercise 1.1 Domain Entities: For the fixed topic, selected by you, list some dozen or so domain entities. Give suitably short names for their types and describe these, whether simple or composite, and, if composite, describe their composition.

Exercise 1.2 Domain Functions: For the fixed topic, selected by you, list some half dozen or so domain functions: Give suitably short names to these functions, and describe their signatures, that is, which arguments they "take", and what results in the "yield".

Exercise 1.3 Domain Events: For the fixed topic, selected by you, list some half dozen or so domain events. Give suitably short names to these events, and describe them briefly.

Exercise 1.4 Domain Behaviours: For the fixed topic, selected by you, list, say, three behaviours. Give suitably short names to these behaviours, and describe them briefly.

Hint: Think of behaviours as processes, i.e., of a behaviour as "one" process. Then describe that behaviour as it may also interact, or communicate, thus exemplifying events, with other behaviours.

Exercise 1.5 Domain Requirements: For the fixed topic, selected by you, list, say three or four, domain requirements. Describe them briefly and informally.

Exercise 1.6 Interface Requirements: For the fixed topic, selected by you, list two or three interface requirements. Describe them briefly and informally.

Exercise 1.7 Machine Requirements: For the fixed topic, selected by you, list one machine requirements for each of the "standard" areas: performance, dependability, maintenance, platform, and documentation. Describe them briefly and informally.

Exercise 1.8 Software Architecture Design: For the fixed topic, selected by you, attempt, admittedly rather prematurely, to sketch a software architecture — say in terms of (briefly specified) boxes and (briefly specified) arrows, where boxes denote single-thread processes and arrows denote interactions (messages) between processes. Do this only informally.

Exercise 1.9 Software Component Design: This exercise is a continuation of Exercise 1.8. For the architecture sketch given, by you, in answer to Exercise 1.8, single out one or two "boxes" and specify their data structures and functions. Do this only informally.

Exercise 1.10 The Triptych Process Model: Without referring back to Sect. 1.4, try write down, for yourself, what the essence is of the triptych process model. That is, phases and stages. Name these. Try to sketch some process model diagrams.

2

Documents

- The **prerequisite** for studying this chapter is that you have a basic awareness of the complexities of software engineering — such as, for example, outlined in Chap. 1.
- The **aims** are to introduce the concept of *documentation*, and to introduce the concepts of *informative, descriptive (prescriptive, specificational)* and *analytic* document parts and their many subparts.
- The **objective** is to make you a professional software engineer with respect to the crucial aspect of documentation.
- The **treatment** is informal, almost casual, but rigorous.

Documents and Projects

Documents, such as they are produced during software engineering, belong to projects. Projects either develop domain descriptions, or requirements prescriptions or software specifications, or the first two, or the last two, or all three of these. Software engineering projects basically only develop documents and, less tangibly, human insight. In this chapter we shall be concerned with which kinds of documents are produced by software engineering projects.

2.1 Documentation Is All!

The objective of software engineering is to construct (deploy and maintain) software. Earlier we gave a definition of what we mean by software (see Sect. 1.2.5). The essence of software engineering is to *construct* (and *analyse*) *documents* based on a context of people and the further environment in which this software is to be inserted. We mean not only the code (to be the basis for computations), but also all the other things listed in the characterisation of what software is syntactically. Software engineering does not construct material artifacts as is done in mechanical, civil, chemical, electrical and electronics engineering. It is therefore of utmost importance that the

software engineer is competent wrt. all the aspects of documentation that we shall cover in this chapter. That is, it is imporant that the software engineer is a literate person.

2.2 Kinds of Document Parts

There exists a multitude of documents. So we need first survey this multitude; then we need treat each kind in more detail.

2.2.1 General

We make the distinction between the following three kinds of document texts: (i) texts which *inform*, but that do not (essentially) describe that which the phase of development is to develop, (ii) texts that *describe (prescribe, specify)* and (iii) texts that *analyse*. A development document may contain all of these document parts, but these parts should be clearly delineated and marked.

Sections 2.4–2.6 will now explain what we mean by these parts of proper documentation. The rest of this volume will explain how to construct such proper documentation. But, first some words on description.

2.2.2 What Is a Description?

The term description is not an easy one to capture. To designate what it means for some text to describe something is an almost philosophical task. So it is reasonable that we spend some time on that subject. A little will be brought in here, more will follow in Chaps. 5–7.

Information Versus Description Versus Analysis

Above we made a distinction between three kinds of texts: (i) texts that *inform*, but that do not (essentially) describe, (ii) texts that *describe (prescribe, specify)* that which the phase of development is to develop and (iii) texts that *analyse*. These distinctions need be clarified. But we warn the reader: the distinctions can never be made fully transparent, fully unambiguous. There will be texts which can be claimed to be both descriptive and analytic, there will be texts which can be claimed to be both informative and descriptive (prescriptive), and so on.

In Sect. 4.2.2 we outline the distinction already made, but not properly explained, between description and prescription. For the time being, please take them as semantically synonymous, but pragmatically distinct.

Characterisation. By a *describing (or prescribing) text* we mean a text that designates and/or delineates some physically existing phenomenon, or a text that defines a concept which can be said to be an abstraction of a physically existing phenomenon. ∎

Example 2.1 *A Rough Sketch Inner-City Street System Description:* Let us imagine an inner-city collection of one-way streets. One may, for example, have written the following rough-sketch description of such a system: There is an area of a city called the inner-city. And there is a disjoint, but bordering area of the city called the outer-city. The city consists, trafficwise, of a finite number of streets. A street is something that has two ends, a length (between these), and which allows vehicles to move from one end to the other. Each inner-city street is one-way, i.e., allows traffic in only one direction. Each outer-city street is two-way, i.e., allows traffic in both directions. Either an inner-city street is connected (has an intersection) at each end to one or more inner-city streets, or at each end to one or more outer-city streets, or — in a blend of the above — at one end to one or more inner-city streets, and at the other end, to one or more outer-city streets. A connection (an intersection) of two (or more) one-way streets ("wholly") within the inner-city is further restricted in such a way that any two such connected streets allow traffic only in the same consecutive direction. That is, if the two connected streets are (b_i, s_i, e_i) and (b_j, s_j, e_j) — where (decorated) b's designate the beginning, e's the end of streets, s's the streets themselves, and (b, e)s their direction — then $e_i = b_j$. ∎

Characterisation. By an *analysing text* we mean a text which proves (i.e., reasons over) or designates properties of some other text, or properties that are claimed to hold between pairs of texts. ∎

Example 2.2 *A Rough-Sketch Inner-City Street Analysis:* We continue Example 2.1 by stating properties that resulted from an analysis of the text of that example. (i) It is not possible for traffic in the inner-city to go in circles. (ii) Once inside the inner-city it is indeed possible to also get outside. (iii) It is not possible to get both into and out from the inner-city at the same (two-street) connection. (iv) The maximum number of inner-city one-way streets that one may have to travel to get into and out from the inner-city is the number of inner-city one-way streets. ∎

Characterisation. By an *informing text* we mean a text which is neither basically descriptive nor analytic, i.e., a text which does not designate physical phenomena or concepts directly related to these, but a text which otherwise "points" to or implies descriptive or analytic documents. ∎

Whereas it is relatively easy to characterise what is meant by descriptive (prescriptive) and analytic texts, it is more difficult to characterise what is meant by informative texts. We continue Examples 2.1–2.2. We give examples of informative texts that usually precede the texts of the rough sketches and their analyses.

Example 2.3 *A Rough-Sketch Inner-City Information:* (i) The *partners* to this contractual work are (1) the government of (city) X, the department of roads, and (2) the informatics consultancy firm Y. (ii) The *current situation* is that the inner-city traffic is a mess. (iii) There is thus a *need* for an inner-city traffic system that can improve that situation. (iv) The *idea* is to provide for some kind of point-to-point traffic flow. (v) That is, point-to-point traffic flow is a crucial *concept*. Street segments and street junctions are other crucial concepts. (vi) The *scope* of our concern is that of inner cities. (vii) The *span* of our concern is that of the traffic in their streets. (viii) We intend to *contract* a consulting firm strong in demographics, traffic analysis and prognosis to carry out a further study and come up with recommendations — including giving the city government a *design brief*. ∎

To make it easier for us to delineate what kind of informative texts are necessary in software development projects, we (thus) limit these texts as indicated above: *current situation*, *need*, *idea*, *concept*, *scope*, *span* and *contract*, including *design brief* texts. In addition we may also wish, at times, to formulate a *synopsis*, a kind of *summary* (a *resumé*) of the previous informative text parts. Thus Section 2.4 will cover primarily these information topics. By providing this kind of "information document systematics" we, at the same time, make it easier for the software engineer to know that such informative documents are important, and what they should contain.

Descriptions, Prescriptions and Specifications

We shall, consistent with later material on descriptions versus prescriptions, briefly make the following distinctions. They were already made, but were not fully explained: Domains are *described*, requirements are *prescribed*, and software is *specified*. It is useful to maintain these distinctions.

2.3 Deliverables

Characterisation. By a *deliverable* we shall understand a document which is called for in a contract and which is thus (to be) developed and is thus (to be) exchanged between the partners to that contract — as here software development: whether for just a domain description, or for just a requirements prescription, or for just a software design, or for parts thereof, or for combinations thereof. ∎

There are very many kinds of documents. Hence it is important that a contract makes it very clear which are the deliverables: their form, their expected contents, their expected degrees of detail and formality, whether they have been subject to validation, verification, tests, etc. As we shall also see, many

documents may not (initially) be thought of as deliverables, yet their development serves crucial needs. These volumes are very much centred around the professional production of document deliverables.

2.4 Informative Document Parts

Before serious, usually long and, to some, tedious descriptions can be developed, let alone presented, the developers and their clients, must agree on some basics. Some common information must first be established. The informative documents serve this pragmatic purpose.

Their *syntax* is not prescribed. Their *semantics* cannot usually be clearly defined. But their *pragmatics* is crucial and serves as a basis for management decisions. Usually developers can only make use of small fragments of informative documents — and then only in the role of giving management directions: *What to do!*

We suggest the following kinds of informative document parts:

1. *name, place and date*
2. listing of *partners*
3. *current situation*
4. *needs*
5. *ideas*
6. *concepts*
7. *scope*
8. *span*
9. *synopsis*
10. *assumptions and dependencies*
11. *implicit/derivative goals*
12. *standards*
13. *contract*
14. *design brief*
15. *logbook*

They may serve well to initially establish partner agreements.

2.4.1 Name, Place and Date

A project must have a brief, informative name. A project takes place somewhere, in one or more locations. And an informative project document must be dated.

2.4.2 Partners

Typically there can be many partners to contractual work in informatics. Those whom we typically would call the *clients:* the enterprise, or institution, or others, who wish a problem to be investigated for possible solution by computing, and possibly other such consultancy firms, or others, who may assist, incl., advise, the enterprise in the work with developers. There are also those whom we accordingly would consider the *developers:* the prime developer, possibly also itself a consultancy firm, or, more typically, perhaps, a software house, and possibly specialist consultancy firms, or research institution scientists who may assist, incl., advise, the prime developer in the work with the client. We will now treat these two categories a bit more.

Clients

Clients can contract the development of (i) domain models, inclusive of descriptive and analytic documents, (ii) or clients can contract the development of requirements models, inclusive of prescriptive and analytic documents, and as based on domain models. (iii) Clients can also contract the development of software designs, inclusive of specification and analytic documents, and as based on domain and requirements models. (iv) Clients can contract the development of combinations of domain and requirements models, (v) or clients can contract the development of combinations of requirements models and software designs, (vi) or clients can contract the development of combinations of all: domain and requirements models, and software designs.

To help the client in advising the developer and in evaluating the work (i.e., the deliverable documents) of the developer, the clients often use consultants, either specialists in the domain of the client and in its informatisation, or specialists in the quality assessment of the work of the developers.

An analogue may be in order: Shipping companies, when contracting the building of ships, rather immediately contact such quality assessment consulting firms as Bureau Veritas [51] or Lloyd's Register [224] or Norwegian Veritas [262] or TÜV [356]. These companies, on an unscheduled basis, inspect the design development and the actual building of these ships. The point is that shipping companies normally do not possess the needed expertise to check the oftentimes (mathematically) very sophisticated designs, nor the similarly advanced shipbuilding tool technologies and their correct use, etc. And insurance companies demand the use of these external, third-party evaluators in order to underwrite insurance. To do this work, these companies employ shipbuilding experts of the highest calibre.

Likewise, clients of domain models, requirements models and/or software designs may need experts in informatics to similarly check the quality of development work. Thus a new industry is already growing: one that checks that informatics work is carried out at the necessary level of professionalism.

Developers

Developers can, symmetrically, be contracted for the development of (i) domain models, inclusive of descriptive and analytic documents, (ii) or requirements models, inclusive of prescriptive and analytic documents, and as based on domain models, (iii) or software designs, inclusive of specification and analytic documents, and as based on domain and requirements models. (iv) Developers can also be contracted for the development of combinations of domain and requirements models, (v) or combinations of requirements models and software designs, (vi) or combinations of all: domain and requirements models, and software designs.

To help the developer with the "harder" parts of the work, the developers may use consultants, either specialists in the domain of the client and in its

informatisation, or specialists in the verification of properties of the domain and requirements models, and of the software designs, or in the verification of correctness of stages of development, from domains, via requirements to software, or in specific, novel tool supports (i.e., development technologies).

2.4.3 Current Situation, Needs, Ideas and Concepts

Four diverse notions converge: current situation, needs, ideas, and concepts. We cover them in turn. But first an example.

Example 2.4 *Traffic Connection Across the Sea:* The example is taken from a field completely different from that of software. It concerns the possible establishment of non-ship connection between coastal points of two land masses.

- **Current situation:** The *current situation* is that a lot of people and vehicles: cars, trucks and trains must daily pass between these land masses, and that the time of waiting for and actual ferry transport imposes undue delays and these delays impose undue costs.
- **Needs:** There is therefore a *need* to speed up the waiting and transport times.
- **Ideas:** The *idea* has therefore been put forward that a fixed connection be provided, either (a) a tunnel or (b) a bridge or (c) a combination of tunnel(s) and bridge(s).
- **Concepts:** More specifically the following fixed connection *concept* has been settled upon, namely (i) the designation of two specific coastal areas, π_1, π_2 as the "anchor" points for the fixed connection, (ii) selected because there is a suitable small island, $\emptyset$ between these two points, (iii) the provision of a low combined railway and road bridge between π_1 and some point on $\emptyset$, (iv) the provision of a high road bridge between another point on $\emptyset$ and π_2, and, finally, the provision of a railway tunnel between another point on $\emptyset$ and π_2.

Notice how the issues of the current situation feed into the needs, how the needs feed into the idea, and how the idea leads to albeit loosely formulated concepts. ▪

Example 2.4 expresses a meta-requirement, namely (the idea) that a fixed connection replace ferries. Transferring the above example to software engineering and hence to any of the three phases, the meta-requirement either expresses the ideas of either better understanding the domain, or obtaining a requirements prescription, or designing software. In Example 2.4 there was no expression of any specific requirements to the fixed connection such as being able to carry so-and-so-much road and rail traffic at such-and-such costs etcetera.

Current Situation

Characterisation. By a *current situation* — in the context of informative software development documentation — we shall understand an informal expression of problems of an area of the universe of discourse: be it in connection with a domain, or in connection with requirements, or in connection with software design. ▪

Example 2.5 *Current Railway Net Handling Problem:* The handling of railway net maintenance is very costly. Handling procedures are apt to take a long time and yet be erroneous. Handling is, amongst others, based on the human manipulation of a huge repository of paper documents and drawings concerning the railway net and its actual status (whether worn, past history of repairs, etc.). ▪

The current situation that may motivate just a domain description project — one that may not necessarily be followed by a requirements prescription project — is usually that it is hard to bring new, untrained staff into a domain. There is simply no good introductory training material. The current situation that may motivate a requirements prescription project is usually that software has to be developed.

Needs

Characterisation. By a *need* — in the context of informative domain description documentation — we shall understand an informal expression of the need for a better understanding of the domain — and — in the context of informative software requirements prescription documentation — we shall understand a need as an informal expression of the need for software, and hence for a requirements prescription. ▪

Example 2.6 *Need for Improvement of Railway Net Handling:* We continue Example 2.5. There is, following the identification stated in Example 2.5, thus a need to significantly improve railway net maintenance handling costs, time and accuracy. ▪

Ideas

Characterisation. By an *idea* — in the context of informative software development documentation — we shall understand a very brief formulation of how, in principle, the *need* — in the context of software development — may be fulfilled. ▪

Example 2.7 *Need for Railway Net Support Software:* We continue Example 2.6. The idea is to computerise railway net documentation and its handling procedures, to do so by means of a distributed computing system with transparency between "where stored" and "where used", and to have the system readily store, display and search over texts, drawings, measurements (tables, curves) and photos. ▪

Current problems and hence derived needs may be fulfilled by means other than computing systems: New management and/or better trained staff, or new handling procedures (i.e., business process reengineering), or other.

The idea, usually, in the context of a domain description project, is to develop exactly that: a domain description. And the idea, then, in the context of a requirements prescription project, is to develop exactly that: a requirements prescription. But these ideas may be expressed in a more elaborate, informative form than saying "just that"!

Concepts and Facilities

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.

Characterisation. We use the terms *concepts and facilities* more or less interchangeably: A facility is a physical phenomenon while a concept is a mental construction (covering, usually some physical phenomena or concepts of these). The concepts and facilities referred to here, in the informative parts of project documentation, depend on the universe of discourse. 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 of the domain. 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 of the requirements. ▪

Several concepts and facilities may have to be mentioned in the "early" information documents of a project.

Example 2.8 *Concepts and Facilities of Railway Net Computerisation:* We continue Example 2.7. The concepts and facilities include: (i) To further represent each railway net unit, i.e., each smallest individually handled part of a net, as a tuple, i.e., as a separately storable entity in a relational (say SQL [13,82,180]) database; (ii) to represent relations between (neighbouring, etc.) such units, and units and adjacent (separately, also relation represented) signals, power line wiring, platforms, etc., by similar SQL relations, etc.; (iii)

to also represent civil, mechanical, electromechanical, and electronics engineering images. By these we mean drawings, measurements and photos such that these images can be referred to by suitable attributes — possibly annotating these images in the form of SQL relations. (iv) The conceptual ideas are to provide for a geographical distribution of such databases, (v) to provide relations of all of the above to a geographical information system (a GIS) for the landscapes of the rail nets, and (vi) to provide means of exchange between railway maintenance centres of such data as outlined above by means of XML [202, 291, 307]. ▪

The concepts, in the context of a domain description project, amount to a listing of major entities, functions, events and behaviours of the domain to be described. Possibly the domain description concepts also include a listing of the major intrinsic, support technology, management and organisation, rules and regulation, etcetera, phenomena and concepts.

2.4.4 Scope, Span and Synopsis

The four kinds of informative document parts: current situation, needs, ideas, and concepts, 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 scope, span and synopsis document

Scope

Characterisation. 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 start a new series of examples. Although, as we shall see, also related to railways, the next examples do not relate "directly", but only implicitly, to the previous series of examples (from Example 2.5 to Example 2.8).

Example 2.9 *Scope of Transportation:* We rough-sketch characterise an infrastructure component. The problem, in general, is to understand the domain of transportation: of road, rail, air and sea transport, and of logistics of multimodal freight transport. ▪

Span

Characterisation. 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. ▪

Example 2.10 *Span of Railway Systems:* The problem whose scope was outlined in Example 2.9 is, more specifically, to build a theory of the domain of railway systems: of rail nets, trains, traffic, its planning, monitoring and control, of passengers and freighters, etc. ▪

Synopsis

Characterisation. By a *synopsis*[1] — 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, ideas, concepts, scope and span documentation informing about a universe of discourse for which some development work is desired: (i) the construction of a domain 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). ▪

Example 2.11 *Synopsis for a Railway Systems Domain Theory Project:* The project is to develop and research a domain model for a railway system — such as scope and span indicated above! Thus the domain model is expected to cover such phenomena as: (i) the railway net: its static and dynamic properties, including signaling; (ii) train timetables: their planning, construction and as basis for traffic; (iii) the rolling stock: its composition onto, and decomposition from trains, maintenance, etc.; (iv) traffic: the dispatch, monitoring and control of trains, including rescheduling, etc.; (v) the railway staff: Their rostering onto station, net and train service; (vi...) and so on. ▪

There is, of course, much more to a synopsis than exemplified above. There is more text on scope and span, and more details than just exemplified. Typically between half a page and two thirds of a page. Usually no more. Signatories to a contract will not read any larger such texts!

• • •

Two hard-to-quantify kinds of informative text are often in the backs of the heads of those who procure computing systems. One kind is the assumptions that are made about that which lies outside the domain of the application, and the dependencies any such required computing system will later experience. The other kind are the meta-requirements that are expected to be fulfilled by the deployment of the required computing system. They are treated in the next two sections.

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

2.4.5 Assumptions and Dependencies

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. 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. And for software design there needs to be a requirements prescription. The other kind has to do with delineation of the domain.

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.

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.

Loosely, we admit, that is what we mean by assumptions and dependencies. We shall return to this issue when, in Sect. 23.2.9, we review requirements prescriptions with respect to their being faithful to assumptions and dependencies.

Example 2.12 *Two Sets of Examples of Assumptions and Dependencies:* For the development of a domain description of the financial services industry it is *assumed* that the developers have access to a necessary and sufficient number of professionals possessing the necessary and sufficient knowledge about respective parts of their domain, and the quality of the resulting domain model (also) *depends* on fulfillment of the assumptions. For the development of a requirements prescription for, say a specific bank within the financial service industry domain three sets of *assumptions* are expected to be fulfilled: that there is a mutually accepted domain description of the relevant parts of the financial services industry, that the developers have access to a necessary and sufficient number of professionals possessing the necessary and sufficient knowledge about respective parts of desired requirements and that the client is ready and willing to consider possible business process reengineerings of the current management and operations of the domain. Two sets of *dependencies* are expected to be fulfilled: the quality of the resulting requirements model (also) *depends* on fulfillment of the assumptions, and the users of the eventually developed and installed software loyally adopt possible business process reengineering practices. ▪

2.4.6 Implicit/Derivative Goals

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.

Typical metagoals 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)."

Other kinds of metagoals 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."

We will review implicit and derivative goals in Sect. 23.5 when we review the issue of whether a requirements document complies with these goals.

Example 2.13 *Two Sets of Examples of Implicit/Derivative Goals:* For the development of a domain description of the financial service industry we foresee the following implicit or derivative goals: better training of new banking staff (as based on a trustworthy domain description), a better foundation for discussion of business process reengineering plans for the financial service industry (say due to new government regulations), and improved confidence in tackling ambitious computerisation plans. For the development of a requirements prescription for, say a specific bank within the financial service industry domain we foresee the following implicit or derivative goals: (although explicit requirements cannot be directly implemented as part of the desired computing system) it is hoped that the requirements (to be developed) can lead to a computing system that helps the bank attain a leading-edge position in the industry, while securing more enjoyable staff working conditions and improved profits. ∎

2.4.7 Standards

A distinction is made between development standards and documentation standards.

Development Standards

Usually development occurs in the context of following some development standards (one or more). The Institute of Electrical and Electronics Engineers

(IEEE [179]) 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 [181]) and the International Telecommunication Union (ITU [185]), have established similar standards.

Documentation Standards

Usually documentation occurs in the context of following some documentation standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [179]) 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 [181]) and the International Telecommunications Union (ITU [185]), have also established similar standards.

Standards Versus Recommendations

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

Specific Standards

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.

- *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/`
- *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.
  - ⋆ 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.)
  - ⋆ 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 the process for performing the maintenance of software.
- *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.

2.4.8 Contracts and Design Briefs

Contracts

The current situation, needs, ideas, concepts, scope, span and synopsis document parts are preambles to, 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 contracts (documents) are needed for larger projects.

Characterisation. 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. ∎

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

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.

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.

Design Briefs

Characterisation. 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. ▪

Example 2.14 *A Design Brief:* The supplier is to develop a domain description of the equipment procurement part of company X, by such-and-such a date, using best practices domain description principles, techniques and tools — such as illustrated in such-and-such a textbook, etc., etc. ▪

2.4.9 Logbook

Characterisation. *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. ▪

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

Example 2.15 *Logbook:* A postulated logbook may reveal:

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. ▪

2.4.10 Discussion of Informative Documentation

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. The current situation, needs, ideas,

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

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

Methodological Consequences: Principle, Techniques and Tools

Principle. *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. ▪

Principle. *Information Documents:* The informative documents must be authoritative, definitive and interesting to read. ▪

Techniques. *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. ▪

Tools. *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 [72]) and you have a workable system. ▪

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

2.5 Descriptive Document Parts

In this chapter we use the term description and its derivatives to also cover the terms prescription and specification.

Generalities

This section, although concerned with a crucial aspect of software engineering, namely descriptive documentation, only speaks about descriptive documents. The section does not tell you how to construct descriptive documents. That we do in almost all subsequent chapters! Thus this section prepares you for the kind of descriptive documents that are needed in development — whether they describe domains, prescribe requirements or specify software designs.

Pragmatics, Semantics and Syntax

The *pragmatics* of descriptive documents is to serve as the main manifestations of development.

The *semantics* of descriptive documents is intended to be as follows: A description describes something, either something manifest, or something conceptual. In any case that something can be given at least a mathematical model. Therefore we shall also be able to formalise our informal, say English text descriptions. We shall do so mostly using RSL, or combinations of diagrammatic formalisms such as UML-like class diagrams [44, 193, 264, 303], or Petri nets [196, 273, 293–295], or statecharts [144, 145, 147, 148, 150], or message [182–184] or live sequence charts [73, 149, 203] (with either of the latter two possibly combined with statecharts, and with all of this combined with RSL, or RSL [117] extended with timing (TRSL [120]), or with a duration calculus [381, 382] extension to TRSL [120, 155]. When formally presented, descriptive documents thus denote mathematical entities.

The *syntax* of descriptive documents, when informal, say, in English, is usually reasonably well delineated.

Specifics

To work out, i.e., to develop the descriptive documentation, for any phase, is, in contrast to work on informative documentation, a major and therefore a resource-critical development effort. We remind the reader of our earlier characterisation of the concept of description:

Characterisation. By a *describing (or prescribing) text* we mean a text which designates and/or delineates some physically existing phenomenon, or a text which defines a concept which can be said to be an abstraction of a physically existing phenomenon. ▪

We shall consider four kinds of descriptions:

- rough sketches,
- terminologies,
- narratives and
- formalisations.

The last kind of description, the formal description, was introduced in the first two volumes of this three-volume series on software engineering textbooks.

Basically we consider rough-sketch documents to be primarily informally expressed. Likewise with terminologies: they are also considered here as being basically informally expressed. And certainly narratives are informally expressed.

But good engineering practice allows for rough sketches, terminologies and narratives to be augmented by formalisations. Rough sketches *may* thus typically be interspersed, i.e., broken, by formally expressed sorts (i.e., abstract type definitions) and function signatures (i.e., formal function names and function types). Terminology entries *may* likewise contain formal sorts and function signatures. Narratives, on the other hand, in our mind, typically *must* be complemented by formalisation. No *may* here, just *must!* If you, the reader, are studying this book on the basis of skipping the "formal stuff", then fine. Then you can consider the *must* as a *may*.

Informal and Formal Document Parts

It is thus we see that the documents of software engineering are necessarily, or may be, both informal and formal. Along one axis we have the informal documents:

- rough sketches
- terminologies
- narratives

Along another axis we have

- formalisations

Figure 2.1 attempts to show, by larger or smaller *'s (asterisks) where we might typically intersperse informal texts with formalisations.

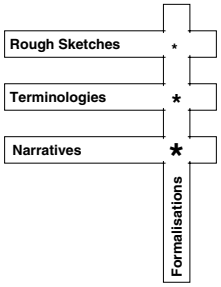


Fig. 2.1. Informal and formal document texts

2.5.1 Rough Sketches

Characterisation. *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, which may still be too concrete, overlapping, or repetitive in its descriptions, and with which the describer has yet to be fully satisfied. ▪

Pragmatics, Semantics and Syntax of Rough Sketches

The *pragmatics* of rough sketches is that their construction serve as the first manifest step of a development, and that the rough sketches can be subject to analysis, and hence the formation of concepts — which all should lead to clear ideas, with these finding their final form in subsequent descriptive documents: terminologies, narratives and formalisations.

The *semantics* of rough sketches is hazy. Rough sketches are indeed intended to denote something, but being rough, and being sketches may mean that there are many loose ends (i.e., incompleteness), some inconsistencies, and that what has been rough-sketch described is not the things the scribe, in the end, really wanted to describe.

The recommended *syntax* of rough sketches, in their informal, typically, as here, English form, is intended to follow the following lines: Separate, in clearly marked paragraphs, the sketching of entities (types and values), functions, events and behaviours. Their order is not prescribed. More will be said later.

Discussion

Rough sketching, as a bona fide activity (although rough sketches are usually not part of deliverables), allows the developer to get started. Such starts represent a less systematic way than subsequent analysis and further descriptions may afford.

Example 2.16 *Rough Sketch of Freight Logistics:* By freight logistics we mean a structure of entities, of functions, events and behaviours that include: (i) senders of freight: inquiring about freight despatch conditions, submitting freight for conveyance, tracing such freight, etc.; (ii) receivers of freight: inquiring about freight arrivals, fetching such freight, etc.; (iii) logistics firms which respond to inquiries (as above), which receive and deliver freight from senders, respectively to receivers; which arrange with transport companies for the transport of freight; which trace (and organise the rerouting of) freight; etc.; (iv) transport companies, such as trucking firms, train operators, shipping lines and airlines, that is, companies which operate line or bulk transfer of freight, whose trucks, trains, ships, and aircraft call at truck and train depots, harbours and airports for transfer of freight, etc.; (v) truck and train

depots, harbours, and airports that provide facilities for temporary storage and the receipt and the handing out of freight, etc. ∎

The above is a description in that certain phenomena of a physical world — entities, functions and behaviours — are identified, and certain relations between them are hinted at. It is a rough-sketch description in that the description raises more questions to be answered than before the reader had read the description. The rough sketch description, as an activity, is therefore to be followed by the *analytic activity* of *concept formation*.

Rough sketching is an iterative, explorative and experimental activity. The developer tries one approach, and analyses the result, and may not be happy. So the developer tries another rough-sketch approach, analyses the result, and may still not be happy. Hopefully this exploration of alternative approaches, experimentation with different starting points, converges. It is crucial that the developer is willing to scrap, or throw away, rough sketches that are not felt conducive to concept formation analysis, that is, for which such analysis does not yield interesting, illuminating concepts.

Rough sketching is an indispensable activity the more research-oriented a development project is. Some development projects, whether of domain descriptions, requirements prescriptions, or a software design, are new: the developers (at least) have no or little experience in the field. Hence experimental or explorative development is required. For such research oriented development one cannot expect fixed, a priori determinable allocation of resources, including staff and time! The need for research-oriented development is often overlooked.

Methodological Consequences: Principles, Techniques and Tools

Principle. *Rough Sketching:* When embarking on a new development, of a domain description, of a requirements prescription, or a software design, first do some rough sketching, present this to colleagues, get comments, revise the rough sketches, etc., until ready for terminologisation and concept analysis. ∎

Techniques. *Rough sketching* — usually based, as we shall see much later in this volume, on domain or requirements acquisition — is, despite the acquisition, still an art. It requires creative skills. It can probably best be learned, by reading many rough sketches (and, subsequently, many narratives). Trial and error with peer review is a good way of learning rough sketching.

Loosen up, relax, go for a walk, think of the topic. Try to sketch, and if you are not satisfied, throw it away — do not try editing an unsatisfactory sketch. Start all over. Remember that the essence is inside your head. If you face continued problems, during a day's work, leave it for that day. Think of it on your evening stroll, before going to bed; think of it while you try fall asleep. In the early morning it will all be much more clear to you! Rough-sketching

is not a group effort; it is not a committee task. Individuals have to show the way. ▪

Tools. *Rough Sketching:* The tools mentioned in Sect. 2.4.10 apply equally well here. ▪

2.5.2 Terminologies

We refer to Vol. 1's glossary Appendix B for a longer story on the terms *glossaries, dictionaries, encyclopedia, ontologies, taxonomies, terminologies,* and *thesauri.* This glossary offers a rather unique and complete, 788 entry, software engineering terminology.

General

Characterisation. By a *terminology* — in the context of descriptive software development documentation — we shall understand a document text (i) which is a collection of term entries, (ii) where a term entry is a pair: the term to be defined and its (narrative) definition, (iii) such that these definitions may contain other terms that need be defined, (iv) but such that a terminology claimed complete has all terms defined and none with unresolved circularities. ▪

Principle. *Terminologisation:* In any development project phase (i) establish, early in the phase, a terminology for the universe of discourse, (ii) adhere, in all documentation, to this terminology and (iii) maintain the terminology. That is, make sure that needs for updates, i.e., changes, are immediately met, and that such changes are immediately reflected in all documentation. ▪

Pragmatics, Semantics and Syntax of Terminologies

The *pragmatics* of a terminology is given just above, under terminologisation.

The *semantics* of a terminology is that every term is well-defined and defines something! What that something is depends on the formality of the term definition. If the definition is (also) given formally, then we expect the defined term to denote some mathematical structure.

The recommended *syntax* of a terminology is suggested to be such as to help facilitate the following operations that terminology creators, users, and maintainers typically perform on terminologies: Separating, i.e., setting out, terms that are terms of a terminology from words that are commonly used, but are not terms of the terminology, that is: Finding out whether a word is a term. Finding out whether a term has been defined. Finding all uses of a term in other definitions. Identifying terms that are used but not defined.

Somehow listing, i.e., displaying, the structure of all term definitions invoked by a given term definition.[2]

Terminologisation — Continued

The terminologisation principle basically requires that separate terminologies are (i) established, (ii) adhered to and (iii) maintained, for each phase: (a) for the professional terms of the application domain, (b) for the professional terms of the requirements domain and (c) for the professional terms of the software technology domain.

Example 2.17 *Railway System Terminology:* We bring in but a fragment of a terminology:

A rail connector is a further unexplained property of a *rail unit*.

A rail line consists of one or more linear rail units, such that these are *sequentially* connected, and such that the *first* and *last rail units* of any line are *connected* to rail units of a station.

A rail unit is a smallest entity of railway signalling concern. As properties a rail unit may have either of the following: A rail unit is either *linear* (straight or curved), or is a *switch*, or is a *crossover*, etc. A rail unit has *rail connectors* by means of which it may be *connected* to other rail units. ∎

In the above example we have hinted at a need for a proper structuring and typography of each entry: Some terms are *designatable*, others are *definable*. Many other remarks could be made about terminologies. We shall later explain the distinction between designations and definitions (Sects. 7.2 and 7.3). We have listed the (only) three entries alphabetically.

On Formal Terminologies cum Ontologies

A terminology may be presented informally, in plain text, mixing ordinary, commonly agreed words with defined terms. And a terminology may be presented formally. Just as we shall see that narratives may be formalised, so we can formalise terminologies. In the "ontology school" of formally defined terminologies these terminologies, now referred to as ontologies, are defined axiomatically, i.e., by properties. We may suggest the ontology school axiomatisation concepts, or we may suggest that the general formalisation concepts introduced in this series of software engineering textbooks be followed. Once we get to the point that we have formalised a narrative description we can, indeed, be said to have formalised the terminology.

[2] This may involve means of identifying and displaying seemingly circularly (i.e., possibly recursively) defined terms. It may also imply identifying possibly erroneously circularly defined terms.

But often the developers create, use and maintain an informal (yet precise) terminology. Formalisations of narratives often are structured differently from how one might wish to structure a formal terminology. But, as a general guideline, attempt to structure a formal narrative so that it can also serve as a formal terminology. And, further, do so hand in hand with the formalisation of the narrative, not with the creation of the terminology. You will often find that the work on terminology creation, which is needed before serious work on a narrative can begin, must be updated, redone and maintained, as and when finalisation of the narrative and its formalisation takes place.

Methodological Consequences: Principles, Techniques and Tools

Constructing a terminology is an art. Some basic principles must be adhered to. Techniques and tools can be suggested.

We previously enunciated a principle of terminologisation. We will, next, repeat a version of this principle, and then follow up with corresponding techniques and tools.

Principle. *Terminologisation:* Main terminology construction principles are: Every term of the professional language of the terminologised universe of discourse must be included in the terminology. And every term must be defined only using ordinary, i.e., commonly agreed-upon natural language and terms of (i.e., defined by) the terminology. ▪

Techniques. *Terminology Documentation:* Some terminology construction techniques are: First identify a smallest, or a reasonably small set of terms whose definition need not involve defined terms. When defining terms formally identify the denotation, i.e., the mathematical, not necessarily the computational, meaning entity τ, of the atomic term. Then define remaining, i.e., composite, terms using one or more of (already, or to be) defined terms $(t_1, \tau_1), (t_2, \tau_2), \ldots, (t_n, \tau_n)$. When defining composite terms formally, identify whether there naturally exists a homomorphic[3] function H by means of which the meaning of the compositely defined term, m, can be given as $H(\tau_1, \tau_2, \ldots, \tau_n)$. If so, then define the term homomorphically. If H is not easily found, then define the term operationally. ▪

Tools. *Terminology Documentation:* The tools mentioned in Sect. 2.4.10 apply equally well here. The ontology school referred to above is represented by a number of tools. These are, however, all experimental. So we will just mention them in passing and otherwise alert the reader to follow up, via the Internet or otherwise, searching for the "latest" ontology creation tools. ▪

[3] The Glossary, Appendix B, Vol. 1, explains a homomorphism as a function, $\phi : A \to A'$, from values of the carrier A of one algebra (A, Ω) to values of the carrier A' of another algebra (A', Ω') is a homomorphism from (A, Ω) to (A', Ω'), if for any $\omega : \Omega$ and for any $a_i : A$, there is a corresponding $\omega' : \Omega'$ such that: $\phi(\omega(a_1, a_2, ..., a_n)) = \omega'(\phi(a_1), \phi(a_2), ..., \phi(a_n))$ [item 330 pages 601–603].

2.5.3 Narratives

Characterisation. By a *narrative* — in the context of descriptive software development documentation — we shall understand a document text which, in precise, unambiguous language, introduces and describes 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. ∎

Pragmatics, Semantics and Syntax of Narratives

The main purposes of constructing, i.e., the *pragmatics*, of narratives are: (i) To secure that those who develop the narrative indeed do understand what they document: the domain, the requirements, or the software design. (ii) To communicate to stakeholders that which is documented: the (described) domain, the (prescribed) requirements or the (specified) software design. These stakeholders include, for domain descriptions and requirements prescriptions: various groups within the contracting client and, in general, within the domain. For software designs the stakeholders include other groups of software developers than those who wrote the software design specification. Of course, a mere narrative description is no guarantee of understanding, but its acceptance by other stakeholders brings an assurance of understanding rather much closer.

The *semantics* of narratives is that they designate something, either physically or mathematically manifest, i.e., narratives describe something!

The *syntax* of narratives typically is structured around clearly identified paragraphs that describe entities, functions, events and behaviours.

A narrative, to be fully satisfactory, is paired with a mathematical model (say, specified in RSL[4]). The narrative can be said to be an informal rendering of the formal model.

Example 2.18 *A Document Narrative:* We introduce a new kind of domain: that of documents.

We rely on three basic, further unexplained notions: text, location, and time. By a document we understand some text. A document can be created, edited, copied or deleted. That is, a document is either an original, a version,

[4] Besides using RSL, the model may also build on the use of such diagrammatic tools (i.e., languages) as UML-like class diagrams, Petri nets [196, 273, 293–295], statecharts [144, 145, 147, 148, 150] and message [182–184] or live sequence charts [73, 149, 203], with either of the latter two possibly combined with statecharts. Additionally, to capture quantitative aspects of time one may use RSL [117, 118] extended with timing (TRSL [120]), or TRSL extended with duration calculus [381, 382] and [120, 155]. All these languages (i.e., tools) were covered extensively in Vol. 2, Chaps. 10, and 12–15 respectively.

or a copy or a former document no longer exists. From a document one can observe its text, and references to the location and time at which the document was either created (as for an original), or edited (as for a version) or copied (as for a copy).

There are, in all, seven operations which involve and/or result in documents. They are:

(i) Creating a document, which constructs such a document "essentially" from "nothing"!

(ii) Editing a document, which takes a document and results in a version of the "same" document such that it is always decidable which editing was done to the input, i.e., the underlying document. In other words, one can observe both the document (and hence text) of the version before and after editing.

(iii) Tracing the origins of a document: from a version and from a copy one can observe the document from which the version was edited, respectively copied.

(iv) Copying a document, which takes a document and results in two things, the unchanged document from which the copying was made, and the copy, which is a document with basically the same text.

(v) Moving a document, over some time, from some location to some other location, where it is assumed that the document "has rested" and "will rest" at the "from", respectively the "to" locations.

(vi) Shredding (deleting) a document: effectively "removing it from existence".

(vii) Comparing two documents: There are basically two kinds of comparisons, one is manifest, the other is conceptual, but cannot be effected! At any one time one can compare two documents to test whether they have the same origin, i.e., are "descended", through copying and "versioning", from a same (not necessarily original) document. We do not detail what the result of a comparison operation could be in this case, but instead leave it to the imagination of the reader. And one can, figuratively speaking, compare two documents, where one document reflects a status at some time t and the other document reflects a status at some therefrom distinct time t'. Also in this case we do not detail what the result of a comparison operation could be in this case, but leave it to the imagination of the reader — it probably would involve some tracing. The idea behind the (unresolved) comparison operation is to be able to speak of two documents at distinct times "being the same" (or being related through suitable copying and versioning).

Some axioms and derivable properties are needed: No two documents can occupy the same location at any one time, hence no two documents can be created at the same time and location. Any one document occupies exactly one location at any one time, and hence cannot be copied, or edited, at the same time in two or more different locations.

We can think of a document system, i.e., a collection of zero, one or more documents, as a function from time to functions from locations to documents.

Such a document system has a time of origin, when the first document was first created.

Further axioms and properties: A document which "exists" (as such) at two distinct times in a document system, exists at all times in between those two times, that is, there are no "ghost documents". The sum total, at any one time, of documents of a document system is the sum total, since the time of origin to the present time, of the number of document creations, plus the number of "copyings", minus the number of shreddings. That is, documents do not suddenly disappear, never to reappear. ▪

Discussion

Constructing pleasing narratives is, perhaps, the most important part of development. Writing these narratives usually requires a major part of the development resources. The last step of development is that of "turning" narratives — or their formalisations — into executable code. In between, but not illustrated in this early chapter, narratives, from first being fully textual, increasingly contain diagrammatic as well as pseudo-program, or even formal texts. Chapters 5–7 will illustrate the smooth transition from "plain texts" to texts increasingly containing diagrams or formalisations.

Methodological Consequences: Principles, Techniques and Tools

Principles. The main principle of *narration* is to describe (prescribe, specify): to only narrate things (phenomena, concepts) that exist (as in the domain), that shall exist (as for requirements), or that will exist (as for software design). A derivative principle of *narration* is to keep what is described (prescribed, specified) apart, including: domain facts, requirements desiderata and software design proposals. A final principle of *narration* is to achieve something: understanding of what has been described (prescribed, specified), ability to communicate that understanding to others, and firm bases for continued, meaningful, productive work. ▪

We refer to Chap. 4 on models and modelling for an extensive discussion of a highly enumerated set of properties of models, that is, of the things we narrate.

Techniques. *Narrative Documents:* A main technique of *narration* is aimed at creating pleasing descriptions (prescriptions and specifications). Many techniques in Vol. 1 of this series of textbooks in software engineering are aimed at creating such documents. Abstraction, also when just using informal language, is of utmost importance. Simplicity and low number of concepts are likewise important. Emphasising properties, as for axiomatic presentations, or (mathematical) models represents a choice between techniques.

Vol. 2 focuses on the following techniques. Hierarchical or composite ("top-down" or "bottom-up") presentations are covered in Vol. 2, Chap. 2. Designating phenomena or concepts by denotations or, operationally, by computations, is covered in Vol. 2, Chap. 3. Composing models from appropriately chosen state and context configurations is covered in Vol. 2, Chap. 4. Expressing concurrent and temporal phenomena is the subject of Chaps. 5, and 12–15 of Vol. 2. So, all in all, previous volumes cover many narration techniques. ▪

Tools. *Narrative Documentation:* The tools mentioned in Sect. 2.4.10 apply equally well here. As narratives are usually large to very large, the tools must be able to handle essentially indefinitely large narratives. ▪

2.5.4 Formal Descriptions

Volumes 1 and 2 of this series of software engineering textbooks covered formal specification.

Characterisation. By *formalisation* we mean a specification (description, prescription) expressed in a formal language. A *formal specification language* is one which has a precise, mathematical syntax, a precise mathematical semantics and, usually, a mathematical logic proof system congruent with the semantics. ▪

Pragmatics, Semantics and Syntax of Formalisations

The reason for, i.e., the *pragmatics* of, formalisation was stated clearly in the preface to Vol. 1. We will summarise salient points here:

(i) Formalisation brings a degree of clarity that cannot be obtained by using only natural plus professional (universe of discourse) language narratives.

(ii) Formalisation allows us to reason precisely about consistency and completeness of the formalised descriptions, and whether these formalisations satisfy otherwise formally expressed properties.

(iii) Formalisation allows us to relate phases, stages and steps of development, including proving correctness of development. In other words, prove that a concrete stage of development is a correct implementation of a previous, more abstract stage of development.

The *semantics* and *syntax* of formalisations are, as the prefix *formal* indicates, precise, and they differ from formal specification language to language.

On Formalisation

Those readers who study the present volume in its informal version may skip the next paragraphs — as they can always skip the specially framed *formal version* texts. The extensive coverage of numerous formalisation principles,

techniques and tools[5] in earlier volumes does not require us to elaborate further on formalisation other than the summary given next.

A formal specification, when expressed in RSL, typically contains the following parts:

(a) Definition of types, i.e., of spaces of values, whether abstractly, in terms of sorts (just named types), or concretely, say in terms of integer, natural number, real, Boolean, set, Cartesian, list, map or function spaces.

(b) If types are defined as sorts, then a specification would typically contain a number of function signatures of functions that observe properties of sort values and which generate such values (i.e., *observers* and *generators*), as well as (c) axioms over functions and values.

(d) Definition of functions (i.e., of function values), where the form of these definitions may vary from explicit function value definitions, via pre/post condition specified function values, to fully algebraically (i.e., axiomatically) specified function values.

The above four formal specification parts are found in almost all practical formalisation cases.

(e) Some formal specifications may be expressed imperatively, in which case the formal specification will additionally contain a declaration of variables.

(f) Some formal specifications may be expressed in terms of processes that may be composed from other processes: In parallel, with external or with internal nondeterminism, or interlocked. These processes synchronise and communicate with one another over declared channels.

These parts may be grouped into schemes, classes and objects, such as covered in Vol. 2, Chap.10.

Example 2.19 *Formalising a Document Domain:* We follow up on Example 2.18 by suggesting a formalisation. Now what often happens, when trying to formalise an informal, i.e., a narrative description, is that that description is found lacking in precision. In item (i) it was stated: *"essentially" from "nothing"*. What was meant was probably that no other information was previously documented, or otherwise made available, from which an original document could be said to be constructed. In the formalisation below we suggest sharpening that *"essentially" from "nothing"* into: *"from some text, and from knowledge of the location and time at which the creation takes place"* — the location and time was made necessary from analysis of item (vii) and the text on axioms that follows (in Example 2.18). Item (ii) (of Example 2.18) hints that from an edited text one can observe the text from which it was edited. In practical life we may mean something like editions showing crossed out (but not entirely "blanked out") texts, new, inserted text, etc. That ob-

[5] Tools being mostly languages: RSL, Petri nets [196,273,293–295], statecharts [144, 145,147,148,150], message [182–184] and live sequence charts [73,149,203], Timed RSL [120] (TRSL) and the duration calculus [381], or the latter as an extension to TRSL [120,155] (Vol. 2, Chaps. 10 and 12–15 incl.).

servation makes us model editing as a pair of functions from texts to texts: one "forward" function, e_f, and one "reverse" function, e_r, such that for any text d the following holds: $e_r(e_f(d)) = d$, and $e_f(e_r(d)) = d$. We leave the rest of the formal model for the reader to decipher the below formalisation.

Formal Presentation: A Document Formalisation

```
type
   D, O, V, C, L, T, Txt
   FWD, REV = Txt → Txt
   EDIT = FWD × REV
axiom
   ∀ (f,r):EDIT • ∀ d:D • r(f(d))=d=f(r(d))
value
   is_O: D → Bool
   is_V: D → Bool
   is_C: D → Bool
   create: Txt × L × T → O
   edit: EDIT × L × T → D → V
   copy: L × T → D → C
   trace: D → D*
axiom ...
```

The reader is encouraged to fill in the ...'s. ▪

Methodological Consequences: Principles, Techniques and Tools

Principles. The principle of *being formal* rests on a number of assumptions: that it is possible to formalise and that it brings benefit to formalise: that formalisations can be communicated and that formalisations can be used in further development, for verification and for theory formation.

Once these tenets hold, we can apply the principles of *formalisation:* Formalise, but do not over-formalise. Choose an appropriate abstraction level. Formalise what is beneficial to formalise: where the formalisation can be communicated and can serve as a basis for further development. ▪

Techniques. Volumes 1 and 2 cover many techniques of *formalisation.* Hence we shall not try here to boil the many subprinciples and subtechniques of formalisation down to a single paragraph. ▪

Tools. *Formalisation Documentation:* The tools mentioned in Sect. 2.4.10 apply here. But, in addition, quite sophisticated software packages are needed as tool support for formalisation. Each formal specification language usually comes with its own more or less complete set of tools: syntax tools: editors,

type checkers, and so on; analysis tools: theorem provers or proof assistants, model checkers, testing tools, and so on; and output tools: symbolic interpreters, compilers, and so on. For the documentation of formal models only the syntax tools are necessary. ▪

2.5.5 Discussion of Descriptive Documentation

Description is an art. Much can, however, be learned. Chaps. 5–7 will present principles, techniques and tools describing phenomena and concepts, by means of designations, definitions and refutable assertions. That is, although we shall say no more in the present chapter on "how to describe", we shall present much more material on this crucial subject in the next chapters.

Volumes 1 and 2 enunciate many principles and techniques for description (prescription, specification).

2.6 Analytic Document Parts

We remind the reader of our earlier characterisation of the concept of analysis:

Characterisation. By an *analysis* we mean a text which proves (i.e., reasons over), or designates, properties of some other text, or properties that are claimed to hold between pairs of texts. ▪

Analysis documents may be informal or formal. Formal analysis documents can only be achieved if the documents being analysed are themselves formal.

Pragmatics, Semantics and Syntax

The *pragmatics* of some analysis document, i.e., our reason for having analysis documents (or document texts), is that we wish to have a certain degree of trust in the single or paired documents that are being analysed. We wish to make sure that they satisfy desired, but not explicitly expressed, properties, including correctness. If we want a very high degree of trust, then the documents being analysed as well as the analysis document must be basically formal.

The *semantics* of formal analysis documents is typically that of a proof,[6] or a model check.[7]

The *syntax* of formal analysis documents is closely tied to the proof system or the model checking system of the formal specification language being used (i.e., proof rules, model checking property specification languages and their computerised support).

[6] Vol. 1, Chap. 9 briefly covered the notion of proof.

[7] We shall not cover formal verification or model checking in these volumes. For seminal textbooks cum monographs on model checking we refer to Gerard Holzmann's seminal [128, 172, 173].

Categories of Analytical Document Parts

We shall consider four kinds of analytical document parts:

- *concept formation* documents
- *validation*
- *verification, model check* and *test* documents, and
- *theory formation* documents

Analysing texts — for any of the purposes implied by the four kinds of analytical document parts — is a hallmark of software development. In other engineering branches the developer, the engineer, sets up, typically differential equations, as models of what is being developed. Analyses these, including performing calculations and even computations over them. In software development, mathematical, model-oriented or algebraic and logic specifications take the place of the civil, or mechanical, or aeronautics engineers' differential equations. And mathematical, algebraic and logical reasoning, i.e., analysis, can be objectively carried out.

In these volumes we shall ("officially") not deploy any formalisms, hence our analyses will be informal. In this, we may be said to not live up to the "hallmark". We have elsewhere stated the reasons for this omission: We basically believe that a series, as this, of general textbooks on software engineering, cannot contain such special material which has yet to find some final, more lasting formulations than can be expressed in connection with today's formal verification methods. Thus we refer the reader to follow up on this in coming years by looking for appropriately general textbooks on the subject of formal verification through proofs. Meanwhile we refer to the `RAISE` Method monograph [118].

2.6.1 Concept Formation

Characterisation. By *concept formation* — in the context of analytic software development documentation — we shall understand the creation, as the result of a study, typically of rough sketches, of a number of concepts, i.e., abstract notions of otherwise described phenomena, or other concepts. ∎

Analysis is partly an art. It takes training and experience to do it well. It involves abstraction. Concept formation is likewise an art. Only by being shown many examples can most students and practitioners of software engineering learn to form abstract, pleasing concepts.

Example 2.20 *Rough Sketch Analysis and Formation of Concepts:* We analyse the rough sketch freight logistics of Example 2.16. There are the concrete phenomena of trucks, freight trains, air cargo aircraft and line or bulk carrier boats. We abstract these into one concept: conveyors. There are the concrete

phenomena of trucking and freight train depots, of airports and of harbours. We abstract these into one concept: hubs. There are the notions of roads, rail lines, air traffic corridors, and sea lanes. We abstract these into one concept: routes. ∎

Pragmatics, Semantics and Syntax

The purpose, i.e., the *pragmatics*, of concept formation is to base the narrative and its formalisation on the abstracted, usually simplifying as well as unifying concepts — rather than on a variety of near-similar concrete phenomena. The concept formation document texts provide the link between the two levels of abstraction. The *semantics* of concept formation is really one of simplification through generalisation and substitution. The *syntax* of concept formation is not prescribed, but could amount to a series of pairs of texts: The (concrete usually manifest phenomenon designating) text to be generalised, and the abstract conceptual text serving as substitution.

2.6.2 Validation

Characterisation. By *validation* — in the context of analytic software development documentation — we shall understand a process, and the resulting (analytic) documents, in which some descriptive (prescriptive, or specification) documents are being inspected by stakeholders of the relevant universe of discourse, and in which whatever is being described (prescribed, specified) is being positively and/or negatively reviewed (i.e., positively and/or negatively criticised) by these, including the pointing out, if necessary, of inconsistencies, incompletenesses, conflicts and errors of description. ∎

We will deal with validation extensively in Chap. 14 (domain validation) and in Chap. 22 (requirements validation). That is: Validation (including its documentation) will take on a rather special role in this volume.

2.6.3 Verification, Model Checking, Testing

We use the terms verification, model checking, and testing as if they were the same. But they are not, as we shall see in Sect. 29.5.

Characterisation. By *verification*, *model checking* and *testing* — in the context of analytic software development documentation — we shall understand a process, and the resulting (analytic) documents, in which some descriptive (prescriptive, or specification) documents are being studied in order to ascertain whether what is described (prescribed or specified) in the analysed descriptive documents satisfies certain (claimed or otherwise expected) properties. ∎

Verification, model checking and testing is typically concerned with such things as: (i) Is a domain requirements prescription properly related to a claimed, underlying domain description? (ii) Does a software design specification represent a correct implementation of its requirements prescription, given the assumptions expressed in a related domain description?

Verification and model checking are essentially only meaningful if the various descriptions, prescriptions and specifications are expressed with a suitable mathematical rigour. Testing is only meaningful if the various descriptions, prescriptions and specifications can form the direct basis for computer executions (as can code). We earlier made statements as to why these volumes do not cover formal verification to any noticeable extent.

2.6.4 Theory Formation

Characterisation. By *theory formation* — in the context of analytic software development documentation — we shall understand a process and the resulting (analytic) documents, i.e., a study of some described universe of discourse with the objective of, and resulting in, the discovery (i.e., formulation), respectively proof (i.e., verification), of properties of the model, i.e., of the descriptions (prescriptions, resp. specifications). ▪

We shall not bring in theory formation examples, but we shall hint at domain theory formation in Chap. 15.

2.6.5 Discussion of Analytic Documentation

General

We have rather briefly surveyed the notion of analysis and analytic documentation. Since we do not bring in substantial material on formal verification, model checking or testing, we shall basically only be illustrating some concept formation. Domain and requirements validation is rather thoroughly covered in Chap. 14 (Domain Validation) and in Chap. 22 (Requirements Validation).

Methodological Consequences: Principles, Techniques and Tools

Although the present editions of these volumes do not provide substantial material on the subject of analysis, we shall nevertheless enunciate some method concerns.

Principle. *Analysis:* Hand in hand with description (prescription, specification), the developer, ideally, reasons about what the descriptive (prescriptive, specifying) documents are describing (prescribing, specifying). The developer does so in either of a number of ways: By systematic, but informal reasoning; or by rigorous reasoning, stating proof obligations, formulating lemmas

and theorems; or by formal reasoning, formally verifying stated lemmas, etc., model checking stated assertions, or testing these rigorously. ▪

Principles. *Analysis documents* must be convincing, definitive and final. ▪

Since we do not bring inany substantial material on analysis, we shall refrain from stating techniques and tools for the construction of analysis documents.

2.7 Discussion

2.7.1 General

Documentation is almost all! In the introduction to this chapter on documentation we claimed *documentation is all!*

So what is the difference? The difference is this: In order to document domains and requirements, we must acquire domain knowledge, respectively requirements expectations. And in order to come up with pleasing models (descriptions, prescriptions, and specifications), we must analyse. So *documentation is all* if we include the documentation of the acquisition and the analysis efforts — as we shall indeed argue should be done. Such arguments will be put forward in Chaps. 12–13, Chaps. 20–21 and Chap. 23.

2.7.2 Summary of Chapter

This chapter has conveyed the following: When developing, hence documenting software, the following documentation must be constructed: (i) informative documents, (ii) descriptive documents, and (iii) analytic documents. Figure 2.2 shows a structure of the various kinds of documentation relevant for any stage of development.

As concerns (i) informative documents, these are the kinds of possibly relevant document parts: (i.1) partners: clients and developer document parts; (i.2) current situation, needs, ideas and concepts document parts; (i.3) scope, span and synopsis document parts; (i.4) assumptions and dependencies + implicit/derivative goals document parts; (i.5) contract and design brief documents; and (i.6) logbook.

As concerns (ii) descriptive documents, these are the kinds of possibly relevant document parts: (ii.1) rough sketches, (ii.2) terminologies, (ii.3) narratives and, ideally, (ii.4) formalisations (although such will only be hinted at in this volume).

And as concerns (iii) analytic documents, these are the kinds of possibly relevant document parts: (iii.1) concept formation, (iii.2) validation, (iii.3) verification, model checking, testing and (iii.4) theory formation.

Figure 2.3 shows a structure of the various kinds of documentation relevant for an entire development. Please note our careful use of the two terms:

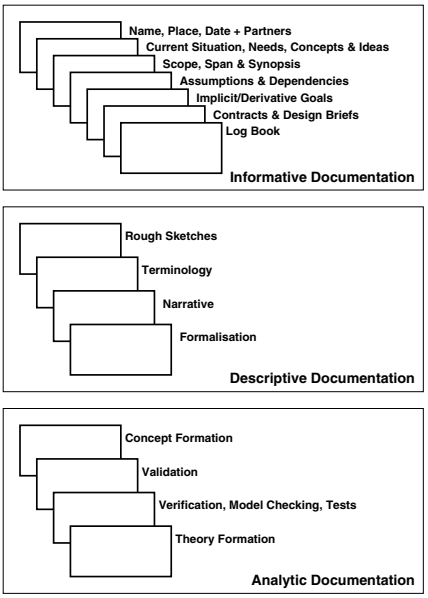


Fig. 2.2. A graphic overview of stage documentation

documentation of one kind or another, but not mixed. And hence, document as such a suitable "mix" of documentation.

The purpose of showing Fig. 2.2 is to structure your awareness of the multitude of document kinds. The purpose of showing Fig. 2.3 is to structure your awareness of the multitude of documents. In any typical development there will, indeed, be very many documents, and it is therefore of utmost importance to keep track of these documents, of their many versions and to be able to compose "configurations" of the right versions.

Methodological Consequences: Principles, Techniques and Tools

This series of textbooks on software engineering espouses the following principle of documentation:

Principles. *Documentation* is everything. *Document everything:* all information, all descriptions (prescriptions, specifications), all analyses. For all phases document domain models, requirements models and software designs. Maintain all documents, monitor and control all versions, and keep them updated, at almost any cost. ∎

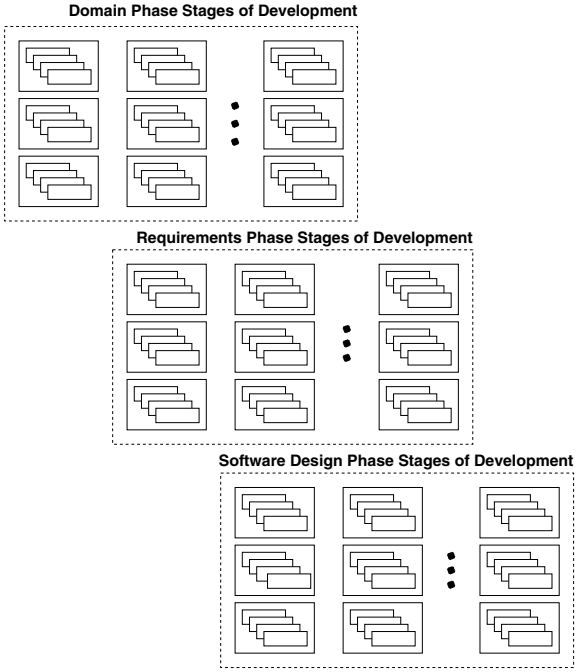


Fig. 2.3. A graphic overview of full development documentation

2.8 Exercises

2.8.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

2.8.2 The Exercises

Do not, at this early stage, shy away from attempting to solve these exercises. Admittedly, solving these exercises at this early stage, and given the genericity of the question (*for your chosen topic*), to answer the exercises requires some creative imagination. Later chapters will bring in more material and will thus enable you to embark on more meaningful, more satisfying solutions. Subsequent exercises, in remaining chapters of this volume, will repeat, somehow these exercises, but then in more elaborate forms.

Exercise 2.1 *Informative Domain Development Documents.* For the fixed topic, selected by you, draft a set of informative documents for a project which is to develop a description of the domain. Set aside one quarter page,

at most, for most of the part answers, and maybe up to one half page for some others (synopsis, design brief).

Exercise 2.2 *Informative Requirements Development Documents.* For the fixed topic, selected by you, draft a set of informative documents for a project which is to develop a requirements for some software to serve in the chosen domain. Reuse, as much as is possible, your answer to Exercise 2.1.

Exercise 2.3 *Descriptive Rough Domain Sketches.* For the fixed topic, selected by you, attempt a rough sketch of some area of the chosen domain. Set aside no more than one page for this. In preparation for Exercise 2.4, try formulate your rough domain sketch such that it lends itself to some simplifying concept formation — in the style, for example, of Example 2.20.

Exercise 2.4 *Concept Analysis of Rough Domain Sketch.* Given your answer to Exercise 2.3, and "inspired", perhaps, by Example 2.20, analyse, with a view towards forming one or more concepts, the rough domain sketch of Exercise 2.3.

Exercise 2.5 *Descriptive Domain Terminology.* On the basis of your answers to Exercises 2.3 and 2.4, establish a tiny terminology of some four to five terms — such that some terms rely on the definition of other terms.

Exercise 2.6 *Descriptive Domain Narrative.* On the basis of your answers to Exercises 2.3–2.5, formulate, over two to three pages, a well-structured narrative.

Exercise 2.7 *Table of Contents.* Draft a possible table of contents of all the documents to be developed during a project that develops a domain description, a requirements prescription and a software design.

Exercise 2.8 *Requirements for a Document Support Tool.* This chapter discussed a domain of document development. It focused on the entities (i.e., the documents or document parts) of such a domain. Rough-sketch requirements for a software package that assist developers in developing, maintaining (i.e., editing, versioning), distributing (including tracing the whereabouts of) development documents, etc.

Part II

CONCEPTUAL FRAMEWORK

3

Methods and Methodology

- The **prerequisites** for studying this chapter are that you have some experience in programming and that you have a minimum of curiosity and inclination towards understanding concepts such as those discussed in this chapter.
- The **aims** are to discuss the concepts of *method* and *methodology;* and to discuss the notions of method *principles,* method *techniques* and method *tools.*
- The **objective** is to help enable you to judiciously select among development principles, techniques and tools.
- The **treatment** is systematic and discursive.

Though this be madness,
yet there is method in it.

William Shakespeare, 1564–1616; Hamlet II, ii

From Merriam–Webster [238] we quote: By a *method* we understand "a procedure or process for attaining an object. As a systematic procedure, technique, or mode of inquiry employed by or proper to a particular discipline or art, a systematic plan followed in presenting material for instruction, a way, technique, or process of or for doing something, a body of skills or techniques". We shall shortly sharpen this characterisation.

3.1 Method

The notions of method and methodology are much bandied about: In "modern" times the term method has, unfortunately, been used instead of what we consider the more appropriate terms procedure or routine. By method we shall thus not mean what in so-called object-oriented programming approaches is called "method". Instead we shall follow the more classical definition: "some rules for engineering conduct", "some notation", or other, is claimed to be

a method. Some methods are claimed formal. In this section we take a first look at what might constitute a method. And we make a necessary distinction between method and methodology.

Characterisation. By a *method* we understand a set of principles for selecting amongst, and applying, a set of designated techniques and tools that allow analysis and construction of artifacts. ∎

Discussion. The selections (of analysis and synthesis techniques and tools) and some of the deployments (of these techniques and tools) are to be carried out by people. The principles are usually of such a nature as to guide the developer. They are not to interfere with that person's possible ingenuity and creativeness, that person's ability to discover, to reflect and be sceptical. Hence we cannot — ever — expect to get anywhere near a formalisation of such principles.

Therefore the term "formal method" is unfortunate. Better would be formal techniques and formally based tools. Even better, to paraphrase Michel Sintzoff,[1] would be to speak of logical or precise techniques and tools, as informal such are very much needed, but illogical or imprecise are not. ∎

3.2 Methodology

Characterisation. By *methodology* we understand the study of and knowledge about methods. ∎

Discussion. The two terms method and methodology are often used interchangeably, it seems, especially in the US. Since there are a number of so-called methods, such as for example: the `B` method [4], the `VDM` (Vienna [software] Development Method) [35, 36, 104] and the `RAISE` method [117, 118], it is relevant to compare these. Hence the concept of methodology. When combining formal techniques then there has been a methodology study behind such proposals. Examples include extending the original `VDM` specification language (`Meta-IV` [35]) with `ccs` (a language for expressing calculations over communicating processes) [244, 245], such as has been done [84, 99], or extending the `RAISE` specification language, `RSL` [117, 118], with timing into timed `RAISE` (`TRSL`) [379], or `TRSL` extended with `Duration Calculus` (`DC` [381, 382]) [155], or extending the `VDM` specification language, `VDM-SL`, with modularity (i.e., objects), into `VDM++` [93]. ∎

[1] Michel Sintzoff is Professor Emeritus at Université Catholique de Louvain, Belgium.

3.3 Method Constituents

Discussion. The above characterisation of the notion of method identifies the following concepts: principle, analysis, construction, technique, tool, and artifact. We need characterise these concepts. In the following we focus on domain descriptions as being the artifacts of interest. ▪

3.3.1 Principle

Characterisation. We quote from [345]: *A* principle *is an accepted or professed rule of action or conduct, ..., a fundamental doctrine, right rules of conduct.* ▪

Discussion. 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. This we also relegate to be implied by some principles, techniques and tools. ▪

3.3.2 Analysis

Characterisation. *Analysis* is performed on domain descriptions, requirements prescriptions or software specifications. There seem to be three kinds of analysis. First, we have informal validation or formal verification, including proof and model checking. This kind of analysis is performed typically on narratives[2], respectively on formal texts. Such analyses lead to statements (i.e., metalinguistic document texts) such as *such and such description text(s) denotes such and such properties* (i.e., *is correct* or *is not correct* [relating one part of the text to another], or *denotes an NP-complete problem,* and so on). Analyses performed on rough sketches, are not formalisable, but have the aim of forming concepts. ▪

[2] We take it for granted that software development (in the domain, requirements or software design phases, and for each refinement or other development stage within phases, and for steps within stages) aims at constructing a number of documents: (a) informative, (b) descriptive — both informal and formal — and (c) analytic. Within informal descriptions we distinguish between those that are [nondeliverable] rough sketches — where rough sketches often contain rough formalisations — and those that are narratives and terminologies. Informative documents inform about the development. Descriptions (prescriptions, specifications) present (i.e., describe, prescribe, resp. specify) a universe of discourse, as here, a domain (requirements, respectively software). And analyses present (i.e., analyse) descriptions (and so on); they are, in that sense, metalinguistic.

Discussion. Descriptions (prescriptions, specifications) describe (prescribe, specify) some universe of discourse (domain, requirements, resp. software). We may claim that we are analysing that universe, but really it is a model of that universe, in the form of some description (prescription, specification), that we analyse. ∎

3.3.3 Construction (or Synthesis)

Characterisation. By *construction* (or *synthesis*) we mean the creation of a description (prescription, specification), and thereby of a theory: a collection of properties that can be deduced from that description (prescription, resp. specification). The creation involves elicitation (acquisition), writing, analysis, rewriting, analysis again, rewriting, etc. ∎

Discussion. Writing informative or analytical documents may not be considered construction. They are necessary documents, but they do not describe manifest phenomena in the domain (etc.). ∎

3.3.4 Techniques

Characterisation. *Technique:* We quote from [345]: Method or technical skill. ∎

Discussion. Already here we see a possible conflict: Our characterisation of method involves the term technique, which by [345] is defined in terms of the term method. We shall use the term technique in the sense of the, or a, specific procedure, routine or approach that characterises the technical skill. This procedural meaning is also the intention of the above dictionary use of the term method. ∎

3.3.5 Tools

Characterisation. *Tool:* We quote from [345]: An instrument for performing mechanical operations, a person used by another for his own ends, . . . , to work or shape with a tool. ∎

Discussion. We shall use the term tool in a wider sense: Any language is a tool, so is paper and pencil, blackboard and chalk, and so is any software package. When we speak of tools for software development: From domain development, via requirements development to software design, we think of tools that support these activities: constructing, verifying and validating domain descriptions, requirements prescriptions and software designs. For such formal software development approaches as those of `B, RAISE, VDM, Z,` etc., there are several tools. ∎

3.4 Development Principles, Techniques and Tools

We further analyse some of the method concepts.

3.4.1 Some Metaprinciples

If, as we are now claiming, and as we have already done, extensively, in previous volumes of this series of software engineering textbooks, one can indeed identify a set of principles, techniques and tools that apply, conditionally, in a number of development situations, then these principles, techniques and tools ought probably also be deployed. Hence, we have:

Principle. *Methodicity:* Being *methodical* is now that of actually deploying relevant domain [and requirements] engineering as well as software design principles, techniques and tools during software development. ∎

Discussion. The hedge here is, obviously, the term relevant. There is thus another metaprinciple buried here. ∎

Principles. *Development Choices:* This principle is unconditional, that is, is part of every principle, technique and tool characterisation. Apply a principle, a technique or a tool only if its preconditions are met. ∎

The preconditions of deploying a principle are usually stated as part of the principle.

Techniques. *Methodicity:* This technique expresses that in respective phases of software development, one adheres to a list of principles, techniques and tools, ensuring that all due considerations are paid to these in the development. ∎

Discussion. This adherence list, just referred to in the above methodicity metatechnique paragraph (and as we shall later see) includes such, so far not explained, principles and techniques as: (i) general abstraction and modelling; (ii) domain attribute, perspective and facet; (iii) requirements principles and techniques: domain projection, instantiation, extension and initialisation; (iv) domain/man-machine interface; (v) machine requirements; (vi) software architecture; (vii) program organisation; and (viii) many other program design principles and techniques. ∎

Techniques. *Development Choice:* This technique expresses, relative to the previous 'methodicity' techniques, that for each of the items of the 'adherence' list of principles, techniques and tools, one carefully writes down the assumptions upon which a choice of specific principle, technique or tool was deployed. ∎

We shall, of course, in these volumes enunciate these conditionals explicitly.

3.4.2 Some Principles, Techniques and Tools

From the previous chapters we need to enunciate some principles, techniques and tools. We postponed enunciating them earlier since we needed the present chapter's systematic treatment of the concepts of method and methodology.

Type Principle, Techniques and Tools

Principle. For every entity (thing of value) being described determine its type. Hence, for every class (i.e., set) of alike entities ascribe that type to that class. ▪

The above principle may sound simple. But it is, in fact, deceptive. So watch out. What are these things anyway? Obviously we shall return to this matter in depth. For the time being those "things" are, for example, the manifest phenomena that can be pointed to in the universe of discourse, a domain, for example: cars, people, books, iron ingots, etc. They are not events occurring to these things, although we could claim so, and although we could associate types with events. And hence they are not many other things either!

Techniques. Initially sketch the type by informal words, then as a formal sort, possibly with, as we shall later see in more detail, observer and generator functions (but see principles and techniques of 'functions' below). Depending on level of abstraction, one may choose to then go on modelling the type or types more concretely, i.e., as Cartesians, as functions, relations, etc. ▪

Tools. For the time being we advise use of the `RSL` type modelling tool: `RSL` type definitions, initially as abstract types, i.e., sorts, eventually as concrete, i.e., model-oriented types (sets, Cartesians, functions, relations, lists, maps, etc.). ▪

Function Principle, Techniques and Tools

Principle. *Functions:* For every described entity, i.e., for every actual described class of entities of the same type, determine which functions that may apply to instances of them: What can be observed from such instances, i.e., such entities, and how to generate such entities. ▪

Techniques. *Functions:* We can, so far, model functions explicitly in terms of λ-functions, and implicitly in terms of axioms that usually relate the functionalities of several thus interrelated functions. ▪

Later we shall see additional means of defining functions.

Tools. *Functions:* So far, and mostly in the following, we use the `RSL` language tool to model functions. ▪

Formal Presentation: Function Modelling Tool

type
A, B
value
[0] f,f′,f″: A → B
[1] f(a) ≡ $\mathcal{E}$(a), f ≡ λa:A•$\mathcal{E}$(a)
[2] f′(a) **as** b, **pre** $\mathcal{P}$(a) **post** $\mathcal{Q}$(a,b)
axiom
[3] ∀ a,a′,a″:A •
... f″(a) ..., ... f″(a′) ..., ... f″(a″) ...,

The above shows three forms of function definitions in RSL. Form [1] admits two variants, both giving names to functions. This allows recursion — when expressed in the first of the two variants. Just stating the expression λa:A•$\mathcal{E}$(a) does indeed define a function. $\mathcal{P}$(a) and $\mathcal{Q}$(a,b), [2], are appropriate predicate expressions.

Relation Principle, Techniques and Tools

As it turns out, we shall not be using relations very much, if at all as a means to abstract actual-world phenomena, but rather as a means of explaining — especially nondeterministic — meanings of RSL and other specification language constructs. Relations, as a major database data structure, is, of course, well-known, and we shall much later have occasion to model relational database concepts.

Principle. *Relations:* When modelling nondeterministic functions do so in terms of something that essentially denotes *relations*. ▪

Techniques. *Relations:* A major technique in expressing a *relational meaning* is usually by defining nondeterministic functions. ▪

In a later chapter we shall return to nondeterminism.

Tools. *Relations:* So far, and mostly in the following, we use the RSL language tool. ▪

Formal Presentation: Relation Modelling Tool

Typically, when we, for example, write:

type
A, B

value
f1,f2,...,fn: A → B
g: A $\tilde{\to}$ B, g(a) ≡ f1(a) ⌈⌉ f2(a) ⌈⌉ ... ⌈⌉ fn(a)
h: A → B-**set**, h(a) ≡ {f1(a),f2(a),...,fn(a)}

we mean that f1,f2,...,fn are postulated to be deterministic functions. They each yield a functional, i.e., a determinate value for each argument. g is a nondeterministic function. The ⌈⌉ (nondeterministic choice) operation arbitrarily yields either its left or its right operand value. And h is a deterministic function: It always yields a set of from 1 to n values — depending on whether two or more of the fi, fj yield the same b value for some a.

Algebra Principle, Techniques and Tools

There are three "obvious" applications of algebras: (1) When describing structures, such as, for example, sets, Cartesians, lists, stacks, and queues; (2) when describing relations between syntactical and semantical structures, such as, for example, a language and its meaning; and (3) when describing the transition from one, say a more abstract structure to a more concrete structure, say the abstract data type of sets and sets represented as linked lists.

Principles. The *algebra* principle states: Apply algebraic concepts when a problem adheres to one of the three cases listed above. ∎

For these three cases the relevant algebraic concepts are: (1) An algebra as a set, A, and a (usually finite) set of operations, Ω; and (2–3) a homomorphism from one algebra to another.

Techniques. A commonly used technique for *modelling algebras* prescribes that one "lumps" (i.e., syntactically groups) together the specification of the carrier set A and the operations Ω. ∎

Discussion. The above may, to the "informal version" reader sound rather abstract. But it really is nothing but what most such readers are practicing when using their favourite object-oriented programming language: Modules, really, designate algebras. ∎

Tools. *Algebra:* The RSL means for specifying algebras is — usually — the **class** construct. ∎

Formal Presentation: Algebra Modelling Tool

Typically:

class
type

X, Y, Z, W, ...
value
f: X → Y, g: Y → Z, h: X × Y → W, ...
axiom
∀ x,x′:X, y,y′:Y, ... • ... f(x) ... g(y) ... h(x′,y′) ...
end

defines an algebra, or, as we shall call it, a class, consisting of carrier sets X, Y, Z, W, ..., and of operations f, g, h, ...

Logic Principle, Techniques and Tools

Principles. *Logics:* When describing phenomena in the actual world, or when prescribing requirements to software, or when specifying software (designs), *emphasise properties.* That is, use *logic.* ▪

Techniques. *Logic:* When defining functions, of domains, of requirements, or of software architectures, and so on, *emphasise* their **pre/post**-*conditions* or define them in terms of **axiom**s. ▪

Tools. *Logic:* Deploy the *higher-order logic* of RSL: use quantified predicates and avail yourself of being able to let their bound variables range over functions and predicates. ▪

Formal Presentation: Logic Modelling Tool

Using the higher-order logic of RSL schematically amounts to:

type
A, B, C, ...
P = A → **Bool**, Q = A × B → **Bool**
value
f: A × B → C
axiom
∀ a:A,b:B,p:P, ∃ q:Q•p(a)⇒(q(a,b)⇒f(a) **as** b **pre** p(a), **post** q(a,b))

3.5 Discussion

We have risked some debate as to whether the delineations of what might constitute a method — as outlined and deployed in this chapter — form a suitable basis for systematic to rigorous work. Since methods are to be deployed primarily by humans we prefer to characterise rather than to define.

Definitions seem to have something more definite, more absolute about them. Characterising seems "being more at ease". Some may argue that the method principles, techniques and tools that we shall continue to enumerate and investigate may unduly constrain the ingenuity of software developers: That having to follow these principles, to use those techniques, and to deploy those tools may stifle their creativity. We believe the contrary: that the principles set the developer free, that having recognised techniques and tools allows the developer to focus on concepts, and puts the mind to work on those. That is, the developer is engaged in thinking, rather than "bureaucratic" labouring.

3.6 Exercises

The exercises of this chapter are *closed book* exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions.

Exercise 3.1 *Method.* Characterise what this book means by a method. (Later check with Sect. 3.1.)

Exercise 3.2 *Methodology.* Characterise what this book means by a methodology (Sect. 3.2).

Exercise 3.3 *Principle.* Characterise what this book means by a principle (Sect. 3.3.1). Be aware that the answer is not that straightforward as there are several ways of characterising the term principle.

Exercise 3.4 *Technique.* Characterise what this book means by a technique (Sect. 3.3.4). Be aware that the answer is not that straightforward as there are several ways of characterising the term technique.

Exercise 3.5 *Tool.* Characterise what this book means by a tool (Sect. 3.3.5).

Exercise 3.6 *Metaprinciple.* List two software development metaprinciples and try characterise them (Sect. 3.4.1).

Exercise 3.7 *Some Development Principles.* List four to five software development principles and try characterise each of them (Sect. 3.4.2).

4

Models and Modelling

- The **prerequisite** for studying this chapter is that you have wondered about what it is you were really doing when, informally, or formally, you were describing domains, or prescribing requirements, or specifying abstract software designs.
- The **aims** are to discuss the concepts of *model* and *modelling,* to discuss the concepts of *iconical, analogical,* and *analytical models,* of *descriptional,* and *prescriptional models,* and of *extensional* and *intensional models,* and to discuss *uses of models.*
- The **objective** is to enable the reader to select the most appropriate purpose and style of model and modelling.
- The **treatment** is systematic and discursive.

4.1 Introductory, Context-Setting Remarks

4.1.1 Models Versus "Possible Worlds"

In the following we shall be using the term model. We shall be using the term as most mathematicians would use it. In contrast, researchers into and practitioners of knowledge engineering use the term possible world. To us these uses are not quite the same — but related.

Characterisation. (I) *Knowledge engineering*, to us, is the study of knowledge and belief, promise and commitment, etc., among agents. ▪

Characterisation. *Agents* typically are either humans of some domain or robots in that domain. ▪

Characterisation. (II) *Knowledge engineering*, then is a study of the logics that arise as the result of what some agent A_i knows or believes (promises or commits, $\oplus' P$), about what some agent A_j knows or believes (promises or commits, $\oplus'' Q$), about what some agent A_k knows or believes (promises or commits, $\oplus''' R$), about ..., where for some A_ℓ we may have that $A_i = A_\ell$. ▪

Characterisation. A *possible world* is an interpretation of free identifiers in the knowledge and belief (promise or commit, etc.) predicates P, Q and R (of the above characterisation). ▪

The decorated operator $\oplus$ stands for either *know, believe, promise, commit*, etc., i.e., are modal operators. In this chapter we shall not consider matters of knowledge engineering and shall thus only be using the term model (not possible world). We refer to [98] for a seminal treatment of *Reasoning About Knowledge*. We review the concept of knowledge engineering in the context of domain engineering in Sect. 11.8.3.

4.1.2 On Models of a Specification

We say that the natural numbers are a model of Peano's axiom system. And we say that plane geometry is a model of Euclid's axiom system.

If a function, say f, is defined such that, in one model, say m_i, for argument a, it may yield the result r, in another model, say m_j, for argument a, it may yield the result r', etc., and in yet another model, say m_k, for argument a, it may yield the result $r''\cdots'$, then we say that the definition of f itself has a set of models:

$$\{[f \mapsto [a \mapsto r, ...]]_{m_i}, [f \mapsto [a \mapsto r', ...]]_{m_j}, \ldots, [f \mapsto [a \mapsto r''^{\cdots'}, ...]]_{m_k}\}.$$

In the expression above the subscripts on the informally shown models of f designate the model from which that f model was derived. Thus the meaning of some identifier of a specification depends on the model of other identifiers.

Characterisation. 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*. ▪

We shall, colloquially, equate a specification with a model. Please observe that a specification may have many (possible) models.

4.1.3 Modelling

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

In this section we shall not cover principles or techniques for *identification* or *choice*, but only analyse the concept of model itself. Many other chapters in this series of software engineering textbooks cover principles and techniques for phenomena and concept *identification* and abstraction *choice*.

Thus a model is not that which it purports to model, but only an abstraction of it! In modelling we specify something syntactically, while expecting that the model we have in mind, in our head, somehow coincides with the denoted models of the specification.

4.1.4 Universes of Discourse

Characterisation. By a *universe of discourse* we mean that about which a group of people wishes to communicate. ▪

There are several kinds of universes of discourse. One set of universes of discourse is the *domains*. The domain is where the application, for which software may be desired, resides. Another set of universes of discourse is the *requirements*. Requirements are those documents which describe what we expect from desired software. And a final set of universes of discourse is the *software designs*. Software designs are either more or less abstract specifications that outline how the program code is to be organised and which functions it is to implement, or software is that program code itself. In software engineering we create successions of models: *domain* → *requirements* → *software designs*. Thus the combined universes of domain engineering, of requirements engineering and of software design, constitute the universe of discourse of software development, i.e., of software engineering. This series of software engineering textbooks has that as its universe of discourse!

4.2 Model Attributes

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

4.2.1 Analogic, Analytic and Iconic Models

Characterisation. An *analogic model* resembles some other *universe* than the *universe of discourse* purported to be modelled. ▪

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

Characterisation. An *iconic model* is an "image" of the *universe of discourse* that is the target of our attention. ▪

Example 4.1 *Analogic, Analytic and Iconic Models:* We lump three kinds of examples into one larger example:

- *Analogic models:* (1) The symbol, on the visual display screen of your computer, of a `trash can`, denoting an ability to delete files.[1] (2) A four-pole, electric circuit network of resistors, inductances, capacitors and current or voltage supplies can be used to analogically model some aspects of the behaviour of certain mechanical vibration and/or spring dampening aggregations. (3) A tomographic image of, say the brain, with its colour-enhanced "blots" is an analogic model of a cross section of that brain!
- *Analytic models:* (4) The differential equations whose variables model spatial x, y, z coordinates and the temporal t dimension, and whose constant, m, model the mass of a stone, may be an analytic model of the dynamics of the throwing of such a stone in a vacuum. (5) A description, in `RSL`, involving quantities that purport to model bank accounts, their balance, time, etc., may be an analytic model of a banking system — in the real world — provided the model reflects at least "some of the things that can go wrong" in actual life. (6) A graph with labelled nodes and weighted arcs may be used as a model of a road net with cities and distances between these, and can be used for the computation of shortest distances, etc.
- *Iconic models:* Typical iconic models are certain advisory or judicially binding traffic signs: (7) The roadside sign showing, typically in Sweden, an `Elk`, denoting that elks may be crossing the road ahead at any time; (8) the roadside sign showing an `automobile` (from behind) `"underlined" with two crossing S curves`, denoting that the road surface ahead may be slippery and hence that automobiles may spin out of control; and (9) the roadside sign showing a `crossed-out horn`, denoting that use of the automobile horn is not allowed.

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

Principles and Techniques

We outline tentative principles and techniques.

Principle. *Analogic Models:* This principle first expresses that a model (of something) is chosen to be analogic, and is typically invoked when informally explaining an unusual concept — by analogy to something more common. It then expresses that the chosen analogy must fit the situation; that its unambiguous connotation, relatively easily, must spring to mind. ∎

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

That is, abstract concepts are often informally explained "by analogy". We comment that we shall probably not be using analogic models very much. Maybe in rough-sketching and perhaps in narratives, as comments, but it seems unlikely that we need analogic formal models!

Principle. *Analytic Models:* This principle first expresses that a model (of something) is chosen to be analytic, and is typically invoked for either of the following cases: the analytic model (of a domain description) is to be the basis for a domain theory, or for the provably relatable development of a requirements prescription, or, when the model is that of a requirements prescription, for the likewise provably relatable development of software. It then expresses that the chosen analytic model must be so formulated as to allow a reasonably straightforward analysis, that is, computations, including proofs or model checking. ∎

Analytic Models

In a sense, all our formal models are analytic. Being formal, thus having access to proof systems and proof assistant tools, possibly theorem provers, possibly model checkers, and possibly, when a specification language is supported by sophisticated abstract interpreters, means that one can perform one or another form of analysis of the formal text and be supported in doing so by mechanised (i.e., computerised) means. So we conclude: With formal models we also have analytic models. This is in addition to the formal models possibly expressing things iconically and/or analogically.

Principle. *Iconic Models:* This principle first expresses that a model (of something) is chosen to be iconic. It is typically invoked when developing man-machine user interfaces. It then expresses that icons must fit the situation; that their unambiguous connotation must spring immediately to the mind of the user. ∎

That is, concrete, known phenomena are often informally visualised by suitably chosen icons. We first comment that we shall probably not find use of the principle of iconic models for other than man-machine interfaces (MMIs), and secondly comment that we shall treat the subject of interface requirements modelling in Sect. 19.5.

It is thus, on the background of now having understood the three kinds of models: analogic, analytical and iconic, that we see that our models usually invoke two or all three of these. They invoke analytic in development and iconic in user-machine interface; or analytic in development, analogic in informal explication (including description), and possibly iconic in user-machine interface.

We now turn to related, possible modelling techniques.

As concerns a formalisation of a model for which an analogue has been identified we can say this: When an analogue, in some *universe of discourse*,

has been identified, for a phenomenon or concept in some, therefrom distinct *universe of discourse,* and if some formal (specification of a) model already exists for the analogue, then that model is chosen, else we just have the standard problem of choosing a formalisation.

Techniques. *Analogic Model:* If a formal model already exists for an analogue of a problem, then that model is chosen — perhaps after some suggestive renaming of identifiers. If not, then only the informal narrative of the analogue can be re-edited, and a formal specification developed from scratch. ▪

It is in the nature of a model that one wishes to be analytic, that it be formalised. And that is independent of whether the model is also desired to be iconic and/or analogic!

Techniques. *Analytic Model:* If the model (of the phenomenon or concept) is (also) to be analytic, then the formal model of the imagery: the textual syntax, the graphical user interface with its diagrams, pictures and images, is also the analytic model. If the model (of the concept) is also to be analytic, and — from the above — we already have a formal model of the analogue, then that is the analytic model. Otherwise we must develop a formal specification from scratch. ▪

Iconic symbols are syntactic markers. Their appearance carries (i.e., is intended to carry, implicitly, tacitly) all of some semantics (and probably also some pragmatics).

Techniques. *Iconic Model:* Modelling iconic models is thus about syntax, i.e., modelling the syntax of text, diagrams, pictures and images. So we use BNF grammars, draw diagrams, exhibit pictures, etc. ▪

In Sect. 19.5.8 we shall illustrate how we abstract families of graphical user interfaces (GUIs).

Discussion

The first class of model attributes: analogic, analytic and iconic, forms one of three "orthogonal", i.e., more or less independent classes. We find that making the distinctions laid down by these three adjectives is useful. We hope you will also find the distinctions useful. To aid in mastering the class of these three attributes, we have put forward some related principles and techniques. They are somewhat speculative. And one, the author certainly, could wish for more succinctly expressed principles and techniques.

4.2.2 Descriptive and Prescriptive Models

Characterisation. A *descriptive model*[2] describes something already existing. ▪

Characterisation. A *prescriptive model*[3] models something as yet to be implemented. ▪

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

Examples

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

Example 4.2 *Descriptive and Prescriptive Models:*

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

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

Notice in the descriptive model the unhedged use of the verbs *consists, connects, is* and *are*. A description is *indicative:*[4] It tells *what there is*. Likewise

[2] *Descriptive*: factually grounded or informative rather than normative, prescriptive, or emotive [238].

[3] *Prescriptive:* to lay down a rule [238].

[4] *Indicative*: of, relating to, or constituting a verb form or set of verb forms that represents the denoted act or state as an objective fact [238].

notice in the prescriptive model the use of the (compelling) verbs *shall* and *must*. A prescription is *putative:*[5] It tells *what there will be.*

Principles and Techniques

We outline tentative principles and techniques.

Principles. *Descriptive models* shall find their main use in domain specifications. ▪

The above does not sound like a principle, but it is. The verb shall turns the statement into a principle: *"Thou" shall use descriptive mode when specifying domain properties!* One may rightfully argue that a software design specification is also, i.e., becomes, a description once the software code that is specified has been successfully completed as per the design specification. Be that as it may, we shall take the view that in order to make the pragmatic distinction between (i) domain models, (ii) requirements models and (iii) design models, we shall say that the first (i) is descriptive, the second (ii) is prescribed, and the last (iii) is specified.

Techniques. *Narrative Descriptive Models:* State facts; neither wishes, nor desires. Use *assertive*[6] language. ▪

Cf. our use of such verbs as *is*, *consists* of, *connects*, etc.

To describe, in natural, albeit professional language is not easy. One usually forgets the is's, the consists, etc. One slips into impressions. Formal languages, by their nature, namely being rather restrictive in what can be expressed, are well suited for descriptions, prescriptions and specifications. Informal languages, by their nature, namely allowing a rich vocabulary and virtually unrestrained ways of combining terms, require great care and discipline, i.e., constraints (as exercised by the human narrators), to achieve narratives that are ideally descriptive, or ideally prescriptive, etc. Note that a formal specification is not hedged: There are no such modalities[7] as is, shall, must, and so on. All is is!

Principles. *Prescriptive models* shall find their main use in requirements specifications. ▪

[5] *Putative:* of "putare" to think, assumed to exist [238].

[6] *Assertive:* disposed to or characterized by bold or confident assertion, i.e., the act of asserting: to demonstrate the existence of or to state positively usually in anticipation of denial or objection [238].

[7] *Modality:* the classification of logical propositions according to their asserting or denying the possibility, impossibility, contingency, or necessity of their content [238].

The above doesn't sound like a principle, but it is. The verb 'shall' turns the statement into a principle: *"Thou" shall use prescriptive mode when specifying desired software properties!* One may rightfully argue that a software design specification is also a prescription, up until the software code that is specified has been successfully completed as per the design specification.

Techniques. *Narrative Prescriptive Models:* State wishes and desires. Use such words as shall and must. ∎

To prescribe, in natural, albeit professional language is not easy. One usually forgets the shalls and the musts. One slips into impressions.

Discussion

The second class of model attributes, descriptive and prescriptive, forms another of three orthogonal, i.e., more or less independent classes. We find that making the distinctions laid down by the two adjectives descriptive and prescriptive, is useful, and we hope you will find them useful too. To aid in "mastering" the class of these two attributes, we have put forward some related principles and techniques. They are somewhat speculative. And one, the author certainly, could wish for more succinctly expressed principles and techniques.

4.2.3 Extensional and Intensional Models

Characterisation. An *extensional model*[8] (black, opaque box) presentation models something as if observed by someone external to the *universe of discourse*. ∎

Characterisation. An *intensional model*[9] (glass (or white), transparent box) presentation models the internal structure of the *universe of discourse*. ∎

[8] *Extensional:* concerned with objective reality

[9] *Intensional:* in logic, correlative words that indicate the reference of a term or concept. Intension indicates the internal content of a term or concept that constitutes its formal definition.

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

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

Denotation: a direct specific meaning as distinct from an implied or associated idea.

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

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

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

Example 4.3 *Extensional Model Presentations:* (1) To explain the square root function, $\sqrt{n} = r$, by explaining that $r \times r = n \wedge \ r \geq 0$ is to give an extensional definition, hence model.

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

Example 4.4 *Intensional Model Presentations:*

(1) To explain the square root function, $\sqrt{n}$, by presenting, e.g., the Newton–Raphson algorithm ([285] pp. 230, 347, 355 and 360), is to give an intensional definition, hence model.

(2) An intensional model of stacks, see item (2) of Example 4.3 may model (a) stacks as lists of (extensionally modelled) elements, and define (c) the (i) empty, (ii) pop, (iii) top, and (iv) push functions in terms of (i) constructing the empty list, of (ii) yielding the tail of a list, of (iii) yielding the head element of a list, and (iv) of concatenating a supplied element to the front of the list — with (b) same signatures as in Example 4.3. ∎

Principles and Techniques

We outline tentative principles and techniques.

Principles. In early phases, stages and steps of development choose *extensional models*. ∎

That is, domain models are usually, desirably extensional. And so are requirements models — some are carried over, as in domain requirements, from the

domain models. And even in first stages of software design, likewise inherited models may remain extensional. When new, auxiliary, implementation-oriented software designs are introduced they may also often first be extensionally defined.

Techniques. *Formal extensional models* are developed in terms of property-oriented features of the specification languages, that is, algebraically, through sorts, function signatures and axioms. ▪

It is not as simple as just that. Some models may be more easily expressed, shorter, and still retain a reasonably high level of abstraction even though they are formulated using the techniques advised for intensional modelling, see below. And, usually hand in hand with the above, developers too easily slip into model-oriented modelling, perhaps a bit too early!

Principles. As machine requirements are being implemented in some software design, previously extensionally defined model parts are usually refined into *intensional models*. ▪

As we noticed above, the ideal demand may be just too idealistic! So refinements into intensional models may take place even in domain modelling. So, from a pragmatics point of view, we had better allow for heavy doses of intensionality early on!

Techniques. *Formal intensional models* are developed in terms of model-oriented features of the specification language, that is, from applicative, via imperative and parallel specification programming, and using set, Cartesian, list and map abstractions — before eventually "translating" low abstraction-level specifications into, for example, `Java` code. ▪

Discussion

The third class of model attributes, extensional and intensional, forms another of three orthogonal, i.e., more or less independent classes. We find that making the distinctions laid down by the two adjectives extensional and intensional, is useful, and we hope you will find them useful too. To aid in "mastering" the class of these two attributes, we have expressed some related principles and techniques. They are, as for descriptive and prescriptive, somewhat speculative. We could wish for more succinct expressions.

4.3 Roles of Models

We pursue modelling for one or more reasons:

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

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

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

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

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

Principles. *Model Roles:* It is very important that the client/developer contract clearly spells out for which number of usually many roles (i.e., purposes) a model is developed. ∎

4.4 The Modelling Principle

Finally we emphasise the most crucial aspects (α, β, γ) of models and modelling:

Principles. The *modelling* principle hinges upon: (α) There is the *problem domain:* the a priori given *universe of discourse;* there is (β) (the *mathematical structure* of) the *model* and there is (γ) the *identification* of the relationship between the *problem domain* and the *model*. The principle expresses that the developer must constantly keep the above triptych in mind. ∎

Discussion. The model is not the universe of discourse, only a representation of it. Whatever properties can be argued about the model are not necessarily properties of the universe of discourse. ∎

4.5 Discussion

Models and modelling are metaconcepts. As a software engineer, your job is to create models, not to be concerned with the concepts of models and modelling as subjects, but with specific, instantiated models of something specific. We have enunciated a number of model attributes: iconic, analogic and analytic; descriptive and prescriptive; and extensional and intensional. We have, for each of these, enunciated modelling principles, and modelling techniques. We hope that the ones we have proposed are found useful. But we are not quite sure! That is, we could wish for sharper forms of expression for

all the principles and techniques put forward in this section! Much programming methodological research, exploration and experimental work need still be done!

4.6 Exercises

Exercisess 4.1–4.3 are closed book exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions.

Exercise 4.1 *Model.* Characterise what this book means by a model.

Exercise 4.2 *Modelling.* Characterise what this book means by modelling.

Exercise 4.3 *Categories of Model Attributes.* List three classes of model attributes by listing their (two or three) members.

Exercise 4.4 *Analogic Models.* Can you think of other examples of analogic models than those already presented in Example 4.1? Please describe such analogic models. You may select them from any branch of human endeavour.

Exercise 4.5 *Analytic Models.* Can you think of other examples of analytic models than those already presented in Example 4.1? Please describe such analytic models. You may select them from any branch of human endeavour.

Exercise 4.6 *Iconic Models.* Can you think of other examples of iconic models than those already presented in Example 4.1? Please describe such iconic models. You may select them from any branch of human endeavour.

Exercise 4.7 *Prescriptive Models.* Can you think of other examples of prescriptive models than those already presented in Example 4.2? Please describe such prescriptive models. You may select them from any branch of human endeavour.

Exercise 4.8 *Descriptive Models.* Can you think of other examples of descriptive models than those already presented in Example 4.2? Please describe such descriptive models. You may select them from any branch of human endeavour.

Exercise 4.9 *Extensional Models.* Can you think of other examples of extensional models than those already presented in Example 4.3? Please describe such extensional models. You may select them from any branch of human endeavour.

Exercise 4.10 *Intensional Models.* Can you think of other examples of intensional models than those already presented in Example 4.4? Please describe such intensional models. You may select them from any branch of human endeavour.

Part III

DESCRIPTIONS: THEORY AND PRACTICE

This part contains three chapters:

- Chap. 5: *Phenomena and Concepts*
- Chap. 6: *On Defining and on Definitions*
- Chap. 7: *Jackson's Description Principles*

They are strongly related. It is all about the crucial transition from a world that is conceived by humans, observed, to a written document that describes, prescribes or specifies that world. Different individual stakeholders may conceive what is claimed to be the same world differently. And the document written by, say, a software engineer (also a stakeholder), when read by the other stakeholders may lead to disagreements.

One aim of these three chapters is to make sure that whoever writes down a description (prescription or specification) of that world follows a common style and basically attempts to describe (prescribe or specify) the same things according to commonly agreed principles, and using commonly agreed techniques and tools. In this way any difference of opinion is no longer one of style, but of substance.

5

Phenomena and Concepts

- The **prerequisite** for studying this chapter is that you are willing to think, and are able, or at least wish to think abstractly.
- The **aims** are to introduce basic principles and techniques for "discovering" *phenomena* that need *conceptualisation,* in particular, to introduce those phenomena whose conceptualisation is in terms of *entities* (information, data), *functions* (and relations), *events* (asynchronous and synchronous) and *behaviours,* and to introduce basic principles and techniques for the description of such phenomena and their underlying concepts.
- The **objective** is to intellectualise the reader, as necessary, but not yet sufficiently enough for the reader to become an effective professional software engineer.
- The **treatment** is from systematic to rigorous.

5.1 Introduction

In this chapter we shall cover a first set of facets of the concept of description: the problem of identification, that is, of being able to identify or delineate, (i.1) *phenomena* and (i.2) *concepts*, of interest; that is (i.1) *physically manifest things*, and (i.2) *mental constructions*.

We shall try to wrestle with some abstract ideas. They impinge upon *what can be known, and what can be described.* As such these abstract ideas border on *philosophy*, in particular such philosophical disciplines as *epistemology*, and *ontology*. For this reason we cannot treat these ideas with the kind of certainty usual in a discourse of mathematics or the natural sciences, but must be prepared for a certain degree of uncertainty!

5.2 Phenomena and Concepts

In this chapter we shall thread a careful course. We do not wish to establish a brand new theory of phenomena. We certainly do not wish to entertain such

ideas as object-orientedness, conceptual schemas, or whatever. We simply wish to go as far as very simple mathematics can support us. By that we mean: types and values, functions, events and behaviours. No further!

But first we discuss the ideas of phenomena and concepts.

We think it important that the professional software engineer clearly understand these two notions (that is, metaconcepts), and that they are not confused.

5.2.1 Physically Manifest Phenomena

In the world there are the physically manifest *phenomena*. We can sense them: touch, see, hear, feel, smell and taste them. Or we can measure them: mechanically, electrically/electronically, chemically, etc. Thus we can point to them and designate them, in one way or another.

5.2.2 Mentally Conceived Concepts

We often abstract a phenomenon into a *concept*.

Example 5.1 *Automobile Phenomenon Versus Car Concept:* The specific phenomenon of "that automobile" is abstracted into the type, the class, the set of all cars, i.e., the concept *car*. ∎

The above example illustrates the concrete *value concept* ("that automobile") versus the abstract *type concept* (*car*).

5.2.3 Categories of Phenomena and Concepts

For pragmatic reasons we categorise phenomena and concepts into four categories: (i) entities — their values, properties and types; (ii) functions over entities; (iii) events (related to behaviours); and (iv) behaviours (as traces of events and function applications, i.e., actions). These category names represent *metaconcepts*.

Characterisation. By a *phenomenon* we shall loosely understand some physical thing that humans can sense or which natural science based technology can measure, and where a phenomenon is typically a natural thing or a human-made artifact. ∎

Characterisation. By a *concept* we shall loosely understand a mental construction — conceived by people — which, in an abstract manner captures an essence of usually a class of phenomena or a class of concepts. ∎

5.2.4 Concrete and Abstract Concepts

Phenomena can be pointed to or measured. They are physical. Concepts are mental constructions. When we "lift" our consideration, for example, from *that specific car over there*, i.e., of a phenomenon, to a representative of the set of cars, then we have abstracted "away" from a specific phenomenon to the concrete concept of car. When we "lift" our considerations, for example, from those of dealing with cars, trains, airplanes or ships, that is from a set of concrete concepts, to consider specimens (i.e., instances) of these as vehicles (or conveyors), then we consider vehicles (conveyors) as abstract concepts. And so on.

Characterisation. By a *concrete concept* we mean an abstraction of a set of similar phenomena into one concept. ▪

Given a description of a concrete concept we can speak of its concretisation as an identification which establishes a relation between the concrete concept and at least one phenomenon which is intended covered by the concrete concept.

Characterisation. By an *abstract concept* we mean an abstraction of a set of similar concepts into one concept. ▪

Given a description of an abstract concept we can speak of its concretisation as an identification which establishes a relation between the abstract concept and at least one concrete concept which is intended covered by the abstract concept.

Discussion. When we say "a car" we mean, not a specific one, that is, not a specific phenomenon, but a concrete concept, not a set of cars, but a single instance (the generic car). If we say "those cars there" we mean a set of phenomena. And if we say "consider a set of cars" then we mean a set of concrete concepts — for short: a concrete concept. We can describe a set, s, of values, a, all of which are of some type A. In that case the set s is a value of type A-**set**. ▪

5.2.5 Categories of Descriptions

Usually we shall be formulating our descriptions in terms of concepts, not in terms of phenomena. But occasionally it is required that a description in terms of phenomena is developed and presented.

Characterisation. A *description* is said to be *phenomenological* if all of its entities, functions, events and behaviours are of phenomena. ▪

Characterisation. A *description* is said to be *conceptual* if all of its entities, functions, events and behaviours are of concepts. ▪

Characterisation. A *description* is said to be *specific problem-oriented* if all of its entities, functions, events and behaviours are of some mixture of both phenomena (one or more) and concepts (one or more). ▪

Discussion. *Realistic Descriptions:* Most actual domain descriptions are carried out in connection with specific customer domains. Such customer domains are "almost" instances of a more general, that is, a generic and hence conceptual domain, and hence possibly of a conceptual domain description. But usually this (or these) conceptual domain description(s) do not exist. And all the customer is willing to finance, typically for reasons of competitiveness of the customer enterprise, is the specific problem-oriented description. In such customer-biased descriptions one is not to conceptualise a number of instances of phenomena but to keep the phenomena-specific. ▪

Example 5.2 *A Specific Problem-Oriented "Toy" Description:* The *South Coast Rail Line* regional railway net consists of a single linear route, r, made up from six stations, $s_1, s_2, s_3, s_4, s_5, s_6$, and five consecutive lines, $\ell_{12}, \ell_{23}, \ell_{34}, \ell_{45}, \ell_{56}$, such that route r can be thought of as a sequence of alternating stations and lines: $\langle s_1, \ell_{12}, s_2, \ell_{23}, s_3, \ell_{34}, s_4, \ell_{45}, s_5, \ell_{56}, s_6 \rangle$ or the other way around: $\langle s_6, \ell_{56}, s_5, \ell_{45}, s_4, \ell_{34}, s_3, \ell_{23}, s_2, \ell_{12}, s_1 \rangle$. More specifically line ℓ_{12} is 17 km long, line ℓ_{23} is 19 km long, ..., and line ℓ_{56} is 23 kms long. Topologically and geodetically line ℓ_{12} runs as follows: ... (etcetera). Stations s_1 is named Arlington, s_2 Burlington, ..., and station s_6 is named Georgetown. Topologically and geodetically station s_1 is organised as follows: ... (etcetera). Etcetera. ▪

5.2.6 What Is a Description?

Finally we are ready to address the crucial issue of what a description is.

What Is a Description?

Characterisation. By a *description* we mean some text which either designates a set of phenomena in such a way that the reader of the description can recognise these phenomena from the description or designates a set of concepts in such a way that the reader of the description can concretise these into recognisable phenomena, or it is a text which designates both recognisable phenomena and recognisable concepts. ▪

Chapter 7 covers (i) recognisability of descriptions, (ii) the issue of providing as few phenomena (or actually concrete concept) descriptions as possible (i.e., the so-called "narrow bridge"), (iii) the issue of instead relying on definitions and (iv) the issue of risking refutable descriptions.

5.3 Entities

One man's entity is another man's function!

We hedge the opening of this section by a caveat, and a veiled warning. When we now shall try to characterise what an entity is, it must be understood as a choice that the developer has to make of whether to consider a phenomenon or a concept as an entity or as a function. Usually that choice is an easy one to make. Colloquially speaking: if you think of the phenomena or concepts as information typically computerisable as data, then the phenomena or concepts are entities. Similar remarks can be made for the possible relations between entities and behaviours.

Characterisation. By an *entity* we shall loosely understand something fixed, immobile or static. Although that thing may move, after it has moved it is essentially the same thing, an entity. ∎

From a pragmatic point of view, entities are the "things" that, if implemented inside computers, could typically be represented as data.

Example 5.3 *Entities:* We give some examples: a (specific) pencil, a (specific) chocolate bar, a (specific) pail of paint, a (specific) person, a (specific) railway net, a (specific) train or a (specific) transport industry. ∎

We make the distinction between atomic entities and composed (i.e., composite) entities.

5.3.1 Atomic Entities

Characterisation. By an *atomic entity* we shall understand an entity which cannot be understood as composed from other entities. ∎

Example 5.4 *Atomic Entities:* We give a few examples: a (specific) pencil, a (specific) chocolate bar, a (specific) person or a (specific) timetable. ∎

If I take the pencil apart, say into its lead core and wooden frame, then these parts are not really proper entities. The taking apart may, for one, have broken the lead core or damaged the wooden frame, and, in any case, the two parts serve no function other than entering into an atomic whole. Similarly for a person: Considering the head, limbs, etc., of a person as separate entities may only make sense in surgery (organ transplantation, etc.), but would render our concept of a whole person somewhat at risk. But this last example is actually well chosen, as one of deciding whether some entity is atomic or not: the decision seems, as here, to depend on the viewpoint on — the context in which we view — the entity.

5.3.2 Composite Entities

Characterisation. By a *composite entity* e we shall understand an entity which can best be understood as composed from other entities, called the subentities, $e_1, e_2, \ldots, e_n$, of entity e. ▪

Example 5.5 *Composed Entities:* We give a few examples: a (specific) railway net (as composed from lines and stations), a (specific, say passenger) train (as composed from passenger cars and engines (locomotives)) or a (specific) transport industry (as composed from the transport net, the transport vehicles, and so on). ▪

5.3.3 Subentities

Characterisation. By a *subentity* we shall understand an entity which is a component of another entity. ▪

Example 5.6 *Subentities:* We give a few examples: the lines and stations of a (specific) railway net, the locomotive(s) and cars of a (specific) train, the railway net of a (specific) railway system. ▪

5.3.4 Values, Mereology and Attributes

Examples 5.4–5.6 designated certain entities. For each of them we can speak of a *value* of that entity. And for each of them we can speak of zero, one or more *attributes* of that entity. A concept of *mereology* is associated only with composite entities and then expresses how the entity is composed from subentities.

Characterisation. By a *value* v_e of an entity we shall loosely understand the following: If the entity is an atomic entity, then the entire set of identified attributes, $a_{1_e}, a_{2_e}, \ldots, a_{n_e}$, of the entity. If the entity is a composite entity (thus consisting, say, of subentities, $e_1, e_2, \ldots, e_m$) then there are three parts to the entity value: how it is composed — its mereology m, the entire set of identified attributes, $a_{1_e}, a_{2_e}, \ldots, a_{n_e}$, of the entity, and (inductively) the identified values, $v_{e_1}, v_{e_2}, \ldots, v_{e_m}$, of respective subentities $(e_1, e_2, \ldots, e_m)$. ▪

Characterisation. By an *attribute* of an entity we shall loosely understand a quality which cannot be separated from the entity. ▪

Example 5.7 *Attributes and Values:* The following are some of the attributes of an atomic pencil entity: the physical length of the pencil, the materials from which it is made, the colour of the (lead) pen in the pencil, all the other

attributes associated with the appearance of the pen (wear and tear), its purchase price, etc. The value of the atomic pencil is thus the (mereological) fact that it is an atomic entity, and the sum total of all the above (including "etc.") attributes. ▪

5.3.5 Entity Mereology

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.

Characterisation. By the *mereology of an entity* we shall loosely understand whether it is atomic, or, when it is composite, then how it is made composite (i.e., from which kind of subentities it is composed). ▪

Please note our attempt to distinguish between entities, entity values, entity attributes and entity mereologies. Note that we use the term attempt. You may rightfully claim that an entity, as observed, is equal, i.e., synonymous, with its value.

Example 5.8 *Mereology Facets of a Railway Net:* The mereology of a railway net transpires from the *italicized* terms. A net is *composed from,* and hence *decomposable into* a *collection* of lines and a *collection* of stations. Any line *links exactly two distinct* stations. Any station *is linked to one or more distinct* lines. A rail line is *composed from* linear rail units. The connectors of a rail unit are (here considered) *inseparable from* the *pairs of* rails (and the ties and their nails, etc.) making up the main part of rail units. We have not said anything above about how the collections form nets. But such formation rules are part of the mereology. ▪

There is no end to the kind of attributes and the form of mereology one may eventually associate with an entity. And for that matter, one can also associate attributes and mereologies with functions and behaviours, as we shall see. It is not productive, we strongly believe, to try enumerate all the possible categories of mereologies and attributes — and hence information — that one may associate with entities (functions and behaviours). One easily becomes lost in philosophical discourse, cf. [340]. Our job is mainly constrained by putting whatever we domain-analyse inside the computer. Hence, in the final analysis we need just resort to what can be described in terms of abstractions of computer data, i.e., values and types, computing routines (algorithms) and computing processes. Thus we shall mainly focus on such mereologies which can be informally, but precisely described, in natural English and which can be mathematically described, or, further constraining the issue of mereologies, which can be represented inside the computer.

5.3.6 Mereologies and Attributes

So an entity e value v_e is made up, in a loose sense, of (i) its mereology — (i.a) whether atomic or (i.b) composed from subentities, and then how — (ii) entity attributes, $a_{1_e}, a_{2_e}, \ldots, a_{n_e}$, and (iii), when the entity, e, is composite, then (inductively) the values, $v_{e_1}, v_{e_2}, \ldots, v_{e_m}$, of these entities $(e_1, e_2, \ldots, e_m)$.

We shall model all this information as values of appropriate types.

5.3.7 Model-Oriented Mereologies

Volumes 1 and 2 of this series of volumes focused on property- and model-oriented ways of specifying mereologies and attributes. In those volumes, except in Chap. 2 in Vol. 2 (*hierarchies and composition*), we did not single out the concept of mereology. But the model-oriented means of modelling composite entities as sets, Cartesians, lists, maps and functions had that intention.

5.3.8 Model-Oriented Attributes — An Aside

Atomic Types and Values

We could consider the constraints that might be put on a model-oriented mereology for some entity either as a property of the mereology or as an attribute of the entity. In addition to this, there are the end types of the atomic entities that eventually make up any entity. The actual values of these atomic types, in our view, constitute the attributes of the entity.

Other Attributes

But, as amply shown in, for example, Vol. 2, there are seemingly composite types that model what we should like to classify as attributes rather than as mereologies: functions that model temporal progression, e.g., traffic, functions that model denotations, and so on. We shall therefore have to resign, for this volume, and say: for a treatment of the proper modelling facets of entities we must refer to the entirety of both Vols. 1 and 2 of this series.

5.3.9 Entity Properties

So an entity value is made up, roughly speaking, of its mereology, its attributes and (if composite) the values of all subentities. We shall, for convenience, lump the two facets, mereologies and attributes, into one concept: properties.

General

To repeat: physically manifest things have values. That is, we take some values to be values of physically manifest things. Physically manifest things, in addition, have mereologies: the value of a physically manifest thing is the sum total of its mereology and all its attributes. Properties (mereology and attributes), like values (in general), are of types and are expressed in terms of (usually nonthing) values.

"Thing" Properties

If it helps, you may divide the world of properties into two kinds: (i) *thing mereologies:* a property of a usually composite manifest phenomenon, that it is a manifest atomic or composite phenomenon itself; and (ii) *attributes:* a property of a manifest phenomenon, which is not a manifest phenomenon. It is more like a property. Both kinds of properties are represented by a type and a value.

Example 5.9 *"Thing" Properties:* A particular railway net is a manifest phenomenon. Any line or station, and, for that matter, any smallest rail unit and many things in between, like platform or siding tracks, are also manifest phenomena, and are thus *thing properties* of the net phenomenon. The curvature of a rail unit, we may claim, is an attribute (i.e., also a property, a concept) of that rail unit. The curvature itself cannot be taken apart from the rail unit and manifested as such. Other examples of railway net *attributes* are: the length of a line; that a unit of a line is closed for traffic; and that a line, when open, can ever only be open in one direction (an up line, as opposed to a down line, for lines connected to a station [into, respectively out from]). ∎

5.3.10 Real Examples and Our Type System

For those readers who have not studied previous volumes of this series we bring in some formal examples of model-oriented type (and value) specifications.

Set Compounds

Example 5.10 *Communities and Community Networks: Intra and Inter:* First an informal description: A community, C, consists of a finite set of one or more people, P. A society, S, consists of a finite set of one or more disjoint communities. An intracommunity network, Intra, consists of a finite set of one or more people within a community. An intercommunity network, Inter, consists of a finite set of one or more people, exactly one from each of a subset of communities of a society.

Then, we give a formal description.

Formal Presentation: Communities and Community Networks

type P
  C = P-**set**
  S = C-**set**
    such that for any two distinct elements c_i, c_j
    of some s in S they share no people in common
    ∀ s:S • ∀ ci,cj:C • {ci,cj}⊆s ∧ ci≠cj ⇒ ci ∩ cj = {}
  Intra = P-**set**
    such that for any i in Intra there exists a com-
    munity c of some s in S of which i is a subset
    ∀ i:Intra • ∃ s:S • ∃ c:C • c ∈ s ⇒ i⊆c
  Inter = P-**set**
    such that for any i in Inter of n elements there exist n
    distinct communities $c_1, c_2, ..., c_n$ of some s in S such
    that each member of i ≡ exactly a member of a respective c_k
    ∀ i:Inter • **card** i = n ⇒
      ∃ s,cs:S • cs⊆s ∧ **card** cs = n ∧ unique(i,cs)
value
  unique: Inter × S → **Bool**
  unique(i,cs) ≡
    ∃ ! p:P,c:C • p ∈ i ∧ c ∈ cs ∧ unique(i \ {p},cs \ {c})
    pre card i = **card** cs

Let suitably subscripted p's designate distinct persons, then:

(1) {{p1,p2,p3},{p4,p5},{p6}}
(2) {p1,p2}, {p5}, {p6}
(3) {p1,p4}

Respectively designate a possible: (1) society, (2) three intracommunity networks and (3) an intercommunity network.

The mereology of the community, society, intracommunity and intercommunity concepts transpire from the above use of the powerset operator (-**set**) and the "such that" constraints. The attributes of the community, society, intracommunity and intercommunity concepts amounts to the cardinality of the involved sets.

■

We may thus use the following pseudo RSL-notation:

type	**type** designates type definitons
A	A is a sort, i.e., abstract type
S = A-**set**	S is a concrete type of sets of A elements

value	**value** designates value definitions
a,b,...,c:A	a,b,...,c are elements of A
{}:S	empty set is in S
{a,b,...,c}:S	set of a,b,...,c is in S
set1∪set2:S	set of elements, in S, of set1 or set2 or both
set1∩set2:S	set of elements, in S, of both set1 and set2
set1⊆set2:**Bool**	set set1 is a subset of set set2
e ∈ set:**Bool**	element e is a member of set

The set operators can be "pronounced" as ∪: union, ∩: intersection, ⊆ (and ⊂): subset (proper subset), and ∈: is a member of.

Cartesian Compounds

Next we give an example illustrating Cartesian compounds.

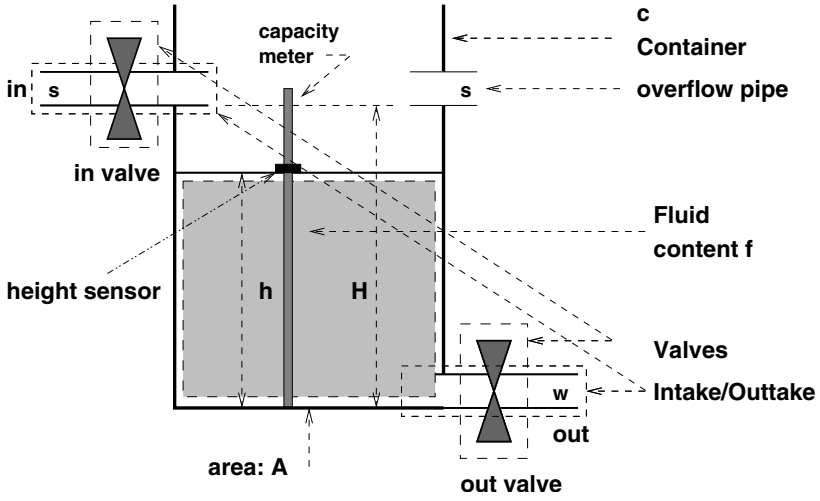


Fig. 5.1. A boxlike fluid container system

Example 5.11 *A Fluid Container Configuration:* We illustrate a boxlike fluid tank in Fig. 5.1. We have the *physically manifest phenomena:* The whole tank, as illustrated, is a physically manifest phenomenon. We can, and will, claim that so are the in(take), overflow and out(take) valves, the height sensor, and the liquid (if any) in the tank. And there are the *attributes:* The height (of the liquid) is an attribute, and so are the open/closed status of the in(take) and out(take) valves.

We can speak of the "value" of the entire fluid container system as composed from context and state components. The context components are the fixed (valued) attributes of the physical fluid container: the width, depth and overall height, i.e., the physical (volume) measurements, the height, over the bottom of the container, of the in(take) valve, the height, over the bottom of the container, of the out(take) valve, and the height, over the bottom of the container, of the overflow pipe. The state components are the variable (valued) attributes of the physical fluid container: the height, over the bottom of the container, of the liquid level, and the open/closed states of the two valves.

We can formalise this:

Formal Presentation: A Fluid Container Configuration

type
LCS = CONTEXT × STATE

The fixed-valued context can be modelled as:

type
CONTEXT = VOL × VALS × OFLOW
VOL = wdth:**Real** × dpth:**Real** × hght:**Real**
VALS = in_valve:(**Real**×DIAM) × out_valve:(**Real**×DIAM)
OFLOW = **Real** × Diam
DIAM = **Real**

Here we have "decorated" some type names with lowercase (selector) names, like field identifiers of record (structures). Chosen properly, these names can help us remember the relation between the formula and the actual phenomenon, the system identification problem. The pairs **Real**×Diam designate the height of the centre of the (assumed) round valve or pipe above bottom level, respectively its diameter. The variable-valued state can be modelled as:

type
STATE = liquid:LEVEL × input_valve:OC × output_valve:OC
LEVEL = **Real**
OC == open | closed

OC stands for the open/close state of a valve. open and closed are atomic tokens, i.e., identifiers. By being different they denote different "values".

Let the various numerals below designate reals (of unit centimeters), then:

lcs: (ctx,sta)
ctx: (vol,vvs,ofw)
vol: (100.0,100.0,300.0)
vvs: ((250.0,5.0),(10.0,5.0))
ofw: (255.0,5.0)
sta: (221.0,(open,closed))

i.e., (((100.0,100.0,300.0),((250.0,5.0),(10.0,5.0)),(255.0,5.0)),(221.0,(open,closed))) exemplifies a configuration.

We thus consider the fluid container system to be an atomic entity. The attributes of the fluid container system amount to container width, depth and height, to valve height positions and diameters, to the open or closed state of the input and output valves, and to the height of the fluid. ▪

List Compounds

The next example illustrates the list compound.

Example 5.12 *Train Journeys:* A train journey is an ordered list of two or more station visits. A station visit is a grouping of arrival time, station and platform names, and departure time.

We can formalise this:

Formal Presentation: Train Journeys

type
Time, Sn, Pn
Journey = Sta_Visit*
Sta_Visit = arrival:Time×sta_name:Sn×pla_name:Pn×dept:Time

Time, Sn and Pn are further unexplained sort names (i.e., abstract type names).

The list (and Cartesian) expression:

$$< (a_1, s_1, p_1, d_1), (a_2, s_2, p_2, d_2), (a_3, s_3, p_3, d_3), (a_4, s4, p_4, d_4) >$$

where suitably subscripted a,s,p and d stand for, respectively, arrival times, station names, platform numbers (or names), and arrival times, designate a journey starting at time d1, ending at time a4, from station s1 via station s2 and s3, in that order, and ending at station s4.

The mereology of the train journey concept transpires from the above use of the list and Cartesian type constructors (*, ×). We thus consider the train journey concept to be a composite entity whose atomic subentities are station visits. We consider only one attribute of a train journey, namely its number of station visits. The attributes of atomic station visits are arrival time, station name, platform number and departure time. ▪

Some list notation may be in order:

type
[1] A, fL = A*, iL = A$^\omega$
value
[2] a,b,c:A, ℓ:fL
[3] ⟨a,b,c⟩:fL
[4] ⟨a⟩$^\frown\ell$, $\ell^\frown$⟨a⟩, **hd** ℓ, **tl** ℓ, **elems** ℓ, **len** ℓ, **inds** ℓ

expresses [1] the sort A and the definition of the concrete types of finite, respectively possibly infinite lists over A elements; [2] that a, b, and c are arbitrary A elements, and that ℓ is an arbitrary fL element; [3] the enumeration of a simple list; and [4] the operations: the concatenation to the front, respectively to the end of a list, the head of a nonempty list (its first element), the tail of a nonempty list (the list of all its remaining elements, if any), the set of all distinct list elements, the length of a list and the (possibly empty) set of integer indices into a list. If nonempty, then the index set integers goes from 1 to n, where n is the length of the list.

Map Compounds

We give an example illustrating map compounds.

Example 5.13 *Computer File Directories:* This example is a "classic". A directory consists of zero, one or more uniquely named files, and zero, one or more uniquely named directories. So directories map file names into files and directory names into directories. Files, file names and directory names are further unexplained entities.

We can formalise this:

Formal Presentation: Computer File Directories

type
File, Fn, Dn
DIR = (Fn $\overrightarrow{m}$ File) × (Dn $\overrightarrow{m}$ DIR)

Here A $\overrightarrow{m}$ B denotes a map from unique A's to not necessarily unique B's. A map is like a function: Given a map m (say in A $\overrightarrow{m}$ B) and an a in A, such that a is in the definition set of m, then m(a) (m applied to a) is some b.

Example directories are:

([],[])
([fn↦file],[])
([fn↦file],[dn↦([],[])])
([fn1↦file1,fn1↦file1],[dn1↦([],[]),dn2↦([fn↦file],[])])
([fn1↦file1,fn1↦file1],[dn1↦([],[]),dn2↦([fn↦file],[dn↦([],[])])])

Here fn, fn1 and fn2 are file names; file, file1 and file2 are files and dn, dn1 and dn2 are directory names. The general map expression [a1↦b1,a2↦b2,...,an↦bn], where all a1, a2, ... and an are distinct, denotes a map from ai to bi.

The mereology of the directory concept transpires from the above use of the map and Cartesian type constructors ($\overrightarrow{m}$, ×). The attributes of a directory is the number and name of files and the number and names of subdirectories.

■

Some map notation may be in order:

type
[1] A, B
[2] M = A $\overrightarrow{m}$ B
value
[3] a:A, b:B
[4] m ∪[a↦b] : M, m†[a↦b] : M, m\{a} : M

The above expresses: [1] abstract types A and B; [2] concrete type M as a set of maps from A into B; [3] arbitrary naming of an element, a, of A and an element, b, of B; [4] the joining of a new mapping [a↦b] to a map m; the replacement of some mapping, say [a↦b′] in m by the mapping [a↦b], where it is assumed that b is different from b′; and the removal of a mapping [a↦b] from a map m (for any b). The ∪ operator is commutative: m∪m′ = m′∪m. The operators can be "pronounced" as ∪: merge, †: override, and \: restrict.

5.3.11 A Type System

Thus we can summarise the type notation that we shall be asking the reader to use. There are *simple type names* (i.e., *type expressions*):

type expressions:
Bool, **Int**, **Nat**, **Real**, **Char**
Tn_1, Tn_2, ..., Tn_m

Here TN_i are user-chosen identifiers (that do not coincide with other identifiers). **Bool**, **Int**, **Nat**, **Real** and **Char** denote the classes of Boolean truth values, integers, natural numbers, reals and character (symbol)s.

There are *composite types names* (i.e., *type expressions*):

type expressions:	**meaning**
A-**set**, A-**infset**	finite, resp. possibly infinite, sets
A × B	Cartesians
A^*, A^ω	finite, resp. possibly infinite, lists
$A \xrightarrow[m]{} B$	finite maps (from A into B)
$A \to B$, $A \xrightarrow{\sim} B$	total, resp. partial, function spaces
A | B	union type of either A or B

Here A and B are any user-chosen identifiers. A-**infset** includes finite sets, i.e., A-**set**. A^ω includes finite lists, i.e., A^*.

And there are *type definitions:*

type definitions:	**annotation:**
A, B, C, ...	sorts
D = A-**set**, E = A-**infset**	D (E): (also in)finite sets
F = A × B × C	F: (triple) Cartesians
$G = A \xrightarrow[m]{} B$	G: maps
$H = A \to B$, $J = A \xrightarrow{\sim} B$	H (J): total (partial) functions
K == alpha | beta | ... | omega	discriminated union of tokens

5.3.12 Type Constraints

In Example 5.10 the informal text lines between the formula lines illustrate the idea of a *type constraint*. These type constraints can be considered part of the mereology of the entities modelled.

Characterisation. By a *type constraint* we shall loosely understand some text which delimits a prior type description to not contain all the values otherwise allowed just by that prior type description. ▪

Example 5.14 *Type Constraints:* We will illustrate some type constraints that should have been expressed in earlier examples:

Liquid container system: (Example 5.11.) The in(take) valve must be placed higher than the out(take) valve. The liquid height cannot be lower than the level of the out(take) valve minus the diameter of that valve. The liquid height cannot be higher than the level of the overflow pipe plus/minus the diameter of that pipe. The diameter of the out(take) valve should stand in the following relation to the diameter of the in(take) valve: ... (etc.!).

Train journeys: (Example 5.12.) Arrival times should be before departure times at any one station, and their difference should be a minimum of t_{lo} minutes and a maximum of t_{hi} minutes, possibly depending on which station these times relate to. Departure times at one station should be before arrival times at any next station, and their difference should be commensurate with the normal travel time between such stations. The sequence of station names must be commensurate with a route through the railway net.[1] No station name can occur more than once. (That is, no cycles.)

Computer directories: (Example 5.13.) One might envisage some strictly not necessary constraints on directories: At any level of a directory, file names (at that level) must be distinct from directory names (at that level). Given a directory $\mathcal{A} = (f, \delta)$, let a valid path p of directory names (of that directory $\mathcal{A}$) be a nonempty sequence (i.e., a list) of directory names such that the first directory name, d, of path p, say $p{=}\langle \mathrm{d} \rangle \hat{}\, \mathrm{p}'$, is indeed the name of a directory ($\mathcal{A}'$) of δ, and such that if the path p is of length two or more, the remaining path p' is a valid path of $\mathcal{A}'$. ▪

As the last example shows, it may sometimes be useful to express the constraints by a bit of set, Cartesian, list, map, and, as we shall soon see, function notation. Hence we presented some of that set, Cartesian, list and map notation above.

5.3.13 Summary: Principles, Techniques and Tools

Principles. The principle of analysing and modelling *entities* is as follows: First decide whether an entity, that is, whether values of a class, i.e., type, of entities are atomic or composite. Then, for atomic entities decide on which one or more attributes these atomic entities have. And, for composite entities, decide on which of the following two aspects these composite entities have: their mereology, i.e., their compositional attributes and their subentities. ▪

Recall that the attributes of a composite entity have two facets: there are the attributes which indicate how the composite entity is composed; and there are other, we could call them the auxiliary attributes. The compositional attributes are expressed by saying that the composite entity consists of a set or a Cartesian or a list or a set of uniquely identified, that is, a map of subentities. The auxiliary attributes of a composite entity are very much like the attributes of atomic entities: they characterise the values of the entities as such, less their possible subentities.

Techniques. *Entities* are modelled as follows: Atomic entities by stating an arbitrary aggregation of possibly constrained types (say sorts), and composite

[1] That is: Station names must name stations of the rail net, and the journey route must be a route of that net.

entities by stating two things: an arbitrary aggregation of possibly constrained types (the auxiliary attributes, e.g., sorts), and a specific set, Cartesian, list, map (or function) composition of types (the compositional attributes). ▪

Tools. Models of *entities* are, for example, expressed using the `RSL` abstract type (sort) or concrete type concept. ▪

5.4 Functions

Characterisation. By a *function* we shall loosely understand something, a mathematical quantity (that no one has ever seen), which when *applied* to something (else), called an *argument* of the function, *yields* something (yet else), called a *result* of the function for that argument. If the function is applied to something which is not a proper *argument* of the function, then the totally undefined result, called **chaos,** is yielded. ▪

The question, for us, when confronted, as we are, with phenomena of one kind or another, is to decide which phenomena we should model as functions, and which we should not.

Example 5.15 *Functions:* We give some rough sketches of examples:

(i) To deposit savings in a savings account can be viewed as a function: the *deposit* function is applied to two arguments, the deposit *amount* and the account *balance*. The function yields a new *balance*.

The above is an appropriate choice of argument and result types if we consider only the account balance (and the amount deposited). If, however, we consider the entire bank (and the amount deposited), then the *deposit* function is more appropriately applied to the following arguments, the *bank*, the *client name*, the *client account number* (hence the account *balance*) and the deposit *amount*, and yields a new *bank*.

If you wish to consider some response to the client, for example that the deposit transaction succeeded, or did not succeed, depending, for example, on the validity of client name and account number, then you may wish to add a further, the response, component to the result, for example, `transaction succeeded` or `transaction did not succeed`.

Another example:

(ii) Inquiring about and actually buying an airplane ticket is abstracted as a function. For example, this can be viewed as: The *purchase* function is applied to three arguments, *travel information* about from where to where, on which flight, etc., the airline *flight reservations register*, and the *price* (in terms of monies). This function yields a "paired" result: the *ticket*, and an updated *flight reservations register*.

Another example:

(iii) To unload a ship in harbour at a quay can be considered a function. For example, viewed as: The *unload* function is applied to two (composite) arguments, a *ship* (loaded with cargo destined for) the *quay* of a harbour. This function yields a "paired" result: the *ship* less its unloaded cargo, and the *quay* "plus" the unloaded cargo.

Yet another example:

(iv) To admit a patient at a hospital can be viewed as a function. For example, viewed as, the *admission* function is applied to a number of arguments, the *patient*, the *hospital* (not registering that patient), the receiving *medical doctor*, *nurse*, etc. This function yields a "composite" result: an updated *hospital* (now registering that patient).

A final example:

(v) To land an aircraft can be viewed as a function. For example, viewed as, the *touchdown* function is applied to two arguments, the *aircraft* (which is in a state of flying), and the *runway* (which is assumed free for landing, i.e., reserved for "that" aircraft). This function yields a "composite" result: the *aircraft* (which is now in a state of running along the runway), and the *runway* (which is occupied by "that" aircraft). ∎

An important task in describing, prescribing or specifying (domains, requirements, respectively software design) is that of identifying all relevant functions, of (i) naming them, their (ii) arguments and (iii) results, and of (iv) defining what they "compute". In the above example we have covered (i–iii) but not (iv).

5.4.1 Function Signatures

Characterisation. By a *function signature* we shall understand the following composite information: The name of the function, a sequence of names of the types of the arguments and a sequence of names of the types of the yielded results. ∎

A function signature is not a definition of the function. But it is a significant indication of what the function "appears to be about".

Example 5.16 *Function Signatures:* The signatures corresponding to the functions mentioned in Example 5.15, are:

(i) Function name: *deposit;* argument types: *amount* and *balance;* and result type: *balance.*

(ii) Function name: *ticket_purchase;* argument types: *travel information*, *reservation register* and *price;* and result types: *reservation register* and *ticket.*

(iii) Function name: *unload;* argument types: *ship* and *quay;* and result type: *ship*, and *quay.*

(iv) Function name: *admission;* argument types: *patient, hospital* and *medical staff;* and result type: *hospital.*

(v) Function name: *touchdown;* argument types: *aircraft* and *runway*, and result types: *aircraft* and *runway.*

Formal Presentation: Function Signatures

We formalise the above:

type
  Amount, Balance
  TravInfo, ReservReg, Price, Ticket
  Ship, Quay
  Patient, Hospital, MedicalStaff
  Aircraft, RunWay
value
  deposit: Amount×Balance → Balance
  ticket_purch: TravInfo×ReservReg×Price → ReservReg×Ticket
  unload: Ship×Quay → Quay×Ship
  admission: Patient×Hospital×MedicalStaff → Hospital
  touch_down: Aircraft×RunWay → Aircraft×RunWay

Observe that there are no "hidden" types. If we had programmed the above, say in some imperative programming language, then some of the types would be omitted since the corresponding argument values and/or result component values would be kept, respectively stored in global variables. ▪

Once a function is rough-sketch identified the developer can determine, at least tentatively, the function signatures. To do so does indeed often require detailed considerations. Delineating the function signatures focuses the developer's mind. Many issues surface during this preliminary rough-sketch step. Writing down, systematically, whether informally only, or also formally, tends to further focus the development: step-by-step "achievements" can be recorded!

5.4.2 Function Definition

We have not spent much time or space, above, but we have indeed mentioned the concept of function definition.

Characterisation. By a *function definition* we shall understand a description, a prescription, or a specification which defines the relationship between arguments and results of a function. ▪

Example 5.17 *Function Definition:* We exemplify function definitions for some of the functions of Examples 5.15 and 5.16.

(i) The deposit function, when applied to an amount (to be deposited) and a(n account) balance, yields a new balance (of that account) which is the sum of the original balance and the deposited amount.

(ii) The air ticket purchase function, when applied to some (relevant) travel information, an airline reservations register, and a(n assumed ticket) price, yields a new airline reservations register and a ticket such that the ticket satisfies the travel information and the ticket (i.e., the reservation that it designates) is properly reflected in the yielded, i.e., new airline reservations register.

More information must be given about the three entities: travel information, airline reservations register and ticket, in order to define what is meant by "satisfaction" and "proper reflection".

(iii) The unload ship function, when applied to a ship and a quay, yields (1) a(n updated) quay which, in addition to the cargo that was already on the quay before unloading, now also stores that cargo from the ship, which was *destined* for that quay, and (2) an updated ship that is less that cargo that has been unloaded, and only less that cargo!

Formal Presentation: The unload Function Definition

```
type
  Sn, Qn /∗ Ship and Quay Designators ∗/
  C /∗ Container ∗/
variable
  s_c:C-set
  q_c:C-set
value
  q:Qn
  is_destined: Qn × C → Bool
  unload: C-set × C-set → C-set × C-set
  unload(scs,qcs) as (scs′,qcs′)
    post
      scs \ scs′ = qcs′ \ qcs
      ∀ c:C • c ∈ scs \ scs′ ⇒ is_destined(qn,c) ∧
      ∼∃ c:C • c ∈ scs′ ∧ is_destined(qn,c)
```

The cargo removed from the ship is the cargo added to the quay. Only and all such cargo was removed from the ship that was destined for that quay. ▪

We observe that attempts to complete function definitions may be frustrated by lack of sufficient details about the types and attributes of arguments and results. Another way of formulating the last sentence above is: Defining func-

tions help us to "discover" the type and attributes of arguments and results, the possible need for auxiliary functions (viz.: 'satisfaction' and 'proper reflection' in Example 5.17 item (ii)). Often one must be prepared to revise function signatures when attempting to complete function definitions.

5.4.3 Algorithms

Care must be taken to distinguish between specifying function definitions, and defining algorithms, including coding programs that implement function definitions. Function definitions are here thought of as being abstract, as emphasising what is to be computed, in contrast to algorithms (and code) which emphasises how a computation might achieve what the function definitions specify.

Characterisation. By an *algorithm* we shall understand a step-by-step specification which can serve as prescription for computation by a mechanical device, i.e., a computer, such that that computer computes what a function definition otherwise has defined. ∎

Example 5.18 *Algorithm:* We finally indicate rough sketches for algorithms for computing the deposit function (item (i)), respectively the ship unload function (item (iii)), of Example 5.17.

(i) *A deposit algorithm:*

Let the amount to be deposited be held in a variable v_d, and let the initial contents of v_d be d.

Let the balance of the account into which a deposit is to take place be held in a variable v_b, and let the initial contents of v_b be b.

Now add the contents d of v_d to the contents b of v_b, yielding $d + b$ as the new contents of variable v_b.

Nothing is said about the final contents of v_d.

(iii) *An unload algorithm:*

Let the cargo area of ship s be represented by a variable s_c. The contents of s_c is assumed to be a set of containers $\{s_{c_1}, s_{c_2}, \ldots, s_{c_n}\}$.

Let the cargo area of the quay, q, at which the ship is docked be represented by a variable q_c. The contents of q_c is assumed to be a set of containers $\{q_{c_1}, q_{c_2}, \ldots, q_{c_m}\}$.

Let an auxiliary predicate function is_destined apply to a quay designator q and ship container s_{c_i}, and let it yield **true** if that container is destined for quay q, **false** otherwise.

Now, for every ship s container s_{c_j} of s_c for which $q(q, s_{c_j})$ holds, remove s_{c_j} from s_c and add s_{c_j} to q_c.

Formal Presentation: An unload Algorithm

type

```
    Sn, Qn /* Ship and Quay Designators */
    C /* Container */
  variable
    s_c:C-set
    q_c:C-set
  value
    q:Qn
    is_destined: Qn × C → Bool
    unload: C-set × C-set → C-set × C-set
    unload(s_c,q_c) ≡
      while ∃ c:C•c ∈ s_c ∧ is_destined(qn,c) do
        let c:C•c ∈ s_c ∧ is_destined(qn,c) in
        s_c := s_c \ {c} || q_c := q_c ∪ {c}
      end end
```

As long as there exists a container c in s_c destined for quay q in parallel, at the same time, remove ($\setminus$) c from s_c and join ($\cup$) c to q_c.

Our informal algorithm presentation could have been presented in the form of a pseudo-program:

```
  action:
    name: unload
  global variables:
    q: Quay_name,
    s_cs: Ship_containers,
    q_cs: Quay_containers.
  pseudo-program:
    while there exists containers in s_cs destined for quay q
      do
        let c be a container in s_cs destined for q:
        remove c from s_cs;
        add c to q_cs
      end
```

Pseudo-programs can be expressed in a rich variety of forms. ∎

We leave it to the reader to examine the three definitions, that of the unload function, that of the unload algorithm and that of the unload pseudo-program. We believe that the difference between those definitions shows aspects of the difference, in general, between function definitions and algorithm specifications.

5.5 Events and Behaviours

To properly explain the concepts of events and behaviours we first, very briefly, review the concepts of states and actions.

5.5.1 States, Actions, Events and Behaviours

We need to characterise a number of concepts. These concepts are concepts of domain, of requirements and of computing.

Characterisation. By a *state* we shall loosely understand a collection of one or more entities whose value may change. ▪

Characterisation. By an *action* we shall loosely understand something which changes a state. ▪

Characterisation. By an *event* we shall loosely understand the occurrence of something that may either trigger an action, or is triggered by an action, or alter the course of a behaviour, or a combination of these. ▪

Characterisation. By a *behaviour* we shall loosely understand a sequence of actions and events. ▪

Example 5.19 *States, Actions, Events and Behaviours:* We show some examples where the four concepts "intermingle". The below examples relate to subexamples (i, ii, iii) of Examples 5.15–5.18.

(i) *Clients and bank accounts:*

- The balance of some client's bank account forms a state.
- Carrying out the functions of depositing (and withdrawing) monies into (respectively from) the account amounts to actions.
- The decisions by a client to make deposits and withdrawals amount to events that trigger respective actions. Assuming that there is a lower credit limit, the situation where a withdrawal action results in a bank account balance that exceeds the credit limit amounts to an event. That event may or may not trigger an action, or may do so in a delayed fashion.
- The sequence of a specific series of deposit and withdrawal events and actions forms a behaviour. For any given client and bank account many such behaviours are possible.

(ii) *Airline ticket purchase:*

- The airline seat reservation register forms a state.
- Carrying out the functions of actually buying a (or cancelling an already bought) ticket amounts to actions.

- The decisions to buy (reserve without paying, or actually reserve and pay), respectively cancel, a ticket are events that trigger respective actions. Having reserved, without actually paying for a ticket, and then not paying for the ticket before a certain date, amounts to an event, which may, or may not trigger an action.
- The sequence of a specific series of one or more ticket purchases and zero, one or more ticket cancellation events and actions forms a behaviour. For any given airline reservation system and potential passengers many such interleave and/or concurrent behaviours are possible.

(iii) *Ship unloading and quay loadings:*

- The ship cargo and the quay cargo storage can be considered either as a combined state, or, respectively, as the states, of a ship and a quay.
- Carrying out the function of unloading a ship amounts to an action.
- The decision to start unloading amounts to an event that triggers the unload action. The phenomenon that there is no more cargo to unload amounts to an event, which may or may not trigger an action.
- Behaviours:
  - ⋆ The quay behaviour: Seen from the point of view of a quay, the sequence of a specific series of unload events and actions, with respect to possibly different ships, forms a behaviour. For any given quay many such behaviours are possible.
  - ⋆ The ship behaviour: Seen from the point of view of a ship, the sequence of a specific series of unload events and actions, with respect to possibly different quays, forms a behaviour. For any given ship many such behaviours are possible.
  - ⋆ The combined quay/ship behaviour: Any set of pairs of commensurate quay and ship behaviours forms a behaviour.

By a pair of commensurate behaviours we mean one in which corresponding pairs of unloads from ships and loads onto quays are matched. ▪

5.5.2 Synchronisation and Communication

From the above we observe two closely related phenomena: That behaviours may *communicate*, and that the communications may take place *synchronously*, or *asynchronously*.

Characterisation. By *communication* we loosely mean the exchange of entities between behaviours, from one to the other, or both ways. ▪

A communication may involve one "sender", i.e., output, and one or more "receivers", i.e., inputs.

Characterisation. By *synchronous communication* we loosely mean the simultaneous communication between behaviours of one entity from one sender to one or more receivers. ▪

Characterisation. By a *shared event* we mean the simultaneous occurrence of one output event in one sender behaviour with its one or more synchronously communicating input events in one or more receiver behaviours. ▪

Thus synchronous communication can be said to be via a zero-capacity buffer between the sending and the receiving behaviours. Whether the synchronous communication is between one sender and one receiver, or several receivers is not stated here — but any description (prescription, specification) must state that, as it must state how the identification of the sending and receiving behaviours is accomplished. The sender behaviour is thus expected to be "held up" from the moment it wishes to synchronously communicate till the moment all communications have been accomplished.

Characterisation. By *asynchronous communication* we shall loosely understand the possibly delayed, e.g., buffered, communication between behaviours of one entity from one sender to one or more receivers. ▪

Thus asynchronous communication can be said to be via a non-zero-capacity buffer between the sending and the receiving behaviours. Whether the buffer acts like a queue or like a heap we do not specify here — but any description (prescription, specification) must state that, as it must state how the identification of the sending and receiving behaviours is accomplished. The sender behaviour is thus expected to be able to proceed with its "own" actions once it has placed its asynchronous communication.

Characterisation. By *synchronisation* we shall loosely understand the explicitly expressed (i.e., controlled) simultaneous occurrence of an event in two or more behaviours. ▪

Thus we consider the output communication in (i.e., from) one sender behaviour to designate the same event as the simultaneous input communication(s) in one (or more) receiver behaviour(s).

We say that synchronisation reflects shared events.

Example 5.20 *Communications:* We continue our line of examples with subexamples (i, iii, v) of Examples 5.15–5.19.

(i) When the bank account holder, i.e., the client, hands over the monies to be deposited to the bank teller, then we say that an output/input event has occurred, that the monies are being communicated and that the communication is synchronous.

(iii) When a ship unloads a (piece of) cargo (i.e., a container) onto a quay, then we say that an output/input event has occurred, that the cargo is being communicated and that the communication is synchronous.

(v) When a landing aircraft touches the ground, then we say that an output/input event has occurred, that the concept "touchdown" is being communicated and that the communication is synchronous. ▪

5.5.3 Processes

Characterisation. By a *process* we shall loosely understand the same as a behaviour. ▪

We shall, at times, make a pragmatic distinction: behaviours are what we observe, in the domain, while processes are what computers facilitate. We may implement a few, many, or all observable actions and events of behaviours in terms of computer processes.

Example 5.21 *Processes:* We continue subexample (iii) of Examples 5.15–5.20.

We focus just on the ship unloading example: We now consider the unloading as that of the interaction between two behaviours, two processes: ship and quay. The ship recurrently examines all its cargo. For every cargo destined for a designated quay, that cargo is removed from the ship's cargo, and communicated to the quay. When it contains no more such cargo, the ship "goes on to do other business" — which is not specified. The quay recurrently accepts all communications of cargo, from whichever ship. And it keeps doing that "ad infinitum"!

Formal Presentation: Ship Unloading and Quay Loading Processes

```
type
   C, Qn
channel
   sq:C
value
   qn:Qn

   is_destined: Qn × C → Bool

   ship: C-set → out sq  Unit
   quay: C-set → in sq  Unit

   ship(cs) ≡
      if ∃ c:C • c ∈ cs ∧ is_destined(qn,c)
```

```
        then
            let c:C • c ∈ cs ∧ is_destined(qn,c) in
            sq ! c ;
            ship(cs\{c}) end
        else skip
    end

quay(cs) ≡ let c = sq ? in  quay(cs ∪ {c}) end
```

Please observe the close correspondence between the informal and the formal explication above. ∎

5.5.4 Traces

Characterisation. By a *trace* we shall loosely understand almost the same as a behaviour: a sequence of actions (usually action identifications) and (usually action identifications) events of a single process.

Since, say, the parallel composition of two or more behaviours also forms a behaviour, hence a process, we need to clarify the notion of trace for such composite behaviours. Either, in a trace, we focus on the events shared between two or more behaviours, or we also include the actions of each of these behaviours. ∎

Example 5.22 *Traces:* We continue subexample (iii) of Examples 5.15–5.21.

The event trace of a ship in harbour, more precisely, during unloading, is a finite sequence of zero, one or more unload events. The event trace of a harbour quay, with respect to ship unloadings, is an indefinite length sequence of zero, one or more unload events. ∎

5.5.5 Process Definition Languages

The formal part of Example 5.21 illustrated textual (RSL/CSP) definitions of processes [168,301,311]. RSL/CSP was introduced in Vol 1, Chap. 21, and was used extensively in Vol. 2.

There are also a number of two-dimensional, diagrammatic ways of "rendering" the progress and interaction of two or more processes. These were covered extensively in Vol. 2's Chaps. 12–14: Various forms of *Petri nets* (*condition event nets*, *place transition nets*, and *coloured Petri nets*) [196, 273, 293–295], *message* [182–184] and *live sequence charts* [73, 149, 203] (MSCs and LCSs) and *statecharts* [144, 145, 147, 148, 150]. Variations of these diagrammatic forms are currently embodied in UML [44, 193, 264, 303].

5.6 Choice on Modelling Phenomena and Concepts

On one hand, we have a universe of discourse conceived of in terms of phenomena and concepts. On the other hand, we have some principles, techniques and tools for modelling phenomena and concepts.

Sections 5.3–5.5 have covered the model concepts of entities, functions, events, and behaviours. Confronted with a description (prescription, specification) task there are a number of modelling questions. Some we covered in earlier volumes of this series of software engineering textbooks. Some of these will be covered in this section. Others will be covered in several of the remaining chapters of the present volume.

5.6.1 Qualitative Characteristics

In which order do we describe (prescribe, specify) phenomena and concepts? Do we model them according to whether they are modelled as entities, or functions, or events or behaviours? The answer to these questions is relatively simple: It depends on which phenomena and concepts that best "characterise" the universe of discourse being described (prescribed, specified)!

5.6.2 Quantitative Characteristics

Thus the describer (prescriber, specifier), i.e., the software engineer, is confronted with another question: What does it mean that a phenomenon or concept characterises a universe of discourse "better" than another phenomenon or concept? The answer is, basically, a matter of style and of taste. But we shall try formulate some guidelines.

To do so we introduce a notion of universe of discourse "intensity".

The background for our "intensity" notion is the qualitative characterisation of a phenomena or a concept as being (modelled as):

- an entity
- a function
- an event
- a behaviour

Correspondingly we postulate that one can quantitatively characterise a phenomenon or a concept as one or more of:

- *information-intensive*
- *function-intensive*
- *event-intensive*
- *process-intensive*

Usually it only makes sense to speak of intensity if a phenomenon or a concept is predominantly one of the above, or, at the very most, two. If it is three, or all, then we would claim that it has no intensity!

Information-Intensive Universe of Discourse

Characterisation. By a universe of discourse being said to be *information-intensive* we roughly mean that entities, their number and type play a central role in understanding that universe of discourse. ∎

Example 5.23 *Information-Intensive Phenomena:*
An insurance system, with its records of insurances, claims and decisions (wrt. claims), can be said to be an information-intensive system.
A cadastral and cartographic system, with its many maps on properties and geographical areas, can be said to be an information-intensive system.
The military intelligence agency of a country, with its many reports on supposed (military strengths and weaknesses, strategies and tactics, etc., of) enemies, can be said to be an information-intensive system. ∎

Information-intensive systems — when partly (or wholly) computerised — typically give rise to information management systems based on extensive data base systems, document-handling systems and data communication.

Function-Intensive Universe of Discourse

Characterisation. By a universe of discourse being said to be *function-intensive* we roughly mean that functions, their definition, their application and their composition, play a central role in understanding that universe of discourse. ∎

Example 5.24 *Function-Intensive Phenomena:* Most resource planning, scheduling and allocation, and rescheduling systems (say, for production and for transport) can be said to be function-intensive. ∎

Function-intensive systems — when partly (or wholly) computerised — typically give rise to operations research-based optimisation software.

Event-Intensive Universe of Discourse

Characterisation. By a universe of discourse being said to be *event-intensive* we roughly mean that events, that is, the synchronous or asynchronous exchange of entities between behaviours, play a central role in understanding that universe of discourse. ∎

Example 5.25 *Event-Intensive Phenomena:*
A railway train traffic system with its movement of trains, their arrival and departure from stations, their reaching intermediate positions along a

line, and the attendant rail-point interlocking and signal settings, can be said to be an event-intensive system.

Similarly for air traffic systems: The arrival and departure of aircraft, in and out of ground, approach and regional monitoring and control zones, can be said to be event-intensive systems.

Telephone exchange systems, with the handling of single- and multiple-party telephone calls, hang-ups, simple service-oriented inquiries, etc., can be said to be event-intensive systems. ▪

Event-intensive systems — when partly (or wholly) computerised — typically give rise to real-time, often embedded and safety-critical software systems.

Process Intensive Universe of Discourse

Characterisation. By a universe of discourse being said to be *process-intensive* we roughly mean that behaviours play a central role in understanding that universe of discourse. ▪

Example 5.26 *Behaviour-Intensive Phenomena:*

Freight logistics systems with their handling and actual flow of freight by senders and logistics firms, and from senders via logistics firms, transport hubs, and transport conveyors to receivers, can be said to be behaviour-intensive systems.

Healthcare systems, with their flow of people (patients and staff), material (medicine, etc.), information (patient medical records), and control, can be said to be behaviour-intensive systems.

Harbours, railway stations and airports, with their handling of ship, train and aircraft arrivals and departures, assignment to quays, platforms and gates, unloading and loading, and servicing of a multitude of kinds: baggage, catering, fuel, cleaning, etc., can be said to be behaviour-intensive systems. ▪

Process-intensive systems — when partly (or wholly) computerised — typically give rise to distributed systems.

Discussion. Since events and behaviours are closely related, we shall find that both event- and behaviour-intensive systems give rise to implementations in terms of processes. ▪

5.6.3 Principles, Techniques and Tools

We close this chapter by basically presenting the methodological consequences of what we have covered and how we covered it! We shall use the term describing, in this section, synonymously with the terms prescribing and specifying; likewise for the terms derivable from description.

Principles. When describing a universe of discourse focus on identifying and describing *phenomena* and *concepts*. ▪

Discussion. If what you think is a phenomenon, i.e., a physically observable "thing", but you find that it does not fit in "any way, shape or form" into the entity, function, event and behaviour "mold" set out in this chapter, then what you have identified may not be a proper phenomenon, and similarly for concepts. ▪

Techniques. Analyse the identified *phenomenon* or *concept* and decide whether to model as an entity, including as a set (i.e., type) of entities, or as a function, or as an event, or as a behaviour. Then use the elsewhere given techniques (and tools) for respectively modelling entities, functions, events and behaviours. ▪

Discussion. There are some dualities. An entity can possibly be modelled as a function, or an entity can possibly be modelled as a behaviour, etc.

We can hint at this as follows.

Entity as function: you may consider the phenomenon, p, for example, modelled as an integer, as an entity, i_p, or as the function, f_p. As an entity i_p denotes a value. As a function f_p is to be applied to an empty argument, $f_p()$, and then yields the value.

Entity as behaviour: you may consider the phenomenon, p, for example, modelled as an integer, as an entity, i_p, or as the behaviour, b_p. As an entity i_p denotes a value. As a behaviour b_p is a behaviour composed, in parallel, with that of the phenomenon, i.e., an inquiring behaviour, q_p, which wishes to know, to use, the value of the integer phenomenon. This is modelled as follows: There are two behaviours, the inquiring behaviour q_p, and the integer behaviour b_p. To ascertain the value of p behaviour q_p synchronises and communicates with b_p by requesting the value of the integer phenomenon and obtaining, in return, that value.

Thus, whether we model a phenomenon, that appears to be an entity, as an entity, or as a function, or as a behaviour, is not always that straightforward. If the phenomenon is otherwise simple, i.e., is not subject to the constraints listed for the next two alternatives, then model it as an entity. If instead the phenomenon is shared among many behavioural phenomena then model it as a behaviour. ▪

Example 5.27 *Shared Documents:* A (perhaps even information-intensive) phenomenon can be thought of as a large collection of uniquely identified documents. Users of these documents may wish to read, copy or edit some document(s). We consider these users as a set of many behavioural phenomena, hence we model the document collection as a behaviour. This document collection behaviour is centred around an internal entity: the document storage. User behaviours now communicate with the document collection behaviour, by

requesting copies of documents for reading, and hence no return, by requesting copies of documents for copying, and hence no return, or by requesting original documents for editing, and hence for subsequent return. Requests are communicated from a user behaviour to the document collection behaviour. Responses are communicated from the document behaviour to the requesting user behaviour. Document returns are communicated from a user behaviour to the document collection behaviour. ▪

Tools. The tools for *modelling entities* are those of sorts and concrete types, as well as axioms, i.e., constraints over these. The tools for modelling functions are those of function signatures and function definitions, whether by explicit (i.e., function body) definition, or by pre/postconditions, or by axioms. The tools for modelling behaviours are either textual or diagrammatic, or a combination of these. Typically the textual tool is, as in these volumes, the `RSL/CSP` specification language. The diagrammatic tools are either one or more of various forms of *Petri nets* (*condition event nets, place transition nets*, and *coloured Petri nets*), cf. Vol. 2, Chap. 12 [196, 273, 293–295], *message* [182–184] and *live sequence charts*, cf. Vol. 2, Chap. 13 [73, 149, 203] and *statecharts*, cf. Vol. 2, Chap. 14 [144, 145, 147, 148, 150]. ▪

5.7 Discussion

5.7.1 Entities, Functions, Events and Behaviours

We have postulated that the concepts of entities, functions, events and behaviours offer a suitable complement of choices when modelling phenomena and concepts.

How do we know that? From where do we know that?

Well, pragmatics, i.e., experience, has shown that they suffice, to a large extent. And the four notions each fit into nicely complementing theories: of data types, of recursive function theory and of process algebras. But, as shown in Vols. 1 and 2 of this series, and as also shown in several chapters of the present volume, there is much, much more to modelling phenomena and concepts than just shown in the present chapter.

5.7.2 Intensity and Problem Frames

The notion of intensity is very much a part of that of Michael Jackson's idea of *problem frame* [189, 191]. We consider the notion of intensity to form one dimension along which the broader notion of problem frames can be characterised. In the above intensity classes we did not, for example, distinguish between the types of entities, within an entity-intensive phenomenon or within any of the other intensity phenomena. Once we add that dimension to our categorisation, then we start moving closer to Michael Jackson's notion of problem frames. We shall take a deeper look at problem frames in Chap. 28.

5.8 Bibliographical Notes

We refer to Michael Jackson's delightful book [189]: *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices.*

5.9 Exercises

5.9.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

5.9.2 The Exercises

If you are studying this volume on the basis of informal treatment only, then informal answers should suffice. Otherwise you are expected to suggest formalisations in addition to narrative texts or annotations of formulas.Subsequent exercises, in remaining chapters of this volume, will repeat, somehow these exercises, but then in more elaborate forms. Do not, at this early stage, shy away from attempting to solve these exercises. Admittedly, solving these exercises at this early stage, and given the genericity of the question (*for your chosen topic*), to answer the exercises requires some creative imagination. Later chapters will bring in more material and will thus enable you to embark on more meaningful, more satisfying solutions.

Exercise 5.1 *Entities.* For the fixed topic, selected by you, identify some 10–12 entities. Describe their types, i.e., whether simple (atomic) or composite. Try list some attributes of atomic entities and some properties of composite entities.

Exercise 5.2 *Functions.* For the fixed topic, selected by you, identify some 5–7 functions. Describe their signature. Describe their functionality, that is, what results they yield when applied to arguments — where arguments and results are entities.

Exercise 5.3 *Events.* For the fixed topic, selected by you, identify some 4–6 events. Describe these loosely (say in terms of entities being expressed during the event, i.e., being communicated).

Exercise 5.4 *Behaviours.* For the fixed topic, selected by you, identify some 4–6 behaviours. Describe these loosely — in terms of entities, functions and events, that is, in terms also of interactions with other behaviours.

Exercise 5.5 For your suggested solutions — to Exercises 5.1–5.4 — which entities, functions, etc., designate physically manifest phenomena, and which designate concepts?

6

On Defining and on Definitions

- The **prerequisite** for studying this chapter is that you have some experience in defining types and functions, but also that you have, perhaps often, wondered whether you really understood the "art of defining"?
- The **aims** are to discuss more general forms of definition, say, such as occur in philosophical discourse, and to discuss the more specific forms of definition, namely such as occur in software engineering.
- The **objective** is to help you to reflect seriously on the definitions that you will be making, and hence to help ensure that you will become a literate software engineer.
- The **treatment** is systematic and discursive.

Opto magis sentire compunctionem quam scire eius definitionem
(I had rather feel compunction[1] than understand the definition thereof.)
Thomas à Kempis, 1380–1471 [222]

Definition: The setting of bounds.
The action of determining a question at issue.
The action of defining: Setting forth the essential nature of.
A precise statement of the essential nature of a thing.
A declaration of the signification of a word or phrase.
The action of making definite.
The condition of being made, or of being definite, in form or outline.
Determination of the boundary and limitations of.
Oxford English Dictionary [222]

In nature everything is distinct, yet nothing defined.
William Wordsworth, 1770–1850

[1] Compunction: anxiety arising from awareness of guilt; distress of mind over an anticipated action or result [238]. `http://www.m-w.com/cgi-bin/dictionary`

The Encyclopædia Brittanica Online [95] characterises definitions as follows: *In philosophy, the specification of the meaning of an expression relative to a language. Definitions may be classified as lexical, ostensive and stipulative.*

- *Lexical definition specifies the meaning of an expression by stating it in terms of other expressions whose meaning is assumed to be known (e.g., a ewe is a female sheep).*
- *Ostensive definition specifies the meaning of an expression by pointing to examples of things to which the expression applies (e.g., green is the color of grass, limes, lily pads, and emeralds).*
- *Stipulative definition assigns a new meaning to an expression (or a meaning to a new expression); the expression defined (definiendum) may either be a new expression that is being introduced into the language for the first time, or an expression that is already current.*

We shall sometimes deal with lexical definitions and sometimes with stipulative definitions, but mostly the former. We shall call ostensive definitions by the name designations.

Characterisation. A *proper definition* sets bounds. It "divides" a world of phenomena and concepts in two, both nonempty, nontrivial halves:[2] One whose phenomena or concepts satisfy the definition, the other whose phenomena or concepts do not satisfy the definition. ▪

Characterisation. A *useful proper definition* is formulated such as to make the boundary between "its two halves" a sharp and precise, unambiguous boundary. ▪

• • •

We have, in this volume, in Chap. 3, characterised what we mean by method, methodology, principle, technique, tool, artifact, and so on. In doing so we have attempted to define these terms. The problem is, however, that some forms of definition, because of what they define, can be truly said to precisely delineate the thing being defined, while other forms of definition attempts are not of that nature! That is, some concepts can be defined formally precisely, while other concepts elude such precision. Since the software engineer shall indeed be formulating many definitions, we shall in this chapter therefore briefly discuss the concept of definitions. The discussion is more one of philosophy than of science or engineering.

[2] One can divide the entire population of mankind into two halves: One whose members divide the world into two halves, the other which does not!

A *personal remark* may be allowed: I think you might enjoy this section. In Sect. 6.2 I present a set of attempts of delineations of the subject: *What is Art?* You may find it entertaining, informative, even amusing. But my objectives are more serious: I wish to instill in you a certain respect for "that which can be defined, and defined reasonably objectively,[3] and that which cannot". The borderline is very fuzzy, and very hazy. Yet we shall later ask such questions as: *What is a Railway System?*, *What is Logistics?*, *What is a Financial Services System?*, *What is Health Care?*, *What is 'The Market'?*, *What is a Project and What is Production?*, *What is a University?* etcetera. Since stakeholders of these man-made systems are humans, one will get all kinds of answers to the above kind of questions when eliciting domain knowledge from the stakeholders. Sentiments, feelings, aesthetics,[4] and political opinions are OK, but they cannot be put to effective use for the purposes of understanding those aspects of these infrastructure and other societal components with respect to their effective support by computing and communication. Hence we must be prepared to weigh what constitutes proper, productive, useful input to our definitional work, as software developers, against broader, human issues — many of which, luckily, are not computable!

6.1 A Pragmatics of Definitions

First we must settle the *pragmatics* issue of why we define and the *semantics* issue of what we in general understand by a definition. Then we can discuss the metaissue of whether a mathematically precise definition can be made, and the *syntax* issue of the form of the definition itself.

6.1.1 Phenomena, Artifacts and Concepts

Definitions are about "things". What these things are is of importance when examining or analysing the concept of definition. We therefore exemplify three such things.

Characterisation. By a *manifest* (natural) *phenomenon* we understand a thing we can point to, something that exists in a physical world, like a river, a lake or a mountain. ∎

[3] By *objectively* is here, somewhat narrowly, meant: objectively with respect to the issue of whether it is of relevance to any form of mechanised treatment, i.e., computing.

[4] *Aesthetics:* (i) a branch of philosophy dealing with the nature of beauty, art, and taste and with the creation and appreciation of beauty; (ii) a particular theory or conception of beauty or art, a particular taste for or approach to what is pleasing to the senses and especially sight (modernist aesthetics); (iii) a pleasing appearance or effect. From *Merriam–Webster's Collegiate Dictionary [238].*

Characterisation. By an *artifact* we understand a thing created by humans, either manifest or intellectual. ∎

Examples of manifest artifacts are: a chair, a house, an automobile, a painting or a book. Examples of intellectual artifacts are: a painting considered as a work of art or a computer program.

Characterisation. By an *intellectual concept* we understand an abstraction, something usually not manifest, an idea, a notion, a thought construct. ∎

Examples of intellectual concepts are: A definition (yes!), a method, a methodology, a principle or a technique.

As the examples show, the border lines between the three seemingly distinct notions are not sharp, or rather: The pragmatics of what we wish to consider determines the classification. Some paintings, besides being paintings, are also works of art, i.e., besides many properties, also possess intellectual attributes, which we claim are absent in nonartistic paintings. About the three classes outlined above one could remark the following: physicists, biologists, etc., study the manifest world around us; computing scientists, engineers, designers, craftsmen, etc., study and deal with the world of manifest artifacts. Art historians, literary scholars, philosophers, etc., study the world of intellectual artifacts.

6.1.2 What Are Definitions?

The previous section was full of definitions! Definitions are forms of explaining a term. As such the term is a shorthand for the explanation. Some phenomena must be taken for granted, and thus need no definitions. Usually such phenomena are simple, that is, atomic. And usually the targets of definitions, those things we may wish to define, are composite, and are sometimes considered complex — with the pragmatic aim of a definition being that of simplifying one's understanding of complexity: to make this understanding less complex.

One does not have to define what a chair is, nor what a table is. One can point to them. From the point of view of pragmatics, a chair is an instrument for humans to sit on, or in. One can, however, try to define what a chair is: Such a definition would, it seems, also have to define such concepts as person, standing, sitting, resting, etc. And such definitions must not exclude anything that may serve as a chair although it was perhaps not designed, as an artifact, to serve as such (for example, a stone, the branch of a tree, the trunk of a felled tree, and so on).

6.1.3 The Nature of Concepts Being Defined

Above we have touched upon three classes of "things" that can be defined. And we saw that these three classes have some fundamental differences. The

things defined relative to some things manifested in a physical world can have their definition "validated" by somehow "comparing" the definition — and especially any theory derived from it — with example manifest phenomena in that physical world. The manifest artifact being defined can be verified in a different way than the above-mentioned validation comparison: A physical house can be confirmed to adhere, or not adhere, as the case may be, to some architect's plans for that house. Whereas an intellectual concept, a philosophical idea, a mathematical principle or a computing science concept can never itself be proven valid. So it seems! We shall examine some of the above in more detail below.

6.1.4 Mathematical Definitions

In Vol. 1, Chaps. 2–9 on mathematics, in particular logic, we saw several mathematical definitions. They were all of the nature: *Take it, or leave it.* That is, it was not a matter of whether you agreed or not with the definition; you had to play the game, accept it, or go elsewhere, and play another game.

They also had something else in common: *they began from first principles.* That is, *everything was explained down to a stone; not a concept was left undefined.* Properties, when not defined, were assumed. That is, *assumptions were clearly, indisputably stated.*

Notable examples of mathematical definitions were those of functions (Vol. 1, Chap. 6): function maps, types and attributes; the λ-calculus (Vol. 1, Chap. 7): free and bound names, substitution, alpha renaming and beta reduction; algebras (Vol. 1, Chap. 8): various forms of algebras, the morphism concept, special kinds of morphism; and logic (Vol. 1, Chap. 9), languages of Boolean algebras, propositions and predicates: their syntactic forms and their semantic evaluations. The pragmatics of the referenced definitions was, in all instances, to define precise mathematical concepts. As mathematical definitions they have to be accepted — or ignored! Their validity is solely dependent on whether the mathematical theory into which they enter is considered interesting, or not! We (re)invite the reader to review those parts of that volume in the light of the present discussion of the concept of definitions.

6.1.5 Physical World Definitions

By the physical world we understand the ensemble of such physical phenomena as are hinted at by Kepler's and Newton's laws (i.e., dynamics), hydrodynamics, thermodynamics, electricity, nuclear (atom) physics (wave mechanics, etc.), chemistry, biology, etc. That world, in some sense, exists. It is there and can be observed. When making mathematical models of segments of the physical world, the physicists formulate definitions of abstract concepts.

Example concept definitions are *velocity:* physical distance, say in kilometers, covered per time unit, say per hour; and *acceleration:* increase, per time unit, in velocity. These definitions designate things that are not immediately

observable, but can be calculated from observable phenomena: *distance, time,* and so on. The abstract physical concepts enter into physical laws, such as *voltage (across a two-pole electric circuit) equals current (through that circuit) multiplied by resistance (across the circuit).* The validity (or invalidity) of the concept definitions can thus be established by verifying the laws into which the concepts enter, respectively by refuting those laws!

• • •

Philosophers or just reasonably knowledgeable students of philosophy, may, when reading the above subsections, have shuddered! They may wonder: *Does he really know what he is talking about? He is skipping very important issues of "being", of "properties" [237], of "sense and meaning" [74], etc.* These are indeed core issues.

So, we are concerned. But we can refer interested readers to the vast literature on this subject. A rather personal selection, in addition to the ones just given above, is: Favrholdt's *Philosophical Codex — On Motivating Human Reasoning ("erkenntniss")* [100] (in Danish); Martin Heidegger's *Being and Time* [160]; and Sir Karl Popper's monumental works: *The Logic of Scientific Discovery; Conjectures and Refutations: The Growth of Scientific Knowledge*; and the essays *The Myth of the Framework. In Defence of Science and Rationality* [278, 279, 283].

6.1.6 Formal Definitions

Mathematical definitions are usually precise but usually not formal. A formal definition is expressed in a language with a formal, i.e., precise, unambiguous syntax, where the sentences designated by the syntax have a precise, i.e., formal (including mathematical) semantics, and where the definitions, as axioms, can enter into proofs of properties of the above-mentioned sentences.

One can formally define (many) mathematical concepts — that was how mathematical logic arose — and thereby "proceduralise" (check or even generate) certain previously informal mathematical proofs. One can formally define physical concepts and build software systems that can compute physical quantities solely on the basis of law definitions. And one can formally define many man-made universes: *What is a railway system?, What is a financial services sector?, What is a logistics system?, What is a healthcare sector?, What is an airport?,* etc. It is for this purpose that this volume is written.

6.2 Varieties of Philosophy Definitions

In this section we shall very briefly touch upon some definitional concerns in philosophical discourses. The main issue is that definitions in treatises on

philosophy deal with highly speculative matters. Just like we have numbers and among these transcendental numbers, so, it seems, we have definitions, and among these are those that concern philosophical matters.

6.2.1 Six Varietal Characterisations of Art

Take, as an example: *What is Art?*, and *What is a work of Art?*, a subject of philosophical aesthetics. The philosophical literature abounds with proposed definitions. Their pragmatic backgrounds differ widely. Some stem, it seems, from a sociopolitical background, others from a sociopsychological background, and so on. We shall just briefly mention — without discussion — a few examples. The examples (Examples 6.1–6.6) are all taken from David Favrholdt's Danish-language book *Aesthetics and Philosophy* [101].

Example 6.1 *Subjective Relativistic Definition of Art: Art is a matter of taste and pleasure* is basically what subjective relativism is about. *Everything, so it is claimed, is art!* ∎

Example 6.2 *Essentialistic Definition of Art:* That which is the essence of art, and which is not a property of other phenomena, is here attempted to define art. A claim for such an essence is the *personal experience of a peculiar emotion, and the arrangement and combinations of significant forms (lines, colours, and so on).* (Clive Bell, 1995 [24]) ∎

Example 6.3 *Conjunctive Definition of Art:* Artifacts are works of art if they satisfy a number of criteria: (a) *the release, the satisfaction of some metaphysical craving, usually accomplished through* (b) *a certain combination of form and materials, in other words* (c) *a unique thing, an original.* (d) *An autonomous artifact: the artistic value is a function only of the work itself.* (e) *A distillation of the subjectivity of its creator.* (f) *A subjective conditioned reconstruction in the mind of the spectator, who shares with the creator and other spectators* (g) *a common human (social) space.* (Jakob Wamberg, 1999 [371]) ∎

Example 6.4 *Disjunctive Definition of Art: Art is a conscious human activity of either reproducing things, or a construction of forms, or an expression of experiences such that it is capable of evoking delight, or emotion, or shock. A work of art accordingly follows the above.* (Władysław Tatarkiewicz, 1971 [352]) ∎

Example 6.5 *"Family Likeness" Definition of Art:* Wittgenstein claims that such ideas as art cannot be defined explicitly. But one can learn to use the concept (here art, or work of art) correctly. The meaning of a word, as work of

art, is not some entity that the word refers to, but its use, its application: *The meaning is the use.* Some concepts can only be characterised by characterising relations between each concept in a cluster of (thus) interdependent concepts. The concept terms (i.e., words) have to be used according to special, concept-dependent rules in order to describe these concepts and our understanding of them unambiguously. (Ludwig Josef Johan Wittgenstein [375]) ∎

Example 6.6 *The Parameter Theory Definition of Art:* The parameter theory establishes ten criteria according to which anything that purports to be a work of art can be evaluated.

(1) *Integration:* The interplay between details, at various levels, to form wholes or unities, at various levels.

(2) *Multiplicity and complexity:* The multitude of elements, again forming levels of such.

(3) *Technique:* Technical skills.

(4) *Aesthetic beauty* — versus ugliness.

(5) *Personality:* How and in which ways the personality of the supposed artist flavours the appraised artifact.

(6) *Repeatability:* The property that the artifact reveals the other qualities repeatedly, including from new sides, evoking new emotions, again and again.

(7) *Intellectual appeal:* Does the work of art purport to portray an idea (political, social, psychological, etc.), and, if so, how well.

(8) *Emotional appeal:* Evocation of sentiments, good or bad, as intended.

(9) *Other suggestive qualities:* In lieu of aesthetic beauty, other qualities could be "counted": being ugly, sinister, scary, etc.

(10) *The indescribable, inexplicable:* That, in the experience of a work of art which is not describable, which, so to speak, is metaphysical, or "out of this world".

For each of the ten criteria one can then ascribe, for a given artifact claimed to be a work of art, a grade, in a uniform scale, say 1–10, as to how well they satisfy the specific criterion. Artifacts above 60 could then, by agreement, be considered works of art. (David Favrholdt, 2000 [101]) ∎

6.2.2 Discussion

There are followers and critics for each of these schools of thought. Such a situation shows the vibrancy of philosophy and that it is indeed a nonformalisable endeavour.

We have gone into some detail, reporting on some thoughts about what the concept work of art is, in order also to illustrate the following point: Philosophers abstain from characterising such things as pleasure, emotion, delight — feelings of any kind. They take them for basic primitives, basically understood by all. When we, in developing, for example, computer-human

interfaces, express such requirements as *user-friendly, pleasing,* etc., then as software engineers, we should refuse to accept such superficially stated (and, as we shall call them, interface) requirements. As we shall see, in Chap. 19, we shall try establish "hard criteria" for these "soft wishes"!

6.2.3 A Possible Objection

Some "clever" reader may object: *But this is a set-up: The above six examples, i.e., the varietal characterisations, are really not definitions, but discussions.* Well, if you believe so, then what? Do they still not exemplify the difficulty of defining? We obviously believe so.

6.3 Preliminary Discussion

We have discussed, in Sects. 6.1 and 6.2, various categories of definitions: mathematical, physical world, formal and philosophical. The purpose of diverting, as it may seem, our attention away from software engineering, and conducting this discourse into metalinguistic and philosophical issues has been to show the reader that the issue of definition is not as straightforward as we may be used to in plain mathematics or in computer science. The purpose has thereby been to prepare the reader for the fact that we, as software developers (in describing domains, prescribing requirements and specifying software designs), will need to avail ourselves of a broad spectrum of definitional forms. When we define concepts arising out of a domain being described, then we must be prepared, as is the philosopher, for varietal approaches: one form of definition that may suit one domain stakeholder may not suit other domain stakeholders, for whom other forms may be more appropriate. But eventually we wish to formalise our definitions. When we define concepts arising out of prescribing requirements related to the domain — and since we are now faced with specifying "things" that need be computed — we must, on one hand, satisfy the possible variety of domain definitions, and, on the other hand, satisfy some computability criterion. That is, our definitions must eventually be formalised. And the varietal stakeholder views must be reconciled into a consistent and relatively complete whole. When we define concepts arising out of describing software designs we can start with formal definitions and render these in an informal form that is readable by professional software engineers for ease of acquisition. When formulating a definition we must constantly be concerned with how to best express the pragmatics and semantics of the issue being characterised.

6.4 A Syntax of Formal Definitions

Mathematical as well as formal definitions share in the following: A *definition* has basically two parts: A *definiendum (something to be defined),* i.e.,

the name of that which is to be defined, and a *definiens, an expression that defines*, i.e., *the defining expression.* Observe how the sentence *A definition has basically two parts*, is a form of *defining name.* On the other hand, the sentence *A defining name for that which is to be defined, and a defining expression, i.e., that which defines it!* is the *defining expression.* That is, we have syntactically defined definitions!

For complex concepts the defining expression part contains one or more other concepts which may also need definition. For simple, basic or, as we shall also call them, primitive concepts, the defining expression part contains just those terms that are understood by all! We shall later examine formal definitions more closely. We will then be interested in such things as: *circularity, do they actually define something constructively,* and so on.

Example 6.7 *Definition of Trees:* To define the concept of trees we first assume some further unexplained concepts:

Basis clause: A *root* is a further unexplained atomic quantity.

Inductive clauses: A *tree* consists of a *root* and a set of zero, one, or more *subtrees.* A *subtree* is a *tree.*

Extremal clause: A *thing* is only a *tree* if it follows from a finite number of applications of the basis and inductive clauses. ∎

By this definition a *tree* may either consist of just a *root;* or be of finite width in that all *subtrees* within a *tree* have a finite number of *subtrees;* or be of infinite width in that some *subtrees* within a *tree* have an infinite number of *subtrees.* But no *tree* is of infinite depth since we are allowed to use the above *tree* construction rules only a finite number of times. Observe that the trees defined above did not have tree and subtree roots labelled. The trees were unlabelled.

Example 6.8 *Definition of Graphs:* To define a concept of graphs we first assume some further unexplained concepts.

Basis clauses: A *node* — besides what is implied below — is a further unexplained atomic quantity. An *edge* — besides what is implied below — is a further unexplained atomic quantity.

Inductive clauses: A *graph* consists of a number of distinct nodes and a number of distinct edges such that every edge connects[5] two not necessarily distinct nodes.

Extremal clause: A *thing* is only a *graph* if it follows from (one single use of) the basis and inductive clauses. ∎

By the above definition *graphs* may either be empty, that is, consisting of no nodes and hence no edges, or be of finite size in that they have a finite number of *nodes* and a finite number of *edges,* or be of infinite size in that they have infinite numbers of either *nodes,* or *edges* or both!

[5] We assume that the ontological concept of being connected is known.

6.4.1 Recognition and Reproduction

It was said earlier that a definition serves to divide the world of things into two classes: *Those things that belong to the first class satisfy the definition; the others do not!* Presently the things we are defining can be considered syntactic (so far, static) structures. To decide whether a structure satisfies a definition we thus have to apply the definition to the structure and, somehow, *recognise* that the one fits the other! How such recognition can be performed given a definition and a structure (i.e., a thing) will not be dealt with here. Once we have recognised a structure as satisfying a definition, we may then wish, from only the recognition process (in a sense not from the structure itself) to reconstruct that structure! An example is needed.

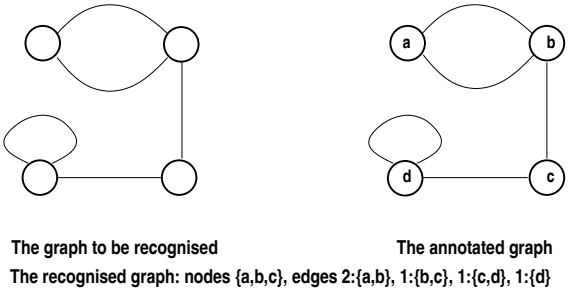


Fig. 6.1. A graph, its annotation, and its recognition

Example 6.9 *Recognising and Reconstructing Graphs:* Let us repeat, briefly, a definition of graphs: There is a finite set, N, of nodes and a finite set, E, of edges. Each of these edges, i.e., $e : E$, connects to exactly two not necessarily distinct nodes, i.e., $n, n' : N$. The left side of Fig. 6.1 shows a graph according to this definition. Like the definition, it displays nodes and edges, and the edges connect nodes of the graph. Now, what would constitute a recognition of the leftmost graph of Fig. 6.1? One suggestion could be: a listing of the nodes, and a listing of the edges, with the latter list somehow expressing *to which nodes each edge connects.* But how do we do this when the graph, like the leftmost graph on Fig. 6.1, has no labels adorning nodes or edges? How do we identify which is what? One answer would be to refer informally to the positions of the nodes, for example, by their north/south/east/west (N/S/E/W) orientation (or X, Y coordinate position on the paper). This is easy for the shown example, but becomes messy for graphs with many nodes. But the N/S/E/W (or coordinate position) idea tells us what to do: namely to annotate the graph. That is, to temporarily ascribe unique names to nodes, here a, b, c, d, and to list each edge by the set of one or two nodes it connects

together with a natural number which designates the number of so connecting edges. This recognition is shown to the right in Fig. 6.1. From such a recognition it should now be easy to see that it is a straightforward matter to reconstruct the graph and then delete the node labels so as to get a picture of a graph that is isomorphic to the original graph. ▪

6.4.2 Uniqueness and Identification

Figure 6.2 shows four graphs: one without labels, one with labels on nodes only, one with labels on edges only and one with both labels on nodes and on edges.

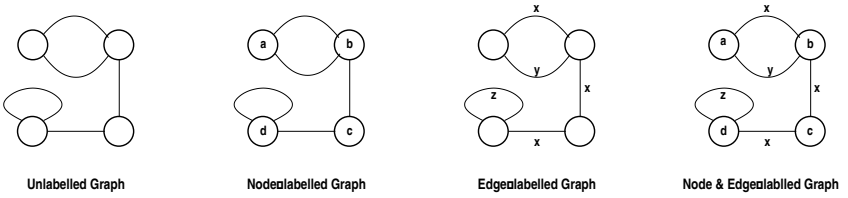


Fig. 6.2. Four graphs

It is indeed possible to properly define such graphs. Examples and exercises in Vol. 1, Chap. 16, Sects. 16.4.1, 16.4.5–.6 and 16.8 illustrated that. But how does one, for example for unlabelled graphs, secure that a given graph, like the leftmost of Fig. 6.2, when *recognised* can be *reproduced*? Well, as we discovered in the previous section, we need some form of identification in order to uniquely designate how many and where. Whether that identification (i.e., labelling) is part of the graph as defined, or whether any so-defined (i.e., unlabelled) graph can be annotated so as to create the appearance of uniqueness is a matter of choice. Observe from Fig. 6.2 that the rightmost graph does not enjoy a unique labelling of edges; yet, the graph is recognisable and uniquely reconstructible (to within isomorphism). There is a principle and a possible technique buried here:

Principles. *Definition/Unique Recognitions:* Whenever a definition is presented make sure that the things defined can be uniquely recognised to within isomorphism. ▪

Techniques. One way to adhere to the *principle of definition/unique recognition* is to label, as part of the definition, substructures of the thing being defined in such a way as to ensure unique recognition. ▪

There is no unique way to do the labelling. But all such labelling is isomorphic under renaming.

6.4.3 Ontological Terms

Footnote 5 reveals that we are always, when defining, i.e., when specifying, dealing with at least three kinds of terms: those which we are defining, those which are used in the defining expression and which we all take for granted and those which we are also using in the definiens, but which are technical in the sense that if they can be misunderstood, then there are bound to be inconsistencies.

The term connect is such a term. It is metalinguistic[6] with respect to the term being defined, viz.: graph. We shall say that such terms are ontological[7] with respect to the universe of discourse being dealt with (here the mereological part-whole relationships of graphs).

6.5 A Semantics of Formal Definitions

There are two issues at stake when asking the question: *if a definition, as a piece of text, is considered a syntactic quantity, then what is its semantics?* The first issue we could call the semantic, or just the formalistic issue of the semantics of a definition. The second issue we shall call the pragmatic issue of the semantics of a definition.

In the formalistic issue we simply ask: what is the meaning of the definition, and does it really designate anything? Is the definition vacuous, or does it define the universe of all things? In the examples given earlier we saw that there was at least one image put on paper that adhered to the definition; and it is not hard to imagine other images that did not adhere to the definition: Assume a proper graph. Then delete a node onto which some edges are incident. Is the result a graph? No!

The basic pragmatic idea of *the semantics of a definition* is that there are phenomena, in the real world, or concepts, say in mathematics, that — abstractly viewed — satisfy the definition, while there are other such phenomena, that — similarly viewed — do not satisfy the definition. To the basic pragmatic idea is also added that it is clear, for all cases, whether satisfaction holds or not.

For the kind of definitions that we shall be learning about and constructing the matter is as follows: If the definitions purport to relate to the real — the actual world, the *domain* — that is, if the defined terms name concepts of that world, then the consequences drawn from the definitions must similarly, realistically, relate to that world. If the definitions are of *requirements* to software, then they must designate *computable objects*. And if the definitions are of *software designs*, then they too must designate *computable objects*.

[6] A term is *metalinguistic* if it is part of the language used to describe another language.

[7] *Ontological:* Relating to or based upon being or existence. An ontology is a specification of a conceptualization.

The notion of realistically relate to that world is not an easy one to fully satisfactorily explain. To do so, in the end, brings us into the philosophical universes of *epistemology*,[8] and *ontology*, amongst others.

The domain descriptions, requirements prescriptions and software design specifications that we shall be pursuing all have the following in common: we will first express these definitions informally, in clear, natural language, typically in some professional (domain-specific and/or software engineering) language. And then we will balance that definition by a formal definition, expressed in some formal language, usually `RSL`. Thus the meaning of definitions will usually be the meaning of `RSL` (class) expressions: the set of mathematical models that satisfy the `RSL` expressions.

6.6 Discussion

6.6.1 General

Real artifacts, 'God-given' or 'man-made', can be pointed to, but sets of all the designated phenomena usually cannot be pointed to. Types are therefore typically defined. So are many other things close to reality. It may seem that definitions sometimes replace that reality. But definitions, and, in fact, any descriptions whatsoever, are just models of what they purport to describe.

In several later chapters we shall follow up on the examples and ideas of this section. That is: We shall formally model uniquely identifiable things, we shall formally model recognition, and we shall formally model how to reconstruct a recognised thing.

6.6.2 Principles, Techniques and Tools

In these texts and handbooks on software engineering we have chosen to label what might be considered definitions by the term characterisation. The reason is that they are not definitions of hard, say mathematical or physically measurable, things. They are, instead, characterisations of elements of discourse into principles and techniques of human development. They are, therefore, a "bit woolly around the edges", and therefore not precisely statable. When we, in contrast, in our software engineering work, deal with precise matters, then we define.

Tools. *Characterisation and Definition:* There are two kinds of tools. (i) Language: use either precise national, natural and professional language or augment such informal definitions by mathematical expressions. (i) Naming: when we are to delineate a human phenomenon we use the term characterisation, and when we are to delineate a precise phenomenon we use the term definition in designating our effort. ∎

[8] *Epistemology*: The study or a theory of the nature and grounds of knowledge especially with reference to its limits and validity [238].

The borderline between when to characterise and when to define is a soft one; it reflects elements of philosophical nature. As will be shown in the next chapter, one usually starts with a few basic facts: things that exist, physically manifest things that can be designated, pointed to. Sometimes other things (i.e., other phenomena), that are likewise manifest, can be characterised or defined in terms of the basic facts. Sometimes ideas, notions, concepts or things that are not manifest need be characterised or defined.

Principles. There are two sides to *characterisations and definitions:* (i) when we encounter a concept, something not physically manifest, then we must characterise or define it; and (ii) when we encounter a physically manifest thing, i.e., a phenomenon which can be explained in terms of other such designatable things, and/or in terms of other characterisations and definitions, then we likewise characterise or define it. ▪

Techniques. The basic techniques of *characterisations and definitions* are: (i) Give the characterisation or definition a name (the definiendum); (ii) formulate the definiens briefly and precisely, i.e., concisely; and (iii) make sure all terms in the definiens are well understood: that they are either designated or defined. ▪

6.7 Exercises

Exercises 6.1–6.3 are closed book exercises. For each of these three exercises keep repeating your solution until it, in your own considered opinion, comes close enough to ours.

Exercise 6.1 *Definition, Definiens and Definiendum.* Write down in your mother tongue a characterisation of what a definition consists of and name its parts and characterise those parts.

Exercise 6.2 *Basis Clause, Inductive Clause and Extremal Clause.* What role do the basis, the inductive and the extremal clauses serve in definitions?

Exercise 6.3 *What Is Art?* How many characterisations were given of the concept of art? Try reformulate the essence of each of these characterisations.

• • •

Exercises 6.4–6.11 mostly make sense when using this book in a formal course over this volume. The exercises all illustrate the issue of proper definitions, also formally. Anyway, we have hinted at some definitions in our formulations of the exercises. When using this book in an informal course over this volume try reformulate the definitions along the lines of: *A tree consists of a root and a finite set of zero, one or more subtrees. Subtrees are trees* — albeit inserting

root and branch labels as appropriate. We otherwise, for Exercises 6.4–6.11, refer to Sect. 6.4, especially its subsection, Sect. 6.4.2, on uniqueness and identification.

Exercise 6.4 *Finite Root-Labelled Trees.* Define a concept of trees, informally, as in Example 6.7, but for which all roots are labelled — and such that no two 'immediate', but otherwise 'distinct' subtrees of a tree have 'identically labelled' roots.

Suggest what we might mean by immediate, distinct and identical labels.

• • •

Figure 6.3 gives a snapshot of the kind of labelled trees referred to in Example 6.7 and in Exercises 6.4–6.6.

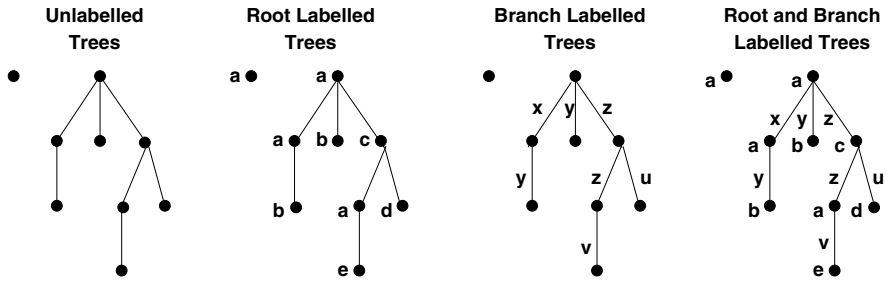


Fig. 6.3. Unlabelled and labelled trees

• • •

Exercise 6.5 *Finite Branch-Labelled Trees.* Define a concept of trees, informally, as in Example 6.7, but for which all branches (the things that "connect" a root of a tree with the roots of its immediate subtrees) are labelled — and such that no two branches of a tree are identically labelled.

Exercise 6.6 *Finite Root- and Branch-Labelled Trees.* Define a concept of trees, informally, as in Example 6.7, but for which all roots and all branches (the things that connect a root of a tree with the roots of its immediate subtrees) are labelled — and such that no two branches emanating from a root (to the root of a subtree) are identically labelled.

Do you need to maintain distinctness of root labels of the subtrees of a root and branch-labelled tree? Motivate your answer.

Exercise 6.7 *Distinctly Labelled Trees.* As for Exercise 6.6, only now it is required that no two root labels are the same, that no two branch labels are the same, and that root and branch labels also differ.

Exercise 6.8 *Forest of Trees.* Based on Exercises 6.4–6.7, define a concept of forest as consisting of trees, unlabelled, or labelled one way or another, but such that no two labels of any two somehow labelled trees are identical.

Does Figure 6.3 "portray" such a forest? Motivate the answer.

Exercise 6.9 *Finite Node-Labelled Graphs.* Define a concept of graphs, informally, as in Example 6.8, but for which all nodes are distinctly labelled.

Exercise 6.10 *Finite Edge-Labelled Graphs.* Define a concept of graphs, informally, as in Example 6.8, but for which all edges between any given pair of (in this exercise unlabelled) nodes are distinctly labelled.

Exercise 6.11 *Finite Node- and Edge-Labelled Graphs.* Define a concept of graphs, informally, as in Example 6.8, but for which all nodes are distinctly labelled, and for which all edges between any given pair of (in this exercise now labelled) nodes are distinctly labelled.

7

Jackson's Description Principles

- The **prerequisite** for studying this chapter is that you have read the two previous chapters.
- The **aims** are to introduce Jackson's concepts of designations, definitions and refutable assertions, and to provide formal tools for the expression of manifestations of these concepts.
- The **objective** is to ensure that the developer becomes a professional specifier.
- The **treatment** is systematic to formal.

We build on ideas eloquently expressed in [189] *Software Requirements & Specifications, a lexicon of practice, principles and prejudices,* by Michael Jackson.

Since all we do is construct, analyse and compare descriptions we shall analyse the concept and constituents of descriptions.[1] A description is about manifest individuals, i.e., phenomena and concepts. Some of these represent, or are intended to represent, facts; others represent mental constructions, i.e., concepts. A description includes *designations*, definitions and refutable descriptions. A description can either be formal or informal. A description sets a scope and a span, and a description expresses moods. By an individual we mean a physically manifest phenomenon. In other contexts we may refer to individuals instead by the term things.

7.1 Phenomena, Facts and Individuals

Phenomena are what appears to exist. Domain phenomena are built up from facts about individuals. A fact is a simple truth about the world: It is the smallest unit of observation. Large and complex observations and truths can

[1] We shall here use the term description as also covering the terms prescription and specification.

be broken down into assertions about several facts. A fact involves one or more *individuals*. Anything can be an individual. Each individual is identical to itself but distinct from all other individuals. If x is identical to y, then x and y are the same individual: $x = y$. If we say that x is similar or equivalent to y then x and y are distinct individuals that share some property, characteristic, attribute or quality. We may use the operator $\equiv$ or $\approx$ to denote this relationship. Although it could be a potential misuse of terminology, we may sometimes say that a phenomenon is identified with (i.e., is bound to) some identifier.

7.2 Designations

The first subsections of this section, although they may use the term "universe of discourse", are formulated to apply primarily to domains (to be described). However, they also apply, inter alia, with only minor textual adjustments, to such universes of discourse as requirements and software design. In this and the following two sections we shall implicitly be quoting from Michael Jackson [189].

Characterisation. By a *designation description* (for short: *designation*) we syntactically mean a textual triple: a *designation term*, a *designation recognition rule*, and a *designation identification*. ▪

Characterisation. A *designation term* is a simple, i.e., atomic name. ▪

Characterisation. A *designation recognition rule* is a text which purports to *designate* something. ▪

Characterisation. The *designation identification* links the *designation recognition rule* to that something (mentioned in the previous characterisation). ▪

To create a *designation* we thus write down two items: a *designation description* and a *designation identification*. The *designation description* is usually of the form:

- *Designated term:* `dt`.
- *Recognition rule:* `dt`'s satisfy the following informally stated properties: ... — where all other words, i.e., terms, are assumed to be well-known!

That is: *recognition rules* are expressed only in terms that are otherwise well understood, i.e., part of the folklore. If a recognition rule thus contains a term that is elsewhere *defined*, then it is not a recognition rule but becomes a *definition*.

Example 7.1 *Rail Units, I:* We give the first in a series of examples relating to rail nets.

- *Designated term:* `rail_unit`, or just `U`.
- *Recognition rule:* A `rail_unit` is a composition of an even number of parallel positioned rails (long, narrow, profiled iron bars) separated such that one can always identify pairs of rails of the composition that are at a specified distance (the rail gauge), and otherwise held together by a set of ties.

When we write `rail_unit` we mean the extensional meaning (the **type**) of that term. ▪

When there is doubt as to whether a value of a type or a type is being referred to we shall always mean type — except where value of a type is specifically mentioned.

Example 7.2 *Rail Units, II:* We give the second in a series of examples relating to rail nets.

- *Designated term:* `linear_rail_unit`, or just `U`, for which a predicate, to be assumed, holds: `is_linear_rail_unit(u)` for all `u` in the extension `U`, i.e., `u:U`.
- *Recognition rule:* A `linear_rail_unit` is a pair of parallel positioned rails (long, narrow, profiled iron bars) separated at a specified distance (the rail gauge) and held together by a set of ties. (The predicate `is_linear_rail_unit(u)` "arises" from the recognition rule.)

Observe that the designation term and the recognition rule dealt with a concrete concept for which the immediate concretisation points to a set of a phenomena. What is being designated and to be recognised is any one member of this set. ▪

The latter example is a specialisation of the former. The former example expressed: ... *composition of an even number of parallel positioned 'rails'* ... thus allowing for, say, two pairs as in a switch or in a simple crossover rail unit.

A *designation identification* is usually of the form: "That 'thing' (u) there is a U. So is that 'thing' (u') over there!" — thus physically pointing out a *designation set*, that is, instances of values that together become part of a type.

7.2.1 Some Observations

In general a *designation description* can be formalised; but one cannot formalise the *identification* — as it relates a formal world to an inherently informal world. As illustrated in the two previous examples, the latter of which

was a specialisation of the former, a *designation* may extensionally denote a class (of things) which can be subdivided into subclasses.

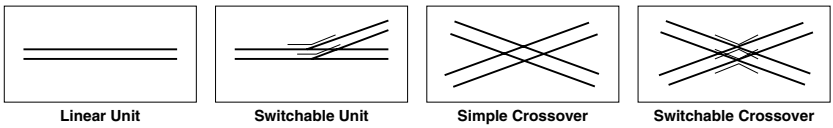


Fig. 7.1. Roughly drawn rail units

Example 7.3 *Rail Units, III:* We give the third in a series of examples relating to rail nets.

- *Designated term:* `rail_unit`.
- *Recognition rule:* — as for the above, previous `rail_unit` example.
- *Designated term:* `linear_rail_unit`.
- *Recognition rule:* — as for the above, previous `linear_rail_unit` example, and additionally: Thus a `linear_rail_unit` has two "ends" and a single (two-way) link between these ends. Any other `rail_unit` may be `connected` to either of these ends (and if so, then the end is called a `connector`).

To support textually expressed recognition rules it is often useful to deploy graphic means as in Fig. 7.1. Even photographic means could be used. And then one usually could show several photos of variations of the same kind of designated individuals.

- *Designated term:* `switchable_rail_unit`.
- *Recognition rule:* — as for `rail_unit` just above, and additionally: A `switchable_rail_unit` has three `connectors` which "define" (allow, permit) two two-way links through the `switchable_rail_unit`.

- *Designated term:* `simple_crossover_rail_unit`.
- *Recognition rule:* — as for `rail_unit` just above, and additionally: A `simple_crossover_rail_unit` has four `connectors` which "define" (allow, permit) two two-way links through the `simple_crossover_rail_unit`.

- *Designated term:* `switchable_crossover_rail_unit`.
- *Recognition rule:* — as for `rail_unit` just above, and additionally: A `switchable_crossover_rail_unit` has four `connectors` which, together with the switching ability of the unit, "define" (allow, permit) four two-way links through the `switchable_crossover_rail_unit`.

- *Designated term:* `connector`.

- *Recognition rule:* A `connector` is that which allows two `rail_units` to be connected, "end to end".

Further recognition rules deal with `connectors` and `rail_units`. *A `connector` is any "end" of a `rail_unit` to which other `rail_units` may be 'connected'. Any one `connector` is shared by at most two `rail_units`. A `rail_unit` is mutually exclusive either a `linear_rail_unit` or a `switchable_rail_unit` or a `simpl_crossover_rail_unit` or a `switchable_crossover_rail_unit`.* See axioms [2,3] in a formalisation of the above, in Sect. 7.2.2. ▪

As we shall see in Section 7.4, doubt may be raised as to whether some of the text parts of the above recognition rules express assertions. (Designations state facts, not assertions.) For example: "has two ends" (or three, or four). These parts are part of recognition rules, but what about: "any one connector is shared by at most two rail units"? Or the part right after it: "... mutually exclusive ..."? For the last (the mutual exclusion) we can claim it to be a fact, hence it is part of a recognition rule. For the "at most two", the case is a bit more complicated. And then, when we formalise the whole thing, as we shall see below, and when such a formalisation is compared to that of a refutable assertion, we shall see that, formally speaking, the difference is almost invisible. Thus we must be prepared for the eventuality that the pragmatics of making a distinction between designations, definitions and refutable assertions can not be carried visibly over into formal models of the things being designated or defined, or for which assertions are expressed.

7.2.2 Formalisation

Which of the above alternative ways of designations are we going to formalise? Typically we formalise a *designation description* as shown in the next example. It allows for the general case of several alternatively expressible designations.

Example 7.4 *Units:*

Formal Presentation: Formalisation of Rail Units, I

We choose to introduce just one type (i.e., sort), U, for rail units, and we choose to let (recognition rule-oriented) observer functions and suitable axioms help separate rail units into a partition of its more specialised rail units.

Let W denote a concept of undirected "ways" through a unit (Fig. 7.2).

type
U, C, W
value
obs_Cs: U→C-**set**

obs_Ws: U→W-**set**
is_linear,is_switch,is_simpl_cross,is_switch_cross: U→**Bool**
axiom
[1] ∀ u:U, ∃ c,c′,c″,c‴:C •
card{c,c′,c″,c‴}=4 ∧ obs_Cs(u)⊆{c,c′,c″,c‴} ⇒
is_linear(u)⇒obs_Cs(u)⊆{c,c′}∧**card** obs_Ws(u)=1 ∨
is_switch(u)⇒obs_Cs(u)={c,c′,c″}∧**card** obs_Ws(u)=2 ∨
is_simpl_cross(u)⇒obs_Cs(u)={c,c′,c″,c‴}∧**card** obs_Ws(u)=2 ∨
is_switch_cross(u)⇒obs_Cs(u)={c,c′,c″,c‴}∧**card** obs_Ws(u)=4,

[2] ∀ c:C • **card**{ u | u:U • c ∈ obs_Cs(u) } ≤ 2,

[3] **let** lus={|u:U•is_linear(u)|}, sus={|u:U•is_switch(u)|},
cus={|u:U•is_simpl_cross(u)|}, scus={|u:U•is_switch_cross(u)|}
in lus ∩ sus = lus ∩ scus = lus ∩ scus = {} ∧
sus ∩ cus = sus ∩ scus = cus ∩ scus = {} **end**

U denotes the possibly infinite set of all *designations* satisfying the rail_unit *recognition rule*. C stands for the possibly infinite set of all *designations* satisfying the connector *recognition rule*. W denotes the concept of way (or link). The predicates is_linear, is_switch, is_simpl_cross and is_switch_cross further constrain the rail_unit *recognition rule*, as indicated by the **axiom**.

Observe how the formalisation fits with the narration. ∎

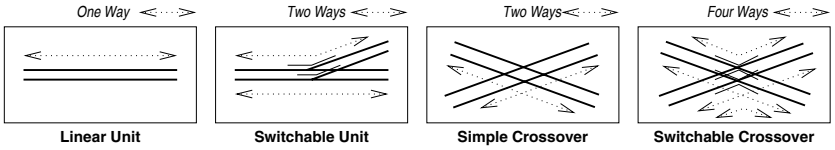


Fig. 7.2. Undirected ways through units

7.2.3 Observer Functions and Identification

The *observer functions*, eg. obs_Cs and obs_Ls, are the closest counterpart to the undefined terms of *recognition rules* that we have. These *observer functions* cannot be defined. They are postulated. They "arise" as the result of *identification*. Given an actual *universe of discourse* — to which the above *designations* are said to apply — one can now "define" these *observer functions* by "walking out and into" the *universe of discourse* and by providing the *identification*. To get a better grasp of possible relationships between recognition rules and observer functions let us give a further example.

Example 7.5 *Rail Net: Lines and Stations: A railway net consists of [one or more] lines and [two or more] stations. A line is a linear sequence of one or more linear rail units. A station is any composition (connection) of rail units. [A line connects exactly two distinct stations.]* The above (excluding the bracketed parts) may be claimed to be a definition, and we shall soon turn to a treatment of definitions. But we claim that each of the (unbracketed parts of the) above notions can be designated. The *italic text* above may not look like *recognition rules*, but they are!

One can indeed "walk out, into" the railway system domain and, with a sweeping hand, express: *That "thing" there is a linear rail unit. That one, next to it, is likewise. That particular sequence of those two linear units I just pointed to forms (part of) a line. This "thing" here is a rail unit of a station. It is a crossover. Here is the connector that separates a line from a station. No rail unit is both a rail unit of a line and of a station, or of two otherwise distinct lines or stations.*

Formal Presentation: Formalisation of Rail Units, II

```
type
  N, L, S, U, C, L
value
  obs_Ls: N → L-set, obs_Ss: N → S-set,
  obs_Us: (N|L|S) → U-set
axiom
  ∀ n:N, l,l′:L, s,s′:S •
    card obs_Ls(n) ≥ 1 ∧ card obs_Ss(n) ≥ 2 ∧
    {l,l′} ⊆ obs_Ls(n) ∧ {s,s′} ⊆ obs_Ss(n) ⇒
      l≠l′ ⇒ obs_Us(l) ∩ obs_Us(l′) = {} ∧
      s≠s′ ⇒ obs_Us(s) ∩ obs_Us(s′) = {} ∧
      obs_Us(l) ∩ obs_Us(s) = {} ∧
      exactly_two_distinct_stations(l,obs_Ss(n))
```

We leave out the definition of exactly_two_distinct_stations. ▪

7.2.4 Mathematical and Computing Entities

From a formal point of view, i.e., from the points of view of mathematics and computer and computing science, which are the kinds of designations? Which kinds of mathematical entities are they? Or, in the jargon of computing: Which types of computer and computing science entities are they?

Mathematical Entities

When viewed mathematically, are the designations scalars, like numbers (integers, reals (or complex), rationals, transcendental), or truth values, etc.? Or

are they composite, like sets, Cartesians, lists, maps or functions (or algebras, etc.)? And if the latter, then which are their component elements?

Computing Entities

When viewed computer and computing sciencewise, are the designations values (of, for example, the above-mentioned mathematical kind (where models are like algebras))? Or are they types of these, i.e., types as we know them in computer science: simple lattices, Scott domains [133, 316–324] or otherwise? Or are they events — whatever they are? If lists, then what do these lists model: behaviours (of processes, in terms of traces of actions and/or events, etc.), or other? In any case, when we model computing entities (values, types, events, (process) behaviours, semantic algebras), then we model them in terms of the above-mentioned mathematical entities.

Awareness

We do not intend to give a full answer to the question: which kind (type) of formal entities can designations be modelled by? We only advise that the practicing, professional software engineer be reasonably well-versed in these matters: modelling designations formally in terms of mathematical entities — perhaps couched in the computing jargon of, for example, types and values, events and (process) behaviours, semantic algebras or other.

Some Guidelines

In the following we shall try to stay clear, if at all scientifically possible, of the deeper problems of conceptual modelling, viz.: [112, 340], as currently studied in computer and computing science, and in various philosophical discourses, viz.: [237]. But we must — since it cannot be avoided (if we are to cover any ground at all) — postulate some possibilities of concept modelling (since that is what it, in essence, is all about). We do so in order to identify some designation (etc.) principles and techniques.

• • •

Some *designations* are specific, "one of a kind" things ("that rail unit there"), or they are specific events, or specific behaviours (etc.). Or *designations* are types or models (i.e., algebras) over these. Some designations are those of components of contexts and states. Contexts are entities whose properties — whose attributes, whose values — remain static over time. States are such whose values change with time — they are dynamic. Examples of contexts are: *road* and *rail nets* (when viewed topologically and over time periods that are not too large), similarly for airline *timetables*. Examples of states are: *road*, rail and air traffic, hence *trucks*, trains and aircraft, and the *hubs* (where they

meet and passengers embark and disembark, or where *freight receipt, transfer* and *delivery* can take place).

A *bill of lading* is somewhere "in between": The *route* along which a *freight item* should be conveyed is, we assume, static, but the *bill of lading* is (probably) marked ("updated") to show the (believed) current (or last recorded) position of the *freight item* for which it is the *bill of lading*.

But, in any case, all these examples can be modelled as typed, usually composite values, fixed or variable. These values we consider inert: They do not change by themselves. Some external action has to change them.

Designations thus could, alternatively, be the, or a, specific action (active phenomenon, *function,* operation, task, procedure) of *inscribing a freight item for conveyance with a logistics firm, loading it onto the truck, unloading it,* etc. In this case we refer to the designations as function values. Or designations could be the events of *truck departure* from, resp. *arrival at a hub*. Or designations could be part or the entire (process) behaviour *from inscription to unloading* described in terms of the specific sequence of inert and dynamic phenomena. In this case we refer to the designations as behaviours (a certain kind of function values).

• • •

The above explication presents just one kind of conceptual modelling approach. It is "slanted" towards CSP and RAISE [118, 301].

Principles. *Conceptual Frameworks:* When setting out on a first description, identify which conceptual modelling framework you intend to work within. ∎

Techniques. *Framework Model:* Among the conceptual modelling frameworks that can be believably offered are:

- B [4], VDM [35, 36, 104], Z [162, 341, 343, 377]
- RAISE [117, 118]
- CafeOBJ [85, 110, 111], CASL [28, 62, 251]

We list three groups. The first are solely model-oriented, the last is solely property-oriented (in the algebraic semantics style). RAISE basically offers both styles. If none of these is fully applicable, then be careful, very careful in choosing some "integrated formal specification" approach which takes a "little from this specification language", a "little from that other language", and so forth. If the underlying semantic models are compatible, then fine, or else there will be problems in securing an integrated semantics [16]. ∎

A "Large" Example

We illustrate the implications of the above by giving a formal definition in which we then identify various "designatable" quantities.

Example 7.6 *Transport:*

Formal Presentation: Formalisation of Transport

The example expands on the *italic text explications* given between the pair of • • • s above.

```
type
  Fre, BoL, Trk, Hub
value
  m,n:Nat
  obs_BoL: Fre → BoL
axiom
  m>0 ∧ n>0
type
  HIdx = {| 1..m |}, TIdx = {| 1..n |}
channel
  { ht[h,t]:(Fre×BoL) • h:HIdx, t:TIdx }
value
  truck: t:TIdx → Trk → Unit → in,out {ht[h,t]|h:HIdx}  Unit
  truck(t)(tr) ≡ (...; let h = ... in truck(t)(unload(tr)(h,t)) end)⌈⌉(...)
  hub: h:HIdx → Hub → in,out {ht[h,t]|t:TIdx} Unit
  hub(h)(hu) ≡ [] {hub(j)(add(ht[h,t]?)(hu))|t:TIdx} ⌈⌉ (...)
  load: Fre×BoL → t:TIdx × h:HIdx → in,out ht[h,t]  Trk
  unload: Trk → (h:HIdx × t:TIdx) → out {ht[i,t]|i:HIdx}  Trk
  unload(tr)(h,t) ≡
    let (f,b):(Fre×BoL)•ii(tr,f,b,h) in ht[h,t]!(f,b);rem(f,b)(tr) end
  ii: tr:Trk×f:Fre×b:BoL×h:HIdx → Bool /* b of f of tr mentions h */
  rem: Fre×BoL → Trk → Trk
  add: Fre×BoL → Hub → Hub
```

Fre, BoM, Trk and Hub name sets of freight items, bills of material, trucks and hubs. TIdx and HIdx name sets of index sets uniquely identifying trucks and hubs. The logistics system consists of n trucks and *m hub*s (here truck yards). The index sets enable us to model that any trucks can dock at any hub. The function names truck and hub model truck and hub behaviours. Channels ht[h,t] model interaction between hubs (h) and trucks (t). The functions load and unload effect the loading and unloading actions; is_in, remove and add name auxiliary functions. The truck and hub processes are cyclic (never ending). The ht[h,t]!(f,b) and ht[h,t]? clauses specify synchronisation and communication events (in CSP [168,301,311]). The event concept is thus expressed in terms of channels and synchronised output/input and communication. ∎

We remind the reader that the above shows just an example of how designations can be modelled — in the RAISE/CSP paradigm.

Formal Modelling

It is not the intention of the present section to suggest neither principles nor techniques for exactly how to formally model designations. Such principles and techniques were and are the subject of earlier volumes and later chapters. Instead it is the aim of the current section to point out the following principle:

Principle. *Choice of Description Style:* In considering which things to designate and to describe, let the informal description style be governed by the chosen, most suitable formal description style. ▪

That is, the describer, i.e., the software engineer, is well served in considering whether that which is to be described informally (rough-sketched, terminologised and narrated) can be given a formal model. If not, then perhaps what one is trying to describe informally is not the right thing to describe! Certainly, if it can be formalised, it can most likely be described well informally. And thus it may be something possibly worth describing.

7.2.5 Discussion: Designations

Principle. *Type Versus Value (Instantiation) Modelling:* In building up designations — within the conceptual paradigm (i.e., framework), say, of `RAISE` — one must decide whether one is trying to designate the type of all those things that are similar to a few designations, i.e., satisfy their common recognition rule — as for Fre, BoL, Trk and Hub above — or one is trying to define only a specific value (one particular thing) — as for hub, truck, load, unload, ii, rem or add above. ▪

This distinction is not, we find, made sufficiently clear in [189]. This is probably because that book, possibly wisely so, and definitely explicitly so, avoids committing itself to any specific formal specification paradigm.

In addition to modelling designations as *type*s, *channel*s and *value*s, one could, in the chosen conceptual framework of `RAISE`, think of other designation models, for example, *schemes*, *classes*, *objects* and *variable*s. Other conceptual frameworks would favour similar or quite dissimilar choices — and the general *designation* principles of [189] apply to any such paradigm.

• • •

We have, so far, only covered one of the three aspects of *descriptions: designations*. Two more remain to be covered: *definitions* and *refutable assertions*. Yet we have been able to demonstrate, in the formal example immediately above, that just by dealing with *designations* one can come a long way.

One may rightfully argue (i) whether all of what that last, albeit only sketched formal example showed was indeed a model only of *designatable* entities, and (ii) whether, as we shall soon see, introducing some *definitions*

might result in a different, perhaps more easily read description. One may also argue (iii) whether some of the axioms are, in fact, *refutable assertions*. We will return to these issues in the closing 'Discussion' sections of this chapter.

7.3 Explicit Definitions

First, designations represent one form of definition. In any description all terms that are special to the *universe of discourse* being described must be *defined* either through *designations* or by *explicit definitions*.

7.3.1 Definitions: "The Narrow Bridge"

The example above contained only designatable quantities, or so we claimed. But that may not always be possible, or convenient. Sometimes it is easier to *define* a `term` by giving it a *definition*, but (notably) using already *defined* (including *designated*) `terms`.

Example 7.7 *Rail Lines: A railway line is a linear, acyclic, sequence of linear rail units, that is, a sequence where adjacent elements of the sequence are rail units that share exactly one connector.*

Formal Presentation: Rail Lines

```
type
   U, C
   L1′ = U-set
   L1 = {| us:L1′ • wf_L1(us) |}
   L2′ = U*
   L2 = {| us:L2′ • wf_L2(us) |}
value
   obs_Cs: U → C-set
   wf_L1: U-set → Bool
   wf_L1(us) ≡ ∃ ul:U* • len ul = card us ∧ elems ul = us ∧ wf_L2(ul)
   wf_L2: U* → Bool
   wf_L2(ul) ≡
      ∀ u:U • u ∈ elems ul ⇒ is_linear(u) ∧
      ∀ i:Nat • {i,i+1}⊆inds ul ⇒
         card(obs_Cs(ul[i]) ∩ obs_Cs(ul[i+1]))=1 ∧
         len ul > 1 ⇒ obs_Cs(hd ul) ∩ obs_Cs(ul[len ul]) = {}
```

Here the terms linear rail unit and connector are already designated terms; and the concept of a line is defined in terms of those designations. Recall an earlier axiom which stated that for any connector there are at most two units sharing that connector. That axiom is assumed above. ∎

Techniques. *Definitions* must be expressed in terms of designated and/or defined terms, and ultimately definitions must be based on designations. ∎

Principles. Although it may seem utterly obvious we urge the developer to *develop alternative definitions.* ∎

Here we developed alternatives L1 and L2. Although one may claim that lines could be designations, since, in a sense, they are tangible, we have here defined the concept of line. We follow the advice of Michael Jackson [189] when we raise it to a principle:

Principle. *The Narrow Bridge:* Seek as few designations as possible: Enough to define all the other possibly designatable as well as all the desirable abstract, nontangible concepts. ∎

7.3.2 Definition of Abstract, Intangible Concepts

We shall next define some abstract, intangible concepts.

Example 7.8 *Path: A rail unit defines a concept of path. A path (through a rail unit) represents the ability for a train to move along that path, through the rail unit.*[2]

(Notice that we neither, at present, designate nor define what we mean by train or move.) *We define (i.e., we model) a path as a pair of connectors.* (Based on the definition of path we then proceed to define further intangible, i.e., not physically manifest, concepts.) *With any rail unit we can associate a state: a possibly empty set of paths. And, finally, we can associate with any rail unit its state space: the set of all the states that a rail unit may "be in" over its lifetime as a rail unit.*

Formal Presentation: Rail Unit States

```
type
   U, C, P = C×C
   Σ = P-set, Ω = Σ-set
axiom
   ∀ (c,c′):P • c≠c′
value
   obs_Σ: U → Σ, obs_Ω: U → Ω
```

In Sect. 7.4 we shall look at possible relations between designations and definitions of the railway system. ∎

[2] When we earlier spoke of a concept of link, that could, in a sense, be seen as an abstract concept, much like a pair of opposite direction paths.

7.3.3 How Much, How Little to Define?

The question is now: how much to *define?* For the *universes of discourse* of *requirements* and *software design* there is a *utility* concept: We know what we aim at, so we *define* at least that; but why more than that? For the *universes of discourse* of *application domains* we do not, as a matter of principle, know what we are aiming at, so here we go for more: as long as some interesting ideas are being *defined,* then why not? It is only, we believe, in "roaming around" and experimenting with *definitions,* and, later, with *refutable assertions,* that we may discover the *most interesting, most relevant, most fitting* domain concepts. It is in this exploration, and based on *definitions* that we can expect to build theories of specific domains.

Principle. *Exploring Theory Bases:* In constructing *domain models,* i.e., *descriptions* of *domains, designate* according to the "*narrow bridge*" principle, and then *define* as many abstract concepts as long as their *definition* (and those of attendant *refutable assertions* help) "reveals" further concepts. ▪

A last example may illustrate the previous point.

Example 7.9 *Routes: (i) A route is a sequence of connected rail units. (ii) A route is an open route if the states of all rail units of the route are such that there are paths in those states that also connect (in one direction or another). (iii) Given a unit within a station define as reachable from the station unit all those lines that are incident upon and (thus) emanate from that station, and for which there is a route between the unit and the line. (iv) Given any two rail units, an 'origin' and a possible 'destination', of a railway network define a more general notion of reachability, one that is satisfied if there are open routes from the 'origin' to the 'destination'.*

And so forth. The present example illustrates the definition of a number of (viz.: four) concepts without a priori making sure that these concepts will serve a useful purpose. ▪

7.3.4 Discussion: Definitions

Choosing an appropriate balance between a reasonably small set of designations and an appropriate (large, "rich") set of definitions is an art![3] No definite advice can be given, other than perhaps that given above, which, when followed, helps ensure an "appropriate balance".

We have now covered two of the informal and formal description principles and their formalisation techniques. From the examples so far we can already now conclude that one cannot see from the formulas whether a description formalises a designation or a definition! That is, the *pragmatics* of distinguishing between designation and definitions is lost in the formulas. Thus it

[3] Of course, the hedge "appropriate balance" justifies the property "is an art".

is important with some *syntax* to start the informal and formal texts in order to alert the reader to which is what!

7.4 Refutable Assertions

Characterisation. By a *refutable assertion* we mean a claim that may be shown to be wrong. ▪

7.4.1 Designation and Definition Assertions

In Example 7.5, where we described railway nets of lines and stations, we put in some bracketed sentence parts. Those parts did not designate manifest things. They expressed suitable, desirable or other relations of designations (and, later, of definitions). We shall refer to such constraints as *refutable assertions.*

Example 7.10 *Rail Net Assertion:* That *a line connects exactly two distinct stations* is one such example of a refutable assertion. It implies that there must be *at least two stations in a railway net* — another refutable assertion — and *at least one line* — yet another refutable assertion. We asserted elsewhere that *every connector is shared by at most two (otherwise distinct) rail units;* that *linear rail units have two connectors and define one link;* that *switchable rail units have three connectors (and define two links);* etc. These sentence parts can all be considered refutable assertions. ▪

Techniques. *Refutable assertions* must be expressed in terms of either designated or defined terms, or both kinds of terms. ▪

Designations, in a sense, are facts. And facts cannot be refuted. Definitions are definitions, and as such must be accepted. But definitions may seem general and may thus need to be characterised ("tied down") through some form of constraining assertions.

Example 7.11 *Unit States:* We continue the rail unit path, state and state space example given earlier. *The paths of a rail unit, i.e., the paths of any of its states, must mention only connectors of that unit.*

Formal Presentation: Unit States

axiom
$\forall$ u:U,c:C,σ:Σ •
obs_Σ(u) $\in$ obs_Ω(u) $\wedge$
$\sigma \in$ obs_Ω(u) $\Rightarrow \forall$ (c,c′):C×C • (c,c′) $\in \sigma \Rightarrow$ {c,c′}$\subseteq$obs_Cs(u)

The above assumes an earlier axiom expressing the fact that the connectors of a path are always distinct. ∎

This is a refutable assertion: One might think of cases where there could be connectors of a path in some unit's state that were not connectors of that unit, or could one? We basically will not know till someone one day shows us an understanding of a railway net that "favours" such an interpretation!

7.4.2 Analysis

What was claimed, above, as examples of refutable assertions, certainly were assertions. Whether they will be refuted remains to be seen, but they are potentially refutable. Take the example of *any connector* being "constrained" *to be shared by at most two rail units*. If one could show, in the future, that there were indeed connectors that were shared by three (or more) rail units, then we have indeed refuted the assertion. But is it likely?

Example 7.12 *Stations and Lines: A station is connected to at least one line.* ∎

Refuting this assertion would amount to allowing railway nets with completely isolated stations. Is that likely?

Example 7.13 *Stations and Open Routes: For any station, there is at least one potentially open route from, and at least one such route to, some other station of the net.* ∎

If this assertion is refuted then there will be at least one station from, or into which one cannot "travel"! Is that likely?

7.4.3 "Dangling" Assertions

It is much too easy to write down texts that "weave a web" of pseudodesignations, pseudodefinitions and pseudoassertions.

Example 7.14 *Freight Transport Bill of Lading:* Suppose the following text was all we had: *A bill of lading lists the transport hubs where the referenced freight items first loaded onto a conveyor are being transferred from one conveyor to a next, and are finally unloaded. A transport according to a bill of lading does not visit a hub more than at most once.* Then what were we to mean?

Suppose further that we went straight ahead and formalised the above *italic text:*

Formal Presentation: Freight Transport Bill of Lading

```
type
  Hub, HIdx, Fre, Con
  BoL′ = (F × (HIdx × HIdx* × HIdx))
  BoL = {| b:BoL′ • wf_BoL(b) |}
  Transport = Fre × (Con × Hub)*
value
  wf_BoL: BoL′ → Bool
  wf_BoL(f,(o,hl,d)) ≡ card({o,d} ∪ elems hl)=2+len bol

  obs_Fn: Fre → FIdx
  load,unload: Fre × BoL → Con → Con
  transfer: Fre × BoL → (Con × Con) → (Con × Con)
  M: BoL → Transport-infset
```

What is the problem? The problem is that we have taken the liberty of mentioning such possibly designatable phenomena as Hub, Hub index, Freight, Freight index, Conveyor, load, unload and transfer, and given signature to the function values without having attempted the slightest bit of designations! We have likewise defined a Meaning function which defines the meaning of a bill of lading as a possibly infinite set of defined *Transport*s.

The assertion *does not visit a hub more than at most once* is thus pretty meaningless. ▪

Techniques. *Dangling Assertions:* Make sure all terms of an assertion are either designated, defined or otherwise bound in the assertion. ▪

Example 7.15 *Transport Subtypes:* With respect to the previous example we should thus also have properly subtyped the *Transport* type, etc. ▪

7.4.4 Discussion: Refutable Assertions

Assertions are like axioms. They are self-evident truths — until refuted by observations made in the referenced universe of discourse. Assertions are thus not facts, nor are they theorems. Theorems are predicate expressions that can be proven to follow from designations, definitions and (the axioms of) refutable assertions. Theorems (lemmas, propositions) together with designations, definitions, the axioms of assertions and other theorems (lemmas, propositions) form a theory of the described universe of discourse.

7.5 Discussion: Description Principles

We have related the description principle concepts of designations, definitions and refutable assertions to (albeit sketchy examples of) formal specifications. We have, most recently above, seen how formal specifications, cf. the latest formalisation example above, without proper, necessarily informal descriptions that adhere to Jackson's principles, can too easily "drug" us into believing that the model identification principle (cf. Sect. 4.4) can be skipped.

We ourselves readily admit to having, far too often for anybody's good, fallen into that trap!

7.6 Bibliographical Notes

There is basically just one, major, overriding reference that "over- and foreshadows" the present chapter, and that is [189].

7.7 Exercises

7.7.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

7.7.2 The Exercises

Exercise 7.1 *Designations and Recognition Rules.* For the fixed topic, selected by you, list some five phenomena that can be designated. State suitable recognition rules.

Exercise 7.2 *Explicit Definitions.* For the fixed topic, selected by you, select some two or three designations that you instead decide to define explicitly in terms of other designations. Then provide these explicit definitions.

Exercise 7.3 *Refutable Assertions.* For the fixed topic, selected by you, try come up with at least one nontrivial possibly refutable assertion. (This may sound difficult, but be open-minded and allow extreme scepticism.)

Part IV

DOMAIN ENGINEERING

In the next nine chapters we shall cover, in some detail, the principles and techniques of the development stages and steps of the domain engineering phase:

- Chap. 8, Overview of Domain Engineering
- Chap. 9, Domain Stakeholders
- Chap. 10, Domain Attributes
- Chap. 11, Domain Facets
- Chap. 12, Domain Acquisition
- Chap. 13, Domain Analysis and Concept Formation
- Chap. 14, Domain Verification and Validation
- Chap. 15, Towards Domain Theories, and
- Chap. 16, The Domain Engineering Process Model

We refer to Chap. 1 for a comprehensive introduction to the three main phases of software development:

- Domain engineering
- Requirements engineering
- Software design

8

Overview of Domain Engineering

- The **prerequisite** for studying this chapter is that you are ready now to embark on the long journey of getting to understand the first of the three core phases of software development. You have understood the material of previous chapters, and, preferably also the (formal) abstraction and modelling principles and techniques of Vols. 1 and 2 of this series of textbooks on software engineering
- The **aims** are to present a capsule view of stages and steps of domain engineering, and to present a capsule view of the documents that result from domain engineering.
- The **objective** is to make you feel at ease with the very many stages and steps of domain development, and the very many parts of resulting documents.
- The **treatment** is informal and systematic.

8.1 Introduction

In this part, starting with the present chapter and going on for eight more chapters, we shall cover one of the three main software development activities: domain engineering. The other main activities are those of requirements engineering (Part V) and computing systems design (Part VI). They are considered main phases of software development in that everything else, i.e., all tools and management activities, group themselves around these three main sets of activities.

In this introductory chapter we shall briefly identify and briefly explain a number of issues that enter into domain engineering. Each of these issues will be dealt with in more detail in following chapters.

As has been argued before:

- Before we can design the software, we must understand its requirements.

- And before we can develop requirements, we must understand the application domain.

In Chap. 1 we reviewed domain engineering. Now we give a more systematic and comprehensive treatment. We shall emphasize principles, techniques and tools of domain engineering.

8.2 A Review of *Why Domain Engineering?*

Characterisation. By a *domain model* we understand the meaning of a domain description. ▪

Characterisation. By a *domain description* we mean a document (or a set of documents) which describes what the domain is, its entities, functions, events and behaviours. ▪

Characterisation. By a *domain theory* we mean a set of theorems that are claimed to hold of the domain model. ▪

Characterisation. By *domain engineering* we mean the processes overviewed in this chapter and otherwise detailed in this part (Part IV). ▪

Just as physicists have researched and developed models of Mother Nature for at least 500 years, and just as classical engineers have designed artifacts based on the theories of the natural sciences, so we shall advocate research into and the development of theories of the man-made domains in which human activities, rather than nature, play the major role. Then we can develop the requirements for and the designs of software in a more trustworthy and in a scientifically more believable manner.

To research and develop domain theories is a new activity. But many present software engineering processes already touch upon domain engineering. In these volumes we bring domain engineering *more out into the open*, thus simplifying many past concerns of software engineering, especially those of requirements engineering. That is, we strongly think that many previously — by other authors — advocated issues of requirements engineering become far easier to handle (or they outright "disappear") once we have done our domain engineering job! So we claim, at least!

8.3 Overview of Part and Chapter

Proper domain engineering, i.e., the proper development of a domain model, proceeds in stages:

- identification of domain stakeholders, Sect. 8.4 and Chap. 9

- domain acquisition, Sect. 8.5 and Chap. 12
- domain analysis and concept formation, Sect. 8.6 and Chap. 13
- domain modelling, Sect. 8.7 and Chaps. 10–11
- domain validation and verification, Sect. 8.5 and Chap. 14
- domain theory formation, Chap. 15

The reader may observe that we are presenting principles and techniques for each of these stages in not quite the order in which they are listed above. The reason is given now and is further elaborated upon later.

Domain Model and Domain Theory

The most important outcome of domain engineering is a domain model and its associated domain theory.

Without knowing what domain models contain one cannot know how to go about constructing them. Chapter 11 presents principles and techniques for what domain models contain. Chapters 12–13 outline how to gather material for domain model construction (domain acquisition) and how to analyse and understand such material (analysis and concept formation). But the issue, the role of stakeholders, is so important and often forgotten (or, at least, "minimised") that we have decided to present principles and techniques for identification of and liaison with stakeholders first, in Chap. 9. Chap. 10 is a preamble for Chap. 11.

8.4 Domain Stakeholders and Their Perspectives

Characterisation. 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. ▪

Identification of domain stakeholders embodies development principles, techniques and tools. These will be surveyed in Chap. 9.

Characterisation. By a *domain stakeholder perspective* we understand the, or an, understanding of the domain shared by the specifically identified stakeholder group — a view that may differ from one stakeholder group to another stakeholder group of the same domain. ▪

Identification of stakeholder perspectives (i.e., views) embodies development principles, techniques and tools. These will be surveyed in Sect. 9.3.

Domain Stakeholders

Without clearly identifying and liaising with all relevant domain stakeholders one cannot hope to construct a believable domain model.

We shall return to the concept of stakeholders in Chap. 9.

8.5 Domain Acquisition and Validation

Characterisation. By *domain acquisition* we understand the gathering, from domain stakeholders, from literature and from our observations, of knowledge about the domain. This knowledge includes phenomenological *entities, functions, events* and *behaviours,* with this "gathering" being manifested in terms of rough statements (i.e., fragments of sketches). ∎

Domain acquisition embodies many development principles, techniques and tools. These will be surveyed in Chap. 12.

Characterisation. By *domain validation* we understand the assurance, with stakeholders, notably clients, that the domain descriptions produced as a result of domain acquisition, domain analysis, concept formation and domain modelling (the latter including the description) is commensurate with how the stakeholders view the domain. ∎

Domain validation embodies many development principles, techniques and tools. These will be surveyed in Sect. 14.3.

8.6 Domain Analysis and Concept Formation

Characterisation. By *domain analysis* we understand a study of domain acquisition (rough) statements, with the aim of discovering inconsistencies, conflicts and incompletenesses within, as well as with the aim of forming concepts from, these domain acquisition statements. ∎

Domain analysis embodies many development principles, techniques and tools. These will be surveyed in sections of Chap. 13.

Characterisation. By *domain concept formation* we understand the abstraction of domain phenomena, as hinted at by domain acquisition (rough) statements, into concepts. ∎

Domain concept formation embodies development principles, techniques and tools. These will be surveyed in sections of Chap. 13.

8.7 Domain Facets

Characterisation. By a *domain facet* we understand one amongst a finite set of generic ways of analysing a domain, that is, a view of the domain such that the different facets cover conceptually different views, and such that these views together cover the domain. ▪

We list the main categories of domain facets:

- *business procedure* facets
- *intrinsic* facets
- *support technology* facets
- *management and organisation* facets
- *rules and regulations* facets
- *script* facets
- *human behaviour*

These facets will be covered in Chap. 11.

Domain Model ≡ Model of Domain Facets

So by a domain model we mean a set of one or more commensurate models of domain facets — these may possibly be rewritten (and reformalised) into one consolidated model.

8.8 Auxiliary Stages of Domain Development

Earlier we used the prefix design when enumerating some stages of development. Now we use the term auxiliary. Why we do this will transpire from the immediately following text.

The *auxiliary stages of development* include the following:

- domain (knowledge) acquisition
- domain (knowledge) analysis and concept formation
- domain (knowledge) verification
- domain (knowledge) validation
- domain theory formation.

We shall cover these in later sections. Suffice it for now to say that they "adorn" the major stages of domain facet modelling: to model a domain facet we must first acquire it; then we must analyse what has been acquired, and form concepts from what has been analysed; then we can describe it: (a) roughly, (b) terminologise it, (c) narrate and (d) possibly formalise the facet. Stages (a–d) form the major stages. In between these latter descriptive activities, we verify properties of the domain model, validate the domain facet description (i.e., the model), and possibly we build up elements of a theory of the domain.

8.9 The Domain Model Document

8.9.1 A Preview of Things to Come

The aim of domain engineering is to create informative, descriptive and analytic documents about and constituting the domain model. Therefore it is important to always keep in mind what a possible contents listing could be of such a complete set of documents. We shall therefore outline, in "capsule" form, what a possible, and, to us, desirable *table of contents* structure could be of such a set of domain documents. The aim of Part IV is, therefore, to present the principles, techniques and tools for creating, i.e., developing, such sets of domain documents.

8.9.2 Contents of a Domain Model Document

We list a comprehensive, desirable *table of contents* structure for a typical set of domain documents. We refer to Chap. 2 for an overview of these kinds of documents, and especially for the first category of informative documents.

A Generic Domain Documentation Contents Listing

1. Information
   - (a) Name, Place and Date
   - (b) Partners
   - (c) Current Situation
   - (d) Needs and Ideas
   - (e) Concepts and Facilities
   - (f) Scope and Span
   - (g) Assumptions and Dependencies
   - (h) Implicit/Derivative Goals
   - (i) Synopsis
   - (j) Standards Compliance
   - (k) Contracts
   - (l) The Teams
     - i. Management
     - ii. Developers
     - iii. Client Staff
     - iv. Consultants
2. Descriptions
   - (a) Stakeholders
   - (b) The Acquisition Process
     - i. Studies
     - ii. Interviews
     - iii. Questionnaires
     - iv. Indexed Description Units
   - (c) Terminology
   - (d) Business Processes
   - (e) Facets:
     - i. Intrinsics
     - ii. Support Technologies
     - iii. Management and Organisation
     - iv. Rules and Regulations
     - v. Scripts
     - vi. Human Behaviour
   - (f) Consolidated Description
3. Analyses
   - (a) Domain Analysis and Concept Formation
     - i. Inconsistencies
     - ii. Conflicts
     - iii. Incompletenesses
     - iv. Resolutions
   - (b) Domain Validation
     - i. Stakeholder Walkthroughs
     - ii. Resolutions
   - (c) Domain Verification
     - i. Model Checkings
     - ii. Theorems and Proofs
     - iii. Test Cases and Tests
   - (d) (Towards a) Domain Theory

8.10 Further Structure of This Part

We start with a brief analysis of the stakeholder concept (Chap. 9). To know how to properly acquire domain knowledge we believe that it is important to know what the end result of domain engineering should be. We therefore detail two core aspects of a domain model: the attributes of the phenomena and concepts modelled (Chap. 10), and the facets of domain phenomena and concepts (Chap. 11). Thus we present principles and techniques for those aspects of domain models. And we do so before we treat principles and techniques for domain acquisition (Chap. 12). Then we cover domain analysis and concept formation (Chap. 13) — on which the domain models build. Once domain models are believed ready, they can be validated (Section 14.3), and stages and steps of domain modelling work can be verified (Sect. 14.2) — often during domain modelling. Chapters 15 and 16 end this part: They deal with thoughts (very briefly) on domain theories, and summarise the domain engineering process model.

We emphasise, to the reader, that the order of chapters of this part does not follow the order of the work to be done in domain development. We repeat: Before we can do proper domain acquisition (Chap. 12), concept analysis and formation work (Chap. 13), we must understand what the form and contents of proper domain models should desirably be (Chaps. 10 and 11). Hence Chaps. 10 and 11 come before Chaps. 12 and 13. It is to *keep our tongues and fingers straight* that we presented the *table of contents* structure for a typical set of domain documents in Sect. 8.9.2.

8.11 Bibliographical Notes

Our approach to domain engineering possesses some rather novel features. That is, we bring new principles and techniques into software engineering — namely the entire concept of domain engineering — that are not covered elsewhere in the currently available literature on software engineering [121, 275, 284, 338, 369].

8.12 Exercises

The exercises of this chapter are *closed book* exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions.

Exercise 8.1 *Why Domain Engineering?* Without consulting chapter texts in this volume, recapitulate, in a few lines of informal text, how this book motivates domain engineering.

Exercise 8.2 *Stages of Domain Engineering.* Without consulting chapter texts in this volume, recapitulate, in some six or so lines of informal text, the ordered stages of domain engineering.

Exercise 8.3 *Substages of Domain Modelling.* Without consulting chapter texts in this volume, recapitulate, in some seven or so lines of informal text, the ordered stages of domain facet modelling.

Exercise 8.4 *Domain Acquisition.* Without consulting chapter texts in this volume, characterise, in a few lines, how this chapter defines domain acquisition.

Exercise 8.5 *Domain Validation.* Without consulting chapter texts in this volume, characterise, in a few lines, how this chapter defines domain validation.

Exercise 8.6 *Domain Analysis.* Without consulting chapter texts in this volume, characterise, in a few lines, how this chapter defines domain analysis.

Exercise 8.7 *Domain Concept Formation.* Without consulting chapter texts in this volume, characterise, in a few lines, how this chapter defines domain concept formation.

Exercise 8.8 *Stakeholder.* Without consulting chapter texts in this volume, characterise, in a few lines, how this chapter defines the concept of a domain stakeholder.

Exercise 8.9 *Stakeholder Perspective.* Without consulting chapter texts in this volume, characterise, in a few lines, how this chapter defines the concept of domain stakeholder perspective.

Exercise 8.10 *Domain Documentation.* Without consulting chapter texts in this volume, list, in as exhaustive and structured a fashion as possible, generic domain documentation table of contents.

9

Domain Stakeholders

- The **prerequisite** for studying this chapter is that you have read Chaps. 1 and 8 of this volume.
- The **aims** are to introduce the concept of (domain) stakeholders, to distinguish between different categories of stakeholders, and to sketch a fairly advanced (also formalised) example of enterprise stakeholders.
- The **objective** is to ensure that you carefully consider and include the concerns of all relevant stakeholders when in future you are developing domain descriptions, requirements prescriptions and software designs.
- The **treatment** is from systematic to formal (sketches).

9.1 Introduction

At the very outset of any phase of development, whether the universe of discourse be some domain model development, requirements development or software design, it is important to identify all possibly relevant stakeholders. Throughout the development phase it is then important to ensure that each stakeholder (group) is properly "taken care of", i.e., their concerns are properly modelled.

9.2 Stakeholders

Characterisation. 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. ▪

Obviously we could "equate" institutions and enterprises with groups of one or more persons. For pragmatic reasons of identification (i.e., "discovery") it is, in cases, sometimes easier, we believe, to think of institutions and enterprises.

9.2.1 General Application Stakeholders

Characterisation. 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. ∎

Thus general application domain stakeholders are typically those whom we can characterise as from such domains as: transportation, manufacturing, mining, financial industries, public government, the service sector, etc.

Example 9.1 *Railway Train System Stakeholders:* When modelling, i.e., describing, the domain of railways, one may be well advised in considering the following stakeholder groups — listed in an order that may reflect the view of the first group: (i) owners (e.g., stockholders or a government), (ii) management (consisting of (ii.1) executive management, (ii.2) mid-level management, (ii.3) operational ("floor") management (i.e., "white collar workers"), etc.), (iii) railway staff at large ("people on the floor" other than "floor" management, i.e., "blue collar" workers — and possibly arranged into several stakeholder groups including families), (iv) customers ((iv.1) passengers and (vi.2) freightors (people etc., sending and receiving freight)), (v) users ((v.1) people coming to send off or receive passengers, and (v.1) people coming to send off or receive freight), (vi) agents ((vi.1) travel bureaus, and (vi.2) logistics firms), (vii) railway infrastructure companies,[1] (viii) suppliers ((viii.1) of day-to-day materiel (consumables), (viii.2) of new railway infrastructure components (i.e., lines, tracks, signals, etc.), and (viii.3) of information technology and software), (ix) railway regulatory agency or agencies, (x) politicians "at large", and (xi) the general public, "at large". ∎

The above example is typical of the kind of rough sketch, or even narrative, documentation text that the software developer has to produce in the course of describing a domain. But the above list is merely indicative, not final. It is also given here to "augment" our characterisation of what is meant by the stakeholder concept. Thus you can take this listing as a cue to construct similar stakeholder listings for other domains.

9.2.2 Software Development Stakeholders

One can identify two extremes of software (SW) developments: turnkey software and commercial off-the-shelf software (COTS).

[1] So we are considering a train service operator, in contrast to those who own (and operate) the rail net. In many countries these are two distinct groups of enterprises.

Turnkey Software Development Stakeholders

Characterisation. By *turnkey software* we understand software that is developed — usually from "almost scratch" — in very specific response to a specific client/developer contract. ▪

Characterisation. By a *turnkey software development stakeholder* we thus understand a stakeholder from the software developer or from that client domain. ▪

Stakeholders from this "extrovert" domain are (thus) typically: (i) The client (i.1) contract management, (i.2) client users and (i.3) customers, affected by the contracted software; and (ii) the software house (ii.1) contract management, (ii.2) software engineers and (ii.3) supporting technicians.

Commercial Off-the-Shelf (COTS) SW Development Stakeholders

By COTS we mean generic kinds of software, i.e., software whose functionality is as much, or more, decided upon by the makers of the software than by the customers and users of the software; software which is expected to cover, or actually covers the needs of many clients, and which the maker thus expects to sell in dozens, hundreds or thousands of copies.

Characterisation. *COTS Stakeholder:* By *COTS stakeholders* we (thus) typically mean: people from software houses: (i–ii) software house owners and management (at least two groups), (iii–v) marketing, sales and service departments (three groups), (vi) the programmers, i.e., the software engineers, (vii) distributors of the software, and (viii) other software houses which base tailor-made software development on COTSs; as well as people from the application domains for which the software house makes these products: (ix) customers (clients) and (x) users. ▪

9.2.3 Purpose of Listing Stakeholders

Lest we forget it, let us remind ourselves why we wish to systematically record all possibly relevant stakeholder groups: It is so that we can systematically and "near exhaustively" consider all relevant stakeholder groups, when we now go on to ascertain their view of, their perspective of, the universe of discourse — here the domain.

9.3 Stakeholder Perspectives

Characterisation. By a *stakeholder perspective* we understand the, or an, understanding of the domain shared by the specifically identified stakeholder group — a view that may differ from one stakeholder group to another stakeholder group of the same domain. ▪

For each stakeholder group we have to investigate (elicit, acquire, and analyse) its perspective with respect to each of the possible domain attributes, as covered next, and each of the possible domain facets, as covered thereafter. With respect to stakeholder perspectives we may be prepared to observe that one and the same phenomenon may be considered by two different groups to possess not quite commensurate attributes, and not quite commensurate facets. And thus two or more such group perspectives can give rise to inconsistent, and/or conflicting overall views on domain attributes and facets. We shall return to the above issues when we later treat the methodological concerns of domain acquisition and validation.

9.3.1 Perspectives of General Applications

The stakeholder perspectives for general application domains are generally of several concerns:

(i–ii) *Client executive and other upper-level management* expects computing systems to improve their company's competitiveness, financial position, etc. These are issues that are very hard to formulate, let alone formalise. Under informative documents we list part of these concerns under the headings *assumptions and dependencies* and *implicit/derivative goals.* For more on this we refer to Sect. 2.4.6.

(iii) *Tactical and operational management* usually have perspectives that pertain to management and organisational issues.

(iv) *Nonmanagement staff* usually have perspectives that pertain to their daily work and to its interface with customers.

(v) All of the above stakeholder groups have perspectives that primarily focus on their shared domain: the general application area.

(vi) This is in contrast to the perspectives of stakeholders of the software house, the developer with whom the client contracts.

(vii) Besides wishing to secure, in their perspectives, the professional integrity of their company, the *software house developer* perspectives include those of satisfying the client.

Example 9.2 *Resource Management:* We now present a rather lengthy example that illustrates the interface between a number of stakeholder perspectives. The stakeholders are (simplifying): an enterprise's top level, executive management (who plan, take and follow up on strategic decisions), its line management (who plan, take and follow up on tactical decisions), its operations management (who plan, take and follow up on operational decisions) and the enterprise "workers" (who carry out decisions through tasks). The management groups have the following kinds of functions. Strategic management has to do with upgrading or downsizing, i.e., converting an enterprise's resources from one form to another — making sure that resources are available for tactical management. Tactical management has to do with temporal, typically medium- to long-term scheduling and spatially allocating these resources,

in preparation for operations management. Operations management plans final (usually short-term) scheduling and allocation of (resource-consuming) tasks, in preparation for actual enterprise ("floor") operations.

After some analysis we arrive at the following: Let R, Rn, L, T, E and A stand for resources, resource names, spatial locations, times, enterprises (with their estimates, service and/or production plans, orders on hand, etc.), respectively tasks (actions). SR, TR and OR stand for strategic, tactical and operational resource views, respectively. SR expresses (temporal) schedules: which sets of resources are either bound or free in which (pragmatically speaking: overall, i.e., "larger") time intervals. TR expresses temporal and spatial allocations of sets of resources, in certain (pragmatically speaking: model finer-grained, i.e., "smaller") time intervals, and to certain locations. OR expresses that certain actions, A, are to be, or are being applied to (parameter-named) resources in certain time intervals.

Formal Presentation: Resource Management

type R, Rn, L, T, E
  RS = R-**set**
  SR = (T×T) $\overrightarrow{m}$ RS, SRS = SR-**infset**
  TR = (T×T) $\overrightarrow{m}$ RS $\overrightarrow{m}$ L, TRS = TR-**set**
  OR = (T×T) $\overrightarrow{m}$ RS $\overrightarrow{m}$ A
  A = (Rn $\overrightarrow{m}$ RS) $\xrightarrow{\sim}$ (Rn $\overrightarrow{m}$ RS)
value
  obs_Rn: R → Rn
  srm: RS → E×E $\xrightarrow{\sim}$ E × (SRS × SR)
  trm: SR → E×E $\xrightarrow{\sim}$ E × (TRS × TR)
  orm: TR → E×E $\xrightarrow{\sim}$ E × OR
  p: RS × E → **Bool**
  ope: OR → TR → SR → (E×E×E×E) → E × RS

The partial, including loosely specified, and in cases nondeterministic functions srm, trm and orm stand for strategic, tactical, respectively operations resource management. p is a predicate which determines whether the enterprise can continue to operate (with its state and in its environment, e), or not. To keep our model small we have had to resort to a "trick": putting all the facts knowable and needed in order for management to function adequately into E. Besides the enterprise itself, E also models its environment: that part of the world which affects the enterprise.

There are, accordingly, the following management functions:

Strategic resource management: srm(rs)(e,e′′′′). Let us call the result (e′,(srs,sr)) [see "definition" of the enterprise "function" below]. srm proceeds on the basis of the enterprise: as it is now (e), and as one would like it to become (e′′′′), as well as its current resources (rs). srm "ideally estimates" all

possible strategic resource acquisitions (upgrading) and/or downsizings (divestments) (srs). And srm selects one desirable strategic resource schedule (sr). The "estimation" is heuristic. Too little is normally known to compute sr algorithmically. One can, however, based on careful analysis of srm's pre/-postconditions, usually provide some form of computerised decision support for strategic management.

Tactical resource management: trm(sr)(e,e′′′′). Let us call the result (e′′,(trs,tr)). trm proceeds on the basis of the enterprise: as it is now (e), and as one would like it to become (e′′′′), as well as one chosen strategic resource view (sr). trm "ideally calculates" all possible tactical resource possibilities (trs). And trm selects one desirable tactical resource schedule and allocation (tr). Again, trm cannot be fully algorithmitised. But some combinations of partial answer computations and decision support can be provided.

Operations resource management: orm(tr)(e,e′′′′). Let us call the result (e′′′,or), orm proceeds on the basis of the enterprise: as it is now (e), and as one would like it to become (e′′′′), as well as one chosen tactical resource view (tr). And orm effectively decides on one operations resource view (or). Typically orm can be algorithmitised — applying standard operations research techniques.

Actual enterprise operation: ope, enables, but does not guarantee, some "common" view of the enterprise. ope depends on the views of the enterprise, its context, state and environment, e, as "passed down" by management; and ope applies, according to prescriptions kept in the enterprise context and state, actions, a, to named (rn:Rn) sets of resources.

The above account is, obviously, rather idealised, but, we hope it is indicative of what is going on. Relating the above schematic example to, for example, the railway domain we may suggest: Resources R include access to (not necessarily ownership of) the rail net, rights to rent passenger train carriages and locomotives, staff, monies, etc. Strategic resources is, for example, about needing additional or changed rail net access rights, needing further or different kinds of train sets, etc. Strategic resource management, srm, typically brings many operators together, negotiating with rail infrastructure owners about access rights and with train set leasing (and lease finance) companies for rental of train sets, etc. srs:SRS designates all possible outcomes of a company's own strategic planning; sr:SR designates a negotiated solution. Tactical resources is, for example, now about the rostering of train staff (crew allocation), allocation of train sets to maintenance locations, etc. Tactical resource management, trm, typically involves negotiation with trade unions, with maintenance units, etc. trs:TRS designates all possible outcomes of a company's own tactical planning (its negotiating options); tr:TR designates a negotiated solution.

To give a further abstraction of the "life cycle" of the enterprise, we idealise it, as now shown:

value

```
enterprise: RS ~→ E ~→ Unit
enterprise(rs)(e) ≡
  if p(rs)(e) then
    let (e′,(srs,sr)) = srm(rs)(e,e′′′′),
        (e′′,(trs,tr)) = trm(sr)(e,e′′′′),
        (e′′′,or) = orm(tr)(e,e′′′′),
        (e′′′′,rs′) = ope(or)(tr)(sr)(e,e′,e′′,e′′′) in
    let e′′′′′:E • p′(e′′′′,e′′′′′) in
    enterprise(rs′)(e′′′′′) end end
  else stop end

p′: E × E → Bool
```

The enterprise reinvocation argument, rs′, a result of operations, is intended to reflect the use of strategically, tactically and operationally acquired, spatially and task allocated and scheduled resources, including partial consumption, "wear and tear", loss, replacements, etc.

The **let** e′′′′′:E • p′(e′′′′,e′′′′′) **in** ... shall model a changing environment.

There were two forms of recursion at play here: The simple tail-recursion (i.e., the recursive invocation of enterprise), and the recursive "build-up" of the enterprise state e′′′′. The former is trivial. The latter is the interesting one: Solution, by iteration towards some acceptable, not necessarily minimal fix-point, "mimics" the way the three levels of management and the "floor" operations change that state and "pass it around, up and down" the management hierarchy. The operate function "unifies" the views that different management levels have of the enterprise, and influences their decision making. Dependence on E also models potential interaction between enterprise management and, conceivably, all other stakeholders.

We remind the reader that — in the previous example — we are "only" modelling the *domain!* That model is, obviously, sketchy. But we believe it portrays important facets of domain modelling and stakeholder perspectives. The stakeholders were, to repeat: strategy ("executive") management (srm, p), tactical ("line") management (trm), operations ("floor") management (orm) and the workers (ope). The perspective being modelled focused on two aspects: their individual jobs, as "modelled" by the "functions" (srm, p, trm, orm, ope), and their interactions, as "modelled" by the passing around of arguments (e, e′, e′′, e′′′, e′′′′). The **let** e′′′′′:E • p′(e′′′′,e′′′′′) **in** ..., which "models" the changing environment, thus summarises the perspectives of "all other" stakeholders!

We are modelling a domain with all its imperfections: We are not specifying anything algorithmically; all functions are rather loosely, hence partially defined; in fact only their signature is given. This means that we model well managed as well as badly, sloppily or disastrously managed enterprises. We can, of course, define a great number of predicates on the enterprise state and

its environment (e:E), and we can partially characterise intrinsics — facts that must always be true of an enterprise, no matter how.

If we "programme-specified" the enterprise then we would not be modelling the domain of enterprises, but a specifically "business process engineered" enterprise. Or we would be into requirements engineering — we claim. ∎

9.3.2 Perspectives of Software Development

If the application domain is that of software development itself then the domain stakeholders are primarily the software house owners and upper management, the software engineers and their immediate managers, the technicians who support the work of the software engineers, and the suppliers of technology (hardware and software) that support the work of management, software engineers and technicians. This is true whether the software development is either just domain engineering, or just requirements engineering, or just software design, or the first two, the last two or all three of the above. We stress the precondition: "if the application domain is that of software development itself". Or, put it differently, the subject domain of these volumes is software development itself.

9.4 Discussion: Stakeholders and Their Perspectives

9.4.1 General

We refer to Chap. 18 for a treatment of requirements stakeholders. This chapter has discussed the concept of stakeholders. In subsequent chapters we shall take up the thread and, occasionally, indicate where we differentiate, in our descriptions, etc., between perspectives of different stakeholders. In Chap. 10 this will not be an issue, but in Chap. 11's treatment of business processes and management and organisation we may occasionally refer to the need for special descriptions of stakeholder perspective.

9.4.2 Principles, Techniques and Tools

Principle. *Domain Stakeholder:* At the very outset of a development project identify all possible and potential domain stakeholders. It is better to include too many, than forget some who can later create a nuisance, or more, when rightfully intervening. Be prepared, throughout a project, to revise the list of domain stakeholders. ∎

Principle. *Domain Stakeholder Perspective:* At the very outset of a development project define, together with designated domain stakeholders, their roles, their "jurisdictions" and their "rights and duties". Be prepared, throughout a project, to revise the roles of domain stakeholders. ∎

Techniques. *Domain Stakeholder Liaison:* (i) Maintain, openly inspectable, lists of all contemplated, respectively of all actual domain stakeholders. (ii) Liaise regularly with all actual domain stakeholders. (iii) Inform all other (contemplated) domain stakeholders of "what's going on". (iv) Write down in clear (natural, yet legally binding) language the role of each actual stakeholder. (v) Maintain a dossier of all communications with all domain stakeholders. Typically such communications deal, as we shall see later, with: role assignments, acquisition and validation. ▪

Tools. *Domain Stakeholder Liaison:* The tools mentioned under information documents (Sect. 2.4.10) apply equally well here. ▪

9.5 Exercises

9.5.1 Preamble

The first 4 exercises (9.1–9.4) of this chapter are *closed book* exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions.

9.5.2 Assignments

Exercise 9.1 *Domain Stakeholder.* This is a "repeat" question (see Exercise 8.8): Without consulting chapter texts in these volumes, try to characterise, in a few lines, how this chapter defines the concept of a domain stakeholder.

Exercise 9.2 *Domain Stakeholder Perspective.* This is a "repeat" question (see Exercise 8.9): Without consulting chapter texts in these volumes, try to characterise, in a few lines, how this chapter defines the concept of domain stakeholder perspective.

Exercise 9.3 *General Application Versus Software Development Stakeholders.* Without consulting chapter texts in these volumes, try to enumerate, in three or so lines, how this chapter perceives of a spectrum from general application stakeholders to software development stakeholders.

Exercise 9.4 *General Application Versus Software Development Stakeholder Perspectives.* Given your answer to the previous exercise (Exercise 9.3 above), augment the answer by providing each entry in your enumerated list with a brief characterisation of corresponding perspectives.

Exercise 9.5 *Domain-Specific Stakeholders.* For the fixed topic, selected by you, present as exhaustive a list of stakeholders as you think is relevant for your domain modelling.

Exercise 9.6 *Domain-Specific Stakeholder Perspectives.* Given your answer to the previous exercise (Exercise 9.5 above), augment the answer by providing for each entry in your enumerated list a brief characterisation of corresponding domain-specific perspectives.

9.5.3 Postlude

Exercises similar to the above will be reposed in Sect. 18.5.2.

10

Domain Attributes

- The **prerequisite** for studying this chapter is that you have studied the abstraction and modelling chapters in Vols. 1 and 2 and that you are in search of further principles and techniques needed for modelling phenomena and concepts.
- The **aims** are to introduce four sets of phenomena and concept notions: (i) continuity, discreteness, hybridicity and chaos; (ii) statics and dynamics (where the latter exhibits a rich structure of subnotions); (iii) tangibility (and hence intangibility); and (iv) dimensionality; and to present principles and techniques for modelling such phenomena and concepts.
- The **objective** is to round off the long line of abstraction and modelling principles in Vols.1 and 2 and thus to further you on the road to becoming a professional software engineer.
- The **treatment** is systematic to formal.

The ideas of Sects. 10.3–10.5 are clearly inspired by, and hence rather directly derive from and, in modest ways, extend, and in some ways reinterpret Michael Jackson's work [189].

10.1 Introduction

Volumes 1 and 2 of this series of volumes covered a great many abstraction and modelling principles and techniques. Each of these applies to specifically characterisable phenomena and concepts. To wit, taken from Vol. 2, we mention: hierarchical and compositional phenomena, denotable and computable concepts, contextual and state concepts, temporal and spatial phenomena, etc.

In this chapter we shall take up this thread of specifically characterisable phenomena and concepts. We shall, so to speak, add a few more such specifically characterisable phenomena and concepts. They are continuity, discreteness, hybridicity and chaos (Sect. 10.2); statics and dynamics (Sect. 10.3); tangibility and intangibility (Sect. 10.4); and dimensionality (Sect. 10.5). Within

dynamic phenomena and concepts (Sect. 10.3.2) there is a rich substructure of further characterisable phenomena and concepts. They include: inert, active (autonomous, biddable, programmable) and reactive phenomena and concepts.

10.2 Continuity, Discreteness and Chaos

The notions of *continuity*, *discreteness* and *chaos* are — in this section — primarily considered as notions related to phenomena exhibiting *temporal* behaviour. The notion of *hybridicity* is here considered likewise, and covers phenomena which at any time exhibit both *continuous* and *discrete* behaviour. But continuity, discreteness, hybridicity and chaos need not exclusively be associated with temporal behaviours. The examples of this section will, however, exclusively illustrate temporal phenomena. After this section on continuity, discreteness, hybridicity and chaos has been studied it should be clearer, to the reader, how to take the "temporality" ideas and "lift" them to arbitrary types.

10.2.1 Time

Volume 2, Chap. 5 brought in various axiom systems for time. They were all embodied in the set of axioms due to Johan van Benthem. Designated, i.e., specific, subsets of these axioms form one or another notion of time: time into the past, from some present or future, or more recently past definite time; time into the future, from some such time; circular time; etc. When, in the following, we make use of some time notion, and, correspondingly, when you make use of some time notion, it is advisable to state which time notion one is using.

10.2.2 Continuity

Characterisation. A domain phenomenon is said to possess the *continuity* attribute if a suitable, perhaps even a "best", abstract model describes it as a continuous function from some point set value type A (A could be time) to some value type (B):

type
  A, B
  Phenomenon = A → B

B need not be a point set type. ∎

By a point set value type is understood a mathematically dense set of values, where density is in the mathematical sense of the differential calculi.

Example 10.1 *Air Traffic:* Let AirSpace, T, F, Fn and P stand for the airspace, time, flight (aircraft), unique flight identifications (names) and (aircraft) position types. Here it is acceptable to think of the type T as a dense interval of reals, from some recent past, to some not too distant future time. Then we can model air traffic, AT, as a continuous function from time to positions of aircraft.

Formal Presentation: Air Traffic

```
type
   AS, T, F, P, Fn
   FPs = F ~m~> P
   AT′ = T → (AS × FPs), AT = {| at:AT′ • wf_AT(at) |}
value
   wf_AT: AT′ → Bool
   obs_Fn: F → Fn
```

Here the well-formedness of air traffic expresses laws of nature: (a) At two infinitesimally close times (t, t', properly within the definition set of the air traffic function), if a flight (i.e., one of a given name) is in the airspace at both times, then their positions are infinitesimally close in space. (b) At two times of the observed air traffic if a flight (of the same name) is in the air traffic, then that flight (of that name) is in the air traffic at all times in-between ("no ghost flights"). (c) All flight positions are within the airspace, etc.

Formal Presentation: Air Traffic Well-formedness

Let Δ be an infinitesimally small time interval, let icis be an infinitesimally close in space predicate, and let $\mathcal{D}$ be a metafunction that applies to functions and yields their definition set.

```
type
   TI
value
   Δ:TI axiom Δ = 0.00...01
   icis: P × P → Bool
   wf_AT(at) ≡
      ∀ t,t′:T • {t,t′}⊂𝒟(at)
         let ((as,fps),(,fps′)) = (at(t),at(t′)) in
         ∀ f,f′:F • f ∈ dom fps ∧ f′ ∈ dom fps′ ∧
            (a) obs_Fn(f)=obs_Fn(f′)⇒t′=t+Δ⇒icis(fps(f),fps′(f′)) ∧
            (b) ∀ t″:T•t″ ∈𝒟(at)∧t<t″<t′⇒
                  ∃ f″:F•obs_Fn(f″)=obs_Fn(f′)⇒f″isin dom at(t″) ∧
            (c) is_in_airspace(f,as) end
```

The definition set function is a mathematically, but not an RSL-definable function.

We must resort (not shown here) to modelling continuity (viz.: $\mathcal{D}$, infinitesimally close in space, is in airspace) as is done in classical mathematical calculus.

■

Principles. *Continuitiess:* When a domain phenomenon can best be described as a function then it should be so described — whether partial or total. ■

Techniques. Deploy the *continuity* technique, as implied by the formulation of the above principle, of using, that is, defining, applying and composing partial or total functions. ■

10.2.3 Discreteness

Characterisation. A domain phenomenon is said to possess the *discreteness* attribute if a suitable, perhaps even a "best" abstract model describes it as a (hence discrete) map from some type A to some type B:

type
A, B
Phenomenon = A $\overrightarrow{m}$ B

Whether the phenomenon then "records" times equidistantly is not necessarily a part of the basic discreteness attribute. Neither, still assuming the kind 'time', is any assumption made on any notion of 'distance' between "successive" time values. That is, we could "subdivide" discrete phenomena into equidistantly discrete, and nonequidistantly endowed phenomena. ■

Example 10.2 *Air Traffic:* We continue our air traffic example. When a radar observes air traffic a discretisation takes place.

Formal Presentation: Discretised Air Traffic

type
FPs = F $\overrightarrow{m}$ P
rAT′ = T → (AS × FPs)
rAT = {| rat:rAT′ • wf_rAT(rat) |}
dAT′ = T $\overrightarrow{m}$ (AS × FPs)
dAT = {| dat:dAT′ • wf_dAT(dat) |}
value

wf_rAT ≡ wf_AT
wf_dAT: dAT′ → **Bool**
wf_dAT(dat) ≡ /∗ a simple variant of wf_AT ∗/
ideal_radar: rAT → dAT

We leave the definition of wf_dAT to the reader.

■

On a Notion of State

We can associate a notion of state with a *phenomenon* behaving as described for the type Phenomenon abstracted above: At any observable time the *phenomenon* takes on a value. An altogether different way of conceiving of a discrete state notion can be thought of. We can think of one in which the next state value depends on the current state and an input stimulus from among a finite set of such stimuli. We can then speak of the observable value as being an observable output result. Now the stimuli are provided, over time:

type
Time, Stimulus, Value, State, Result
Tick == tick_event | stable
Stimuli = Time $\overrightarrow{m}$ Stimulus
Phenomenon = Stimulus → Value $\overrightarrow{m}$ Value
value
obs_Event: Stimulus → Tick
obs_State: Value → State
obs_Result: Value → Result

The phenomenon changes state only when a tick_event occurs. Any stimulus is provided continuously, but the phenomenon only samples the stimulus when a tick_event occurs. Occurrence of such tick_events are thought of as being instantaneous: to occur at a certain time, and to not occur in larger intervals of time immediately before and after an event. The length of these intervals may vary indefinitely.

Methodological Consequences

We summarise:

Principles. *Discretenesss:* When a domain phenomenon can best be described as a discrete map then it should be so described. ■

Techniques. Deploy the *discreteness* technique, as implied by the formulation of the above principle, of using, that is, defining, applying and composing maps. ■

Hybridicity

Characterisation. *Hybridicity:* A system is considered to possess the hybrid attribute if it both possesses the continuity and the discreteness attributes **and** these relate, somehow, to one another. ▪

That is: Some phenomenon (or phenomena) of the system possess the continuity attribute; another, or others, the discreteness attribute. Where, above, under continuity and discreteness, in considering "a system", we focused primarily on single, "indivisible" phenomena, we now consider composite phenomena.

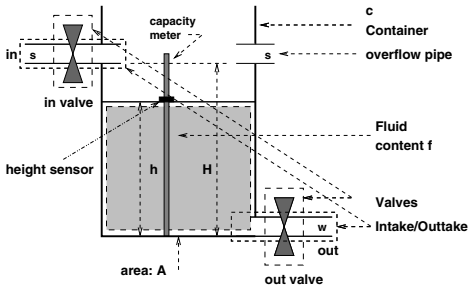


Fig. 10.1. A water tank system

Example 10.3 *Water Tank System:* We illustrate a water tank system in Fig. 10.1. The figure shows a projection, a cross-sectional "cut", of a "real" liquid tank system onto a vertical plane. The large ⊔-shaped form in the middle of the picture illustrates the container (the tank, c). To it is affixed three pipes: upper left, upper right and lower right. Two of these pipes, shown as pair of horizontal lines, are "adorned" with butterfly-like ⧖ X-shapes. They are meant to designate (open, close controllable) valves. In the center of the picture is a vertical, pole-like figure. It shall designate a tank (height) meter. The dark, small rectangle on the pole, seemingly "floating" on the liquid, shall designate the (movable) height meter. The main shaded area in the middle of the picture shall designate a momentary water content of the tank. The five horizontal and three diagonal and dashed single arrows "point" to named components. The two vertical and dashed double arrows (labelled h and H) indicate height measurements. The dashed boxes designate separably identifiable components. Various single-letter identifiers, otherwise referred to in the text around this figure, are respectively affixed to components whose attributes they designate.

This water tank system is a hybrid system: It is discrete in that valves are in either of two states — and with two valves of interest there are thus

four identifiable discrete states. It is continuous in that depending on valve settings, water (i) flows into the tank, (ii) and/or out of the tank through valves, (iii) flows out of the tank through the overflow pipe, (iv) precipitates into the tank during rain falls and (v) evaporates out from the tank — all of these flows being continuous. In addition to the discrete valve states there is also the water height state of either being below the high mark, H, or at that mark.

We take an advance look at some dynamic attribute phenomena — which are covered more systematically in "Dynamic Phenomena and Concepts", in Sect. 10.3.2.

Viewing the system as autonomous, we consider what goes on in the system strictly as we observe it: time as it passes, the system component valves and the flow of water. The valves open or close, and water is either being replenished or taken away. The opening and closing, and hence the supply (and possible overflow) or withdrawal of water happens without our being able to exert any influence. We cannot influence the system.

When we view it as reactive we "open up" the system, meaning — here — that we allow ourselves to change the values of some state components: Now we are allowed to set, to control, the valves. We can open and close them. We can influence the system. This view, the dynamic reactive domain attribute view, is what we shall have in mind in the following.

We model a water tank system where time has been discretised. Time units are assumed "small", sufficient to ensure some "appearance" of continuity. The water tank system has a context, state and an environment. The context consists of those system component values which "never" change, i.e., which are static. The state consists of those system component values which may change, i.e., which are dynamic. The environment consists here of just those component values which may influence the values of the system state. Here precipitation (viz.: rainfall) and evaporation. So the domain which we model consists of the water tank system and the precipitation and evaporation phenomena of the "surrounding" weather.

Formal Presentation: Water Tank System

type
  Height, Area, Supply, Withdraw, Time, Time_Interval
  Context :: Height × Area × Supply × Withdraw
  Σ = Time $\overrightarrow{m}$ State
  State = Valve × Height × Valve
  Valve == closed | open
  P,E /* Precipitation, Evaporation */
  ENV = Time $\overrightarrow{m}$ (P × E)
value
  H:Height, A:Area, s:Supply, w:Withdraw, Δ:Time_Interval
axiom

mk_Context(H,A,s,w):Context $\wedge$ Δ=1 $\wedge$
$\forall$ σ:Σ,ρ:ENV •
  dom σ = { min **dom** σ .. max **dom** σ } $\wedge$
  $\forall$ t,t′:Time • t $\in$ **dom** σ $\wedge$ t′=t+Δ $\Rightarrow$
    let ((iv,h,ov),(iv′,h′,ov′)) = (σ(t),sigma(t′)) **in**
    0 $\leq$ h $\leq$ H $\wedge$
    (iv=open $\wedge$ ov=closed $\Rightarrow$ h>0) $\wedge$
    (ov=open $\wedge$ iv=closed $\Rightarrow$ h<H) $\wedge$
    let (π,ϵ) = ρ(t) **in**
    h′ = **case** (iv,ov) **of**
        (open,open) $\rightarrow$ h + s − w, (open,close) $\rightarrow$ h + s,
        (close,open) $\rightarrow$ s, (close,close) $\rightarrow$ 0 **end** + π + ϵ
    end end

From modelling a domain we proceed via informally stated requirements to model a software design. We do so in order to further capture the consequences of a domain attribute being reactive. Figure 10.2 is intended to abstract the flow of information and control in such a controller.

Given an initial state of valves and height values, a controller programmed (i.e., script) opening and closing of the input and output valves, and assuming no precipitation (rain, into) and no evaporation from the tank, one can make the height behave according to some curve. Allowing for precipitation and evaporation we must regularly sample the height. This sampling is a way of modelling the precipitation and evaporation. The sampling and its use by the controller algorithm reflects an assumption about the domain, namely the one expressed in the last nine lines of the axioms above.

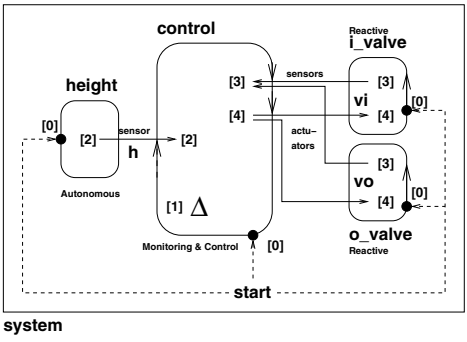


Fig. 10.2. A water tank: a process model

Now, depending on the height, open and/or close respective valves. We therefore model the water tank as a set of concurrent processes: the two valves, and the height. A fourth process models a required controller.

We refer to Fig. 10.2. It shows, as four rounded corner boxes, four processes. Tracing the box outlines, in the directions of their arrows, shall designate process behaviours. Thus the processes are here all seen as cyclic. Imagine a token passing around the box outline. At any point it designates a process point. The four fat dots designate process starting points. The arrows that connect two processes designate channels, and the direction, indicated by the arrow, designates which process outputs and which process inputs (arrowhead) a message. When two processes have reached the program points where one channel intersects respective box outlines, then a rendezvous between the two processes can take place: They are synchronised and can communicate a value. The Δ designates a Δ time unit delay. Bracketed numerals ([0], [1], [2], [3] and [4]) are program point labels and refer to program points in the formal text.

We comment on the *software design* model given below. There we use an imperative version of the RSL process concept. Time is — rather crudely — modelled as a variable of the *system* process. The valve settings are modelled as variables of respective *valve* processes. Valve sensors are modelled as the reading (riv?,rov? [1,2]) of valve settings. Valve actuators are modelled as the writing (wiv!,wov! [3]) of valve settings. The height sensor is modelled as a further unexplained reading (rh? [1]) from the autonomous *height* process. The regular sensing and actuation at Δ time intervals is modelled by a **wait** Δ time period extension to RSL [155, 379].

Formal Presentation: Software Design Specification: Water Tank Controller

```
type
   Curve_Script = Time ~m~> Height
axiom
   ∀ cs:Curve_Script,t:Time • t ∈ dom cs ⇒ t+Δ ∈ dom cs
channel
   rh:Height, riv,rov,wiv,wov:Valve
value
   s:Supply, w:Withdraw, H:Height

   system: Valve×Valve×Curve_Script×Time → Unit
   system(ioc,ooc,cs,it) ≡
      i_valve(ioc)||o_valve(ooc)||height()||control(cs,it)
      pre: it ∈ dom cs

   i_valve: Valve → in wiv out riv Unit
   i_valve(ioc) ≡ [0] variable vi:Valve := ioc;
      while true do [3] riv!vi; [4] vi := wiv? end
```

```
o_valve: Valve → in wov out rov Unit
o_valve(ooc) ≡ [0] variable vo:Valve := ooc;
   while true do [3] rov!vo; [4] vo := wov? end

control: Curve_Script × Time → in rhm,rivm,rov out wiv,wov Unit
control(cs,it) ≡ [0] variable tv:Time := it;
   while true do
         wait Δ;
         let (i,o) = calc((s,w,H),cs,(tv,[1]rh?,[2]riv?,[2]rov?)) in
         [3] (wiv!i||wov!o) end; tv := tv+Δ
   end

calc: (Supply×Withdraw×Height)×Curve_Script×
         (Time×Height×Valve×Valve) → Valve×Valve
```

This ends our multiple attribute (1–3) and universe of discourse (4–5) example of illustrating (1) inert, (2) autonomous and (3) reactive (4) domain and (5) software design attributes. ∎

Methodological Consequences

We summarise:

Principles. *Hybridicitiess:* When a domain phenomenon must be understood in terms of interactions between continuous and discrete phenomena (i.e., behaviours), then it should be described as some such behavioural (process-oriented) system relying on continuous functions and discrete maps. ∎

Techniques. Deploy the *hybridicity* technique, as implied by the formulation of the above principle, of describing the system as composed from (i.e., some Cartesian of) continuous functions and discrete maps, and as a set of interacting (for example, synchronising and communicating) processes. ∎

For proper theories and design techniques for reactive systems we refer to [58, 207, 228–230].

10.2.4 Chaos

Characterisation. *Chaotic Phenomena:* A domain phenomenon is said to possess the chaotic attribute if a suitable, perhaps even a "best" abstract model describes it as a partial function from some dense point set type A to some singleton (or larger, possibly infinite) sets of values of type B:

type
A, B
Phenomenon = A $\xrightarrow{\sim}$ B-**infset**

The interpretation of the infinite set of result values is that only one is chosen, but which one is not a priori decidable, i.e., it reflects the chaotic nature of the phenomenon. ∎

Example 10.4 *Air Traffic:* A final use of the air traffic example may be in order. The world never behaves as we wish it would: radar is not perfect (i.e., ideal). Meteorological phenomena, as well as malfunctioning aircraft responders, including aircraft disasters, may force us, depending on the problem we are to solve (viz.: an air traffic disaster monitoring and control system), to consider — and hence model — air traffic chaotically.

Formal Presentation: Air Traffic

type
FPs = F $\overrightarrow{m}$ P
cFPs = F $\overrightarrow{m}$ P-**infset**

rAT′ = T → (AS × FPs), rAT = {| rat:rAT′ • wf_rAT(rat) |}
dAT′ = T $\overrightarrow{m}$ (AS × FPs), dAT = {| dat:cAT′ • wf_dAT(dat) |}
cAT′ = T $\overrightarrow{m}$ (AS-**infset** × cFPs), cAT = {| cat:cAT′ • wf_cAT(cat) |}

value
wf_cAT ≡ wf_AT
wf_dAT: dAT′ → **Bool**
wf_dAT(dat) ≡ /∗ a simple variant of wf_AT ∗/
radar: rAT → cAT

In a chaotic air traffic domain what are thought of as "moving" flights, may not "move" flights continuously, but "erratically", even forward and backward. Ghost flights may seem to occur — hence aircraft may appear to be outside the airspace (for a while, only to reappear). The position of an aircraft is "blurred": appears to be nondeterminate, and the airspace may itself not be precisely determined — hence the use of partial functionality and infinite sets of air spaces and positions. ∎

Methodological Consequences

We summarise:

Principle. *Chaos:* When a domain phenomenon exhibits the kind of behaviour that is here described as chaotic, then great care must be taken to

identify (i.e., to isolate) which subphenomena are discrete, which are continuous and which really possess the chaotic behaviour. ▪

Techniques. Deploy the *chaos* technique, as implied by the formulation of the above principle, of describing the system as a suitable composition of non-chaotic behaviours with a suitably identified (i.e., isolated) chaotic behaviour. The latter is then, typically, modelled by looseness and internal nondeterministic specification techniques. ▪

10.2.5 Discussion

We have briefly discussed the four concepts of continuity, discreteness, hybridicity and chaos. The discreteness concept can be modelled simply within the specification notation (RSL [117,118]) primarily used in our software engineering methodology papers. To fully satisfactorily model the continuity, the hybridicity and the chaos concepts we need to combine the primary specification notation (whether B [4], RSL, VDM-SL [35,36,104], Z [162,341,343,377] or other) with a (perhaps less "formal", but equally reasonable) notation of classical mathematical calculus. One needs to deploy such notions as limits of approximations, etc., in order to properly handle such indispensable notions as, for example, differentiable and integrable.

10.3 Statics and Dynamics

Static and dynamic attributes can be thought of as pertaining to such phenomena which we may choose, typically, to model as contextual or state components of configurations. We covered the concepts of configurations, contexts and states in Vol. 2, Chap. 4. There we used the terms static contexts and dynamic states. That is, a configuration component, or for that matter any information entity, which never changes value is a static one. If the entity does change value it is a dynamic one.

We also commented that such phenomena, when analysed with respect to their temporal properties (or absence thereof), were more or less contextual, or more or less state-oriented. That is, we speak of a spectrum from (i) hard via (ii) soft to (iii) hard again properties: (i) very static, (ii) somewhat static, but not completely static, and so on, (iii) to very dynamic. Thus the concepts of static and dynamic attributes is certainly one that is associated with time: either the actual time of a domain, or the time experienced during computations. Another way of paraphrasing the above is as follows. Let C be the type of supposedly statically attributed information. Many functions are usually applicable to such information. Among these there will typically not be functions, f, whose signatures are of the form:

type
  C
value
  f: ... × C × ... → ... × C × ...
example:
  f(c) **as** (...,c′,...)
  pre ..., **post** ...

If there was such a function type, then that, to us, could mean either of two things: (1) After the computation of c′ the value c will not be referred to anymore. (2) After the computation of c′ the value c will continue to be referred to, independently of the value c′. In the former case (1) we say that C is possibly of dynamic type. In the latter case (2) we say that C is possibly of static type. Note that we hesitate. There is more to the story — and the rest of this section will reveal more of that story.

10.3.1 Static Phenomena and Concepts

Static Attributes

Very few phenomena in this world are statically attributed. One may think that an entity is of static nature, but then, when scrutinised more closely it is (also) of dynamic nature.

Example 10.5 *Statically Attributed Domain Phenomena:* We list examples of statically attributed domain phenomena. Each example is conditioned by the view in which it is considered to be of static type. (i) A train timetable during season. (ii) A railway net, as seen from the point of view of a passenger, and during the ride of a particularly train, i.e., topologically. (iii) A job application form seen as a template (a type) for all job applications. (iv) A computer program. Please compare the above list to that of Example 10.7. ▪

So why have we introduced the idea? Because, basically, it is a convenient way, for some modelling activities, to consider abstract phenomena as being statically attributed: The lines and stations remain fixed.

Characterisation. *Static Domain Entity:* An individual of a domain entity is static if it is not subject to actions that change its value. ▪

Individuals and phenomena of a static domain have no time dimension. They are not influenced by events; nothing happens, nothing changes.[1] Static domain individuals may be referred to in these volumes as constants. They may eventually, in our software (i.e., in our computing system), entail large composite data structures or database files. Viewed in the context of the creation (initialisation) of these entities they briefly become dynamic individuals.

[1] We refer to the entry on Domain Characteristics, on page 67, fourth paragraph, in [189].

Example 10.6 *Static Books:* By a book we here mean the structured, syntactic collection of pages with printed lines etc., where the structuring is wrt. parts, chapters, sections, subsections, etc. Module (part, chapter, ...) names also "carry with them" an ordering. Such seeming "events" as being sold, taken up for reading, turning a page and being laid aside, do not, per se, change the "state" of the book. Its chapter, section, etc., structure and the printed words remain unchanged. Functions performable on a book could be: display book, part, chapter, section, subsection, item, etc.; show page, line, etc.

Formal Presentation: Static Books, I

```
type
   SBook′ = Text × Modu
   Modu = Txt × (Mn × Modu)*
   Txt = (Word | Itms)*
   Itms = (Entry × Txt)*
   Mn = Pn | Cn | Sn | Ssn | ... /* hierarchy */
   Pn, Cn, Sn, Ssn, ..., In
   SBook = {| (text,modu):SBook′ • wf_Modu(modu) |}
   PBook = Text × Pagelist
   Format
   Pagelist′ = Line*
   Line = (Mn | In | Word)*
   In = L1_In | L2_In | ... | Lm_In
   Pagelist = {| pgl:Pagelist′ • wf_Pagelist(pgl) |}
value
   obs_format: PBook → Format
   wf_Modu: Modu → Bool
   wf_Modu(mod) ≡
      /* appropriate hierarchical structuring and */
      /* nesting of uniquely identified modules, etc. */
   wf_Pagelist: Pagelist → Bool
   wf_Pagelist(pgl) ≡
      /* First page starts with text, and goes on with text */
      /* of first of possibly several outermost modules, etc. */
```

So far we have formalised, rather concretely, the static domain "main type of interest": books. Now we look at some "interesting" functions and their properties.

Formal Presentation: Static Books, II

```
value
   s_p_Book: SBook × Format → PBook
```

```
    p_s_Book: PBook → SBook
axiom
    ∀ pb:PBook • pb = s_p_Book(p_s_Book(pb),obs_format(pb)),
    ∀ sb:SBook, f:Format • sb = p_s_Book(s_p_Book(sb,f)),
    ∀ pb:PBook, ∃ f:Format • f = obs_format(pb)
value
    s_display: Mn* × SBook ~→ Modu
    s_display: Mn* × In* × SBook ~→ Txt
    p_display: Nat × PBook ~→ Page
    p_display: Nat × Nat × PBook ~→ Line
```

We model two sets of stakeholder perspectives: those of authors and readers, the structured book, and those of typographers, typesetters, book-printers, and publishers. We define functions which convert between the two perspectives — introducing for that purpose a notion of formatting. The display (browse, read) functions apply to books but do not alter them. ∎

Techniques. *Static Attribute:* Describe in terms of abstract or concrete, carefully narrated and formalised types with an associated number of observer functions, but no need for generator functions. ∎

Principles. *Static entities* form part of a context, not state. Always identify such entities. Model accordingly. ∎

10.3.2 Dynamic Phenomena and Concepts

We review the statement made above: *One may think that an entity is of static nature, but then, when it is more closely scrutinised it is (also) of dynamic nature.* An example may serve to illuminate the point being made.

Example 10.7 *Dynamically Attributed Domain Phenomena:* We list examples of dynamically attributed domain phenomena. Each example is conditioned by the view in which it is considered to be of dynamic type. Please observe that the examples relate strongly to those of Example 10.5. (i) A train timetable under construction. (ii) A railway net, as seen from the point of view of a train engineer, and during the ride of a particularly train: The switch units, signals and rail unit states in general are changing during the train ride. (iii) A job application form while being filled out. (iv) A computation based on, i.e., as prescribed by, some computer program. ∎

Characterisation. *Dynamic Phenomena, I:* Phenomena for which time is an explicit part, that is, phenomena whose values change with time, possess the dynamic attribute. ∎

Another, perhaps better way of characterising dynamic phenomena is given next.

Characterisation. *Dynamic Phenomena, II:* An individual of a domain entity is dynamic if it is subject to events and changes accordingly. ▪

How time enters abstract models of such phenomena is the subject of this section. Michael Jackson [189] suggests three kinds of dynamic attributed phenomena: inert, active and reactive. For inert phenomena time need not enter the model explicitly. For active and reactive phenomena time enters in various ways — as we shall see.

Inert Phenomena and Concepts

We present a characterisation, show one or more examples, and formulate modelling principles and techniques.

Characterisation. A phenomenon is said to be of, or possess, the *dynamic inert attribute* if it never changes value of its own volition. An inert phenomenon only changes value as the result of external stimuli. These stimuli prescribe exactly which new value they are to change to. ▪

Example 10.8 *Train Timetable:* From the perspective of a railway administration, a train timetable is [of, or has] inert [dynamic attribute] since it only changes value when so directed by seasonal train traffic schedulers. ▪

Example 10.9 *Road Net:* From the perspective of a region's road authorities the road topology is [of the] inert [dynamic attribute] since it only changes values when so directed by that, or those authorities — typically in connection with road maintenance, closing old (no longer to be used) roads, opening new roads or in connection with larger traffic accidents, festivities, crowd control measures or other. ▪

In the above we have considered phenomena in isolation from other phenomena. The former were shown to illustrate inert phenomena, and the latter to represent such "forces" which, or who, could change the values of these inert phenomena.

Principle. *Inert Dynamic Phenomena:* When analysing a phenomenon with respect to it being of a static or a dynamic attribute one must state very precisely from which scope one views the phenomenon. If it is deemed inert, then its scope must not include the "agent" who (or which) effects its value changes. ▪

When the scope includes the agents which effect value changes then the phenomenon cannot be inert.

Techniques. To model an *inert phenomenon* one must describe its values (d), their types (D) and the functions — including the value-changing operations — (g) that apply to these values:

type
  A, D
value
  g: A × D → D

Here A is the type of the auxiliary values that are needed when changing the value of an inert phenomenon. ∎

Typically, for large-scale phenomena one typically can implement the inert phenomena in terms of a query/update database. When a phenomenon is judged inert then one does not have to sample the state in order to see whether the inert phenomena has changed value and then update the database image of that phenomenon.[2]

Example 10.10 *Timetabling:* In a timetabling system we create timetables. A timetable is here considered a set of uniquely train name identified (Tn) and train characteristics annotated (TAnn) train journey descriptions (TJ). A train characteristics annotation (TAnn) describes the train in terms of one or more train categorisations (Cat): optional first (`first`) and always second (`second`) class descriptions; of descriptions of optional meal (restaurant (`rest`)) and refreshment (cafeteria (`cafe`), bar (`bar`)); and descriptions of special train (handicap (`handi`), young parent (`baby`), etc.) facilities. A train journey (TJ) description is a sequence of station visit (TV) descriptions. A station visit description consists of four parts: arrival time (T), platform number (Pno), station name (Sna) and departure time (T).

Formal Presentation: Timetabling, I

type
  TT = Tn $\overrightarrow{m}$ TAnn × TJ
  TAnn = Cat-**set**
  Cat == first | second | rest | cafe | bar | handi | baby | ...
  TJ = TV*
  TV = T × Pno × Sna × T

[2] We bring in this paragraph for two reasons: (i) Among the domains, for which we investigate Michael Jackson's concept of static and dynamic attributes, are the domains also of requirements engineering and of software design. Hence a technique must be advised for the modelling (including "coding") of such phenomena. (ii) Even if the domain had been the universe of discourse of this chapter, that is, typically a non-IT-centred application domain, the consequences, in final software, of being of inert attribute are here deemed worth commenting upon.

The view that we shall take here is of timetables as inert dynamic entities.

Hence we focus only on the operations that create and change (update) train timetables: *Create* an empty timetable of no train journey descriptions. *Insert* a whole properly identified train journey description. *Remove* a properly identified train journey description. *Delete* a properly named station visit from a properly identified train journey description. *Update* train arrival and departure times of a properly named station visit of a properly identified train journey description. *Change* platform number identification of a properly named station visit of a properly identified train journey description.

Throughout we assume that certain, mostly obvious, well-formedness constraints that hold of train timetables also remain in effect as a result of any operation.

Formal Presentation: Timetabling, II

value
creaTT: **Unit** → TT
insertTJ: Tn × TJ → TT $\xrightarrow{\sim}$ TT
remTJ: Tn → TT $\xrightarrow{\sim}$ TT
delSV: Tn × Sna → TT $\xrightarrow{\sim}$ TT
updSVTT: Tn × Sna × (T × T) → TT $\xrightarrow{\sim}$ TT
chgSVPn: Tn × Sna × Pn → TT $\xrightarrow{\sim}$ TT
...

■

Active Dynamic Attribute

We present a characterisation, show one or more examples, and formulate modelling principles and techniques.

Inert dynamic phenomena are said to not be active. Nothing "happens within", i.e., "inside" the phenomenon. It seems — it is — inactive.

Characterisation. A phenomenon is said to be of, or possess, the *dynamic active* attribute if it changes value (also) of its own volition. ■

The above characterisation does not preclude dynamic active phenomena to change value due to 'external stimuli'. Here 'external stimuli' means something 'input' from without (i.e., from "outside") the phenomena.

Example 10.11 *Active Dynamic Phenomena:* We give a simple variety of examples of dynamic active phenomena: (a) the global weather system, (b)

air traffic control systems and (c) simple administrative data processing (i.e., computer) systems. ▪

Michael Jackson [189] suggests three kinds of dynamic active attributed phenomena: (a) autonomous, (b) biddable, and (c) programmable. The three subexamples given in Example 10.11 illustrate, as we shall see, respective kinds of dynamic active phenomena. We shall, in the following, adopt Michael Jackson's theory of dynamic active attributed phenomena, while making minor reinterpretations of the dynamic active attribute concepts. And, as something new, we shall provide formal examples, and development principles and techniques.

Autonomous Active Dynamic Attribute:

Characterisation. We characterise the *autonomous active dynamic* attribute in two ways: (i) A phenomenon is said to be of, or possess, the autonomous active dynamic attribute if it changes value only of its own volition — that is, it cannot also change value as a result of external stimuli; (ii) or when its actions cannot be controlled in any way. That is, they are a "law onto themselves and their surroundings". ▪

For dynamic active autonomous phenomena we do not foresee means of applying external stimuli!

Example 10.12 *Varieties of Autonomous Phenomena:* We give some informal examples: (i) The celestial system will usually be modelled as an autonomous system following the celestial mechanics laws of Nicolaus Copernicus, Johannes Kepler, Isaac Newton and others. Our models then can be analytical. (ii) The (for example, global) weather will usually be modelled as an autonomous system — for which we may find, or decide, that we have no applicable laws — and hence we must model weather observations. Our models are therefore a posteriori statistical. (iii) The flow of time will usually be modelled as an autonomous system. But we cannot stop time, or can we? ▪

When we model simulated (i.e., simulation) time, then it is not as an autonomous system, Rather it is, as we shall see, a programmable, active dynamic phenomenon.

Principle. *Autonomicity:* (i) Only some very basic phenomena of nature seem to be autonomous active dynamic. (i) When modelling a supposedly autonomous phenomenon, an analysis has to be performed in order to determine whether the (assumed) autonomous behaviour is predictable. That is, it follows some laws (of nature or other), in which case these laws have to be modelled when it is not predictable; we must determine whether the

model must include some observation functions, in which case these observation functions have to be given a signature. ▪

Techniques. *Autonomicity:* Besides abstract or concrete type definitions (informal narrative or formal specifications) of the autonomous value classes, informally describing and formally modelling autonomicity is basically "restricted" to the postulation of observers and invariants: At least stating their signatures, and, as far as is possible, expressing some axioms over these functions. In addition it might be advisable to informally describe and formally model the autonomous active "main type of interest" as a space of time--varying functions. Autonomy is now expressed in terms of a refutable assertion, an axiom which expresses that two values of the active, autonomous "main type of interest" which cover overlapping time intervals "coincide" on these intervals, i.e., have the same function values. ▪

Example 10.13 *Weather Forecast:* We let the formulas "speak for themselves", cf. the /* comments */:

Formal Presentation: Weather Forecast, I

```
type
   G  /* Global Weather */
   T  /* Times */
   P  /* Geographic Positions */
   A  /* Atmospheric Measurements */
   /* Actual Weather: */
   Wa′ = T ~m~> (P → A-set)
   Wa = {| wa:Wa′ • wf_Wa(wa) |}
   /* Weather Forecast: */
   Wf′ = T ~m~> (P ~m~> A-set)
   Wf = {| wf:Wf′ • wf_Wf(wf) |}
value
   obs_Wa: (T×T) × P-set → Wa
   wf_Wa: Wa → Bool
   wf_Wf: Wf → Bool
      /* well-formedness predicates */
      /* express continuity properties */
```

The global weather is observed over a certain time interval and at specified positions.

Formal Presentation: Weather Forecast, II

```
axiom
   ∀ wa,wa′:Wa • dom wa ∩ dom wa′ ≠ {} ⇒
```

```
      wa/dom wa ∩ dom wa′ = wa′/dom wa ∩ dom wa′
value
   forecast: (Ta×Ta) × (Tf×Tf) × P-set → Wa ~→ Wf
   forecast((ta,ta′),(tf,tf′),ps)(wa) as wf
      pre {ta,ta′} ⊆ dom wa ∧  ta < ta′ < tf < tf′ ∧ ...
      post {tf,tf′} ⊆ dom wf ∧ ...
```

The axiom says that two weather observations of overlapping periods "coincide" in the common period. The (nondeterministic) forecast function specifies that a forecast is to be made for a forecast period (tf,tf′) based on actual weather observations in some past (preceding) period (ta,ta′). Note that the result of a forecast is not weather, but a weather forecast.

The sets of atmospheric measurements, A-set, can be combinations of wind W, degrees Celsius C, humidity H, etc. The obs_Wa function can be "implemented" by separate observer functions obs_W, obs_C, and obs_H, with results being averaged over time intervals.

Formal Presentation: Weather Forecast, III

```
type
   W /∗ wind ∗/
   C /∗ temp., Celsius ∗/
   H /∗ humidity ∗/, ...
   P /∗ geographical position, sea level ∗/
value
   obs_W: P → G → W
   obs_C: P → G → C
   obs_H: P → G → H, ...
```

This example illustrates the strong dependency on observer (i.e., classification) functions. ∎

Biddable Active Dynamic Attribute

We shall find that many, if not most, of the kinds of domains for which we shall be developing software support are of the biddable nature. They can be directed to perform some actions, but someone needs to monitor that such actions are indeed performed.

Characterisation. *Biddable Active Dynamics:* An active dynamic individual is, and entities of a domain for a subsystem are, biddable if it (they) can be advised (through a contractual arrangement) on which actions are expected of it (or of them) in various states. A biddable individual (etc.) does not have

to take these actions — but then the contractual arrangement need no longer be honoured by other individuals (other subdomains) with which it interacts (i.e., shares phenomena)[3]. ▪

Example 10.14 *Aircraft:* In the larger domain of air traffic control, a controller of a ground air traffic control center can bid, i.e., advise, an aircraft captain to follow a certain route, but the air flight captain has the final word and may choose another flight course. The aircraft pilot is biddable. ▪

More generally:

Example 10.15 *Other Vehicles:* Ship captains, train engineers and automobile drivers are biddable individuals: they know the traffic rules, on the high seas, on the rails and on the roads; they hear what sea pilots tell them, respectively see the track- or roadside traffic signals, but there is no guarantee that they obey them. Most of what we do in our daily work when we follow 'set (production, service or office) procedures' is biddable: We may honestly endeavour to carry out our work as expected, as prescribed, but we are all humans. Some are prone to make more work action mistakes than others, and a few are derelict or even outright malicious in their dispatch of duty. ▪

But not just people may fail. Not all equipment is 100% dependable (reliable or fault free). Requests for acknowledgement from, or commands for action to, such equipment may fail in being completed. We shall, in Sections 11.4, 11.6, and 11.8 further treat the concept of biddability, now in the context of modelling the domain facets of support technologies, rules and regulations and human behaviours, respectively. Many human behaviour modelling issues stem from their being biddable domains.

Example 10.16 *Social Welfare:* We describe two of the stakeholder groups: clients and (some subgroup of) social workers. Clients are, upon request, interviewed by social workers. The social worker decides, based upon social welfare rules and regulations, to offer the client certain social welfare benefits. The client may or may not accept these. Both the social worker and the client are biddable.

Formal Presentation: Social Welfare

type
  Client, SocWrk, Status, RulReg, Dossier
value
  interview: Client×SocWrk×RulReg×Doss $\xrightarrow{\sim}$ Status
  interview(c,w,r,d) ≡

[3] We refer to the entry on Domain Characteristics, on page 70, second paragraph, in [189].

```
    i(c,r,d) ∏ i′(c,r) ∏ i″(c,d) ∏ i‴(c) ∏ i⁗() pre: ...
  benefit: Status×SocWrk×RulReg×Doss ~→ Offer
  benefit(s,w,r,d) ≡
    b(c,r,d) ∏ b′(c,r) ∏ b″(c,d) ∏ b‴(c) ∏ b⁗() pre: ...
  accept: Offer×Client ~→ Bool
  accept(o,c) ≡ true ∏ false
```

We have not bothered to give signatures for the nondeterministically internally chosen subsidiary interview, respectively benefit functions. (The reader can do that easily.)

The narrative text that goes with these functions is: The interviewing social worker shall (i.e., is bid to) follow rules and regulations and to determine status based also on the client's past cases as recorded in the client dossier. In reality the social worker may fail in properly taking all — or, for that matter, any — facts into account. Similarly for the benefit "calculation" procedure. The client is expected (i.e., is bid) to accept the benefit offer. ▪

Example 10.17 *E-Market Buyer/Seller Protocols:* Buyers inquire about and order merchandise, accept or reject delivered such, pay invoices and possibly return faulty merchandise. Sellers quote price and delivery conditions, confirm and deliver orders, invoice delivered merchandise and refund returned (and paid) merchandise.

Formal Presentation: E-Market Buyer/Seller Protocols, I

```
type
  Merch, Money, ...
  BMsg = BuyAct | Merch | Money | ...
  SMsg = SelAct | Merch | Money | ...
  BuyAct = Inq | Ord | Acc | Rej | Pay | Ret
  SelAct = Quo | Con | Del | Inv | Ref
  ...
  Del == mkDel(m:Merch)
  ...
```

We do not concretise the individual action commands other than indicating, as an example, that of the seller delivery action.

Buyers may act arbitrarily. They may order without prior inquiry, pay for invoiced merchandise multiple times, etc. Sellers may act likewise: they may quote without prior inquiry, confirm and/or deliver without prior order, etc. Inquiries, orders, rejects and returns represent buyer bids: Please respond.

Deliveries and invoices represent seller bids: Please respond. And the absence of buyer bids "translates" into seller bids: namely "do nothing".

Formal Presentation: E-Market Buyer/Seller Protocols, II

```
type
  BΣ, SΣ
channel
  bs:BuyAct, sb:SelAct
value
  bσ:B`Sgma, sσ:SΣ
  emkt: BΣ × SΣ → Unit
  emkt(bσ,sσ) ≡ buyer(bσ) || seller(sσ)
  buyer: BΣ → out bs in sb Unit
  seller: SΣ → in bs out sb Unit
```

Our model is grossly simplified: The e-market consists of one buyer and one seller. That is sufficient to illustrate a nature of biddable domain phenomena.

Formal Presentation: E-Market Buyer/Seller Protocols, III

```
value
  buyer(bσ) ≡
    buyer(
      let (msg,bσ′)=b_choice(bσ) in bs!msg;bσ′ end
      ⌈⌉
      let msg=sb? in bσ⊕msg end)

  b_choice: BΣ ~→ BuyAct × BΣ
  ⊕: BΣ × BuyAct → BΣ


  seller(sσ) ≡
    seller(
      let (msg,sσ′)=s_choice(sσ) in sb!msg;sσ′ end
      ⌈⌉
      let msg=bs? in sσ⊕msg end)

  s_choice: SΣ ~→ SelAct × SΣ
  ⊕: SΣ × SelAct → SΣ
```

The choice functions, through their predicates, exhibit nondeterministic behaviour.

```
  b_choice(bσ) ≡
```

```
    let msg = C_b(msg,bσ) in (bσ⊕msg,msg) end
  s_choice(sσ) ≡
    let msg = C_s(msg,sσ) in (sσ⊕msg,msg) end
```

The above model illustrates the arbitrary human behaviour (i.e., diverging from bid behaviour) through the use of internal nondeterminism. Nondeterminism is expressed explicitly through the $\sqcap$, and implicitly through any (buyer, respectively seller) action message (msg) satisfying some predicate $\mathcal{C}_b$, respectively $\mathcal{C}_s$:

Formal Presentation: E-Market Buyer/Seller Protocols, IV

```
type
  B == inq | ord | acc | rej | pay | ret
  S == quo | con | del | inv | ref
value
  C_b: BΣ → BuyAct
  C_b(msg,bσ) ≡
    let m = inq ⌈⌉ ord ⌈⌉ acc ⌈⌉ rej ⌈⌉ pay ⌈⌉ ret in
    case m of
      inq → let i:Inq • P_b(i)(bσ) in i end
      ord → let o:Ord • P_b(o)(bσ) in o end
      ...
      ret → let r:Ret • P_b(r)(bσ) in r end
    end end
  C_s: SΣ → SelAct
  C_s(msg,sσ) ≡
    let m = quo ⌈⌉ con ⌈⌉ del ⌈⌉ inv ⌈⌉ ref in
    case m of
      quo → let q:Quo • P_c(q)(sσ) in q end
      con → let c:Con • P_c(c)(sσ) in c end
      ...
      ref → let r:Ref • P_c(r)(sσ) in r end
    end end
```

If, for example, the buyer state records (through $\oplus$) some invoice, then a payment relevant to that invoice may be made, or some payment to an erroneous one will be made.

Formal Presentation: E-Market Buyer/Seller Protocols, V

```
value
```

$\mathcal{P}_b$: BuyAct $\to$ BΣ $\to$ **Bool**
$\mathcal{P}_s$: SelAct $\to$ SΣ $\to$ **Bool**

Or, if, for example, the seller state records (through $\oplus$) some inquiry, then a quote relevant to that inquiry may be made, or some unsolicited quote will be made.

▪

Techniques. *Biddability:* When modelling many application domain observer and other functions and many operations (which change states of interesting types possessing biddable attributes), we shall find that we can basically just give (i.e., informally and formally describe) the signatures of these predicates, functions and operations. In general we cannot fully define their values (their effects) as they pertain to biddable phenomena and these may fail to honour expectations. One might model the biddable phenomenon as a set of negotiating processes: The thing which is biddable is sent inquiries or commands. The thing which bids is sending these requests. A protocol of negotiation between these two parties, the bidder and the bidded, is defined. This is a protocol which may describe what happens if bids are not obeyed if that is a property of the domain, or if it is a mandate of the (business process reengineering) requirements.

Central to the modelling of biddability is the use of internal nondeterminism ($\prod$). In general, functions expressing biddability are partial:

value
f: A $\xrightarrow{\sim}$ B-**infset**

and may designate indefinite result values. ▪

Principle. *Biddable Active Dynamics:* Consider domain phenomena that involve humans, and/or support technologies to basically be biddable. ▪

Programmable Dynamic Active Attribute

As the adjective programmable hints at, the notion of programmability[4] is, of course, close to our general subject of computing. But please do not be misled by that adjective.

Characterisation. *Programmable Active Dynamics:* An active dynamic phenomenon has the programmable active dynamic attribute if its actions over a future time interval can be accurately prescribed. ▪

[4] We refer to the entry on Domain Characteristics, on page 70, third paragraph, in [189].

Computing and communications platforms are programmable. Since computer equipment also form an important part of domains before we have introduced additional such equipment, we need to also take such individuals into account when creating domain descriptions.

Example 10.18 *A Simple Computer:* When developing a compiler for some programming language such as, for example, Java [11,17,125,221,328,370], C# [161,241,242,274] or the like, some compiler requirements may state that the compiler is to generate code for some specific hardware computer. Hence we must consider specific hardware computer to "belong to the domain", and therefore it must be described. For simplicity we postulate a very simple hardware computer: The computer state entities are: The computer has a data store of 2^n word (n bit) cells addressed (linearly) from address (natural number) 0 to and including address 2^{n-1}. The computer has a code store of 2^m word (q bit) cells addressed (linearly) from address (natural number) 0 to and including address 2^{m-1}, where q is sufficiently larger than n. The computer has an n-bit accumulator, and a q-bit code address register.

Formal Presentation: A Simple Computer, I

```
value
  n:Nat, m:Nat
type
  Σ = Acc × Ins × Data × Code
  Acc = Word
  Ins = Ptr
  Data′ = Addr →m Word
  Addr = Nat
  Data = {| d:Data′ • dom d = {0..2^(n-1)} |}
  Code′ = Ptr →m Ins
  Ptr = Nat
  Code = {| c:Code′ • dom c = {0..2^(m-1)} |}
```

The computer code concept is as follows: At any one time the computer is executing according to a program stored in its code store. A computer program consists of a linear sequence of at most m instructions. (Each instruction can be assumed to fit in q bits.)

We now list and narrate a number of computer instructions. Store instructions prescribe the copying of the number stored in data store at address a to the accumulator, thus replacing its former content, and a is part of the instruction. Load instructions prescribe the copying accumulator content into the data store cell at address a, thus replacing the former content of the accumulator, a is part of the instruction. Add instructions prescribe the addition of the number stored in data store at address a to the accumulator, and a is

part of the instruction. Subtract instructions prescribe the subtraction of the number stored in data store at address a from the accumulator, a part of the instruction. And so on for integer multiply and integer division instructions. Explicit jump instructions replace the content of the code address register with an address b, where b is part of the instruction. Conditional jump instructions replace the content of the code address register with an address b, only if the content of the accumulator is all zeroes: $00\cdots000$, b is part of the instruction. Stop instructions halt execution of the computer. *There are other instructions.*

Formal Presentation: A Simple Computer, II

Ins = Sto | Loa | Add | Sub | ... | Jmp | CJp | Stp | ...
Sto == Addr
Loa == Addr
Add == Addr
Sub == Addr
...
Jmp == Ptr
CJp == Ptr
Stp == nil
...

Some external stimuli, not described here, set the initial contents of the code address register. Execution, at any time, proceeds with the instruction, of the code store designated by the code address register. We simplify our treatment by omitting consideration of more realistic instructions, external interrupts, including operator panel intervention, and input/output.

Formal Presentation: A Simple Computer, III

M: $\Sigma \xrightarrow{\sim} \Sigma$
M(α,ι,δ,γ) $\equiv$
  if $\iota = 2^m$
    then (α,ι,δ,γ)
    else
      let ins = $\gamma(\iota)$ **in**
      let σ = **case** ins **of**
          mk_Sto(a) $\rightarrow$ (α,ι+1,$\delta\dagger$[a$\mapsto\alpha$],γ),
          mk_Loa(a) $\rightarrow$ (δ(a),ι+1,δ,γ),
          mk_Add(a) $\rightarrow$ (α+δ(a),ι+1,δ,γ),
          mk_Sub(a) $\rightarrow$ (α−δ(a),ι+1,δ,γ),
          ...
          mk_Jmp(i) $\rightarrow$ (α,i,δ,γ),

```
                    mk_CJp(i) → (α,if α=0 then i else ι+1 end,δ,γ),
                    mk_Stp(n) → (α,2^m,δ,γ), ... end
        in M(σ) end end end
```

■

The computer, as a domain, and represented by M, Ins and Σ, has the programmable active dynamic attribute.

Principle. *Programmable Active Dynamics:* Few domain phenomena are of the programmable active dynamic attribute. Even "programmed" computer and communications systems may fail. ■

Techniques. *Programmable Active Dynamics:* Usually internal nondeterminism is not at play in programmable active dynamic phenomena. Otherwise, as for biddable active dynamic phenomena, the full spectrum of modelling techniques apply across the board, that is, without discrimination. In other words, we cannot emphasise special techniques. Thus context- and state-dependent functions may take on a full variety of signatures:

value
  eVALuate: ... Context → State → Value
  INTerpret: ... Context → State → State
  ELABorate: ... Context → State → State × Value
  DECLare: ... Context → State → State × Context

The latter signature is typical of computations over declarations which create new contexts. ■

Reactive Dynamics Attribute

We present a characterisation, show one or more examples and formulate modelling principles and techniques.

Characterisation. *Reactive Dynamics:* An domain phenomenon is reactive if it performs actions in response to external stimuli.[5] Thus three properties must be satisfied for a system (a subset of a larger domain) 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. ■

[5] We refer to the entry on Domain Characteristics, on page 69, fourth paragraph, in [189].

The above characteristation does not exclude that a reactive domain phenomenon additionally changes state of it own volition. Inert phenomena only change state as the result of explicit external stimuli and the new state of a phenomenon can be predicted on the basis of the old state of the phenomenon and the external stimulus "input" (i.e., applied) to the phenomenon.

Example 10.19 *Railway Fork Switches:* A railway fork switch is a reactive phenomenon. It is basically expected to only change state, opening/closing paths through the switch unit, as the result of being supplied with appropriate mechanical or electrical stimuli. ∎

Example 10.20 *Variety of Reactive Dynamic Phenomena:* We briefly hint at examples of reactive dynamic phenomena: With reference to the examples above of biddable active dynamic phenomena, that is, Examples 10.14 and 10.15, we can say that aircraft, ships, automobiles and trains are reactive phenomena, whether or not we include pilots, captains (first stewards), drivers and engine drivers. ∎

Principles. *Reactive dynamic* phenomena are crucially characterised by the interaction between these phenomena (the system) of the domain and the environment (of this system). The domain of reactive dynamic phenomena is the main target for applications of so-called real-time, safety-critical and embedded computing systems. ∎

Volume 2, Chaps. 12–15 (*Petri Nets, Statecharts, Message and Live Sequence Charts* and *Quantitative Models of Time*) gave a long list of examples and detailed principles and techniques for modelling reactive systems.

Techniques. *Reactive Dynamics:* One or more processes model the domain. External events, i.e., channels that share events with the domain, but which communicate messages from an 'environment' — external to the domain — model that environment. Sometimes we may choose also to model part of the environment as processes from which these "external" events emanate. Thus the domain synchronises with events of the environment and responds by performing actions. Otherwise the full complement of modelling techniques apply. ∎

Tools. *Reactive Dynamics Phenomena:* Since time often is of crucial importance in modelling reactive dynamic phenomena appropriate languages for formal descriptions are needed. Example of such languages (and their support software packages) are: Statechart (Statemate) [144,145,147,148,150], Duration Calculi [381,382], TLA+ [209,210,239], Esterel [27], Lustre [136,137], Signal [25] and others. ∎

Discussion

General: The concept of dynamic domain is basically synonymous with the concept of *state*. *A state is everything that is true at a particular [point in] time.*[6] *A state is a summary of process actions. A state is the value of a predicate over variables.* The concepts of dynamic domain and state are thus strongly related to the concepts of events and processes.

Interplay between static and dynamic attributes: Seen from one viewpoint we may consider an entity of static nature, being a constant, immutable. Seen from another viewpoint we may consider that same entity of dynamic nature, taking on varying values.

Example 10.21 *Rail Units:* We refer to Examples 7.1–7.4 and 7.11. In those examples, any one rail unit, seen from a spatial viewpoint, never changed spatial characteristics. But it might, from time to time, change state. ▪

10.4 Tangibility and Intangibility

The issue of tangibility is an issue of human versus machine, that is, an issue of how to record, how to observe phenomena. We may distinguish between three kinds of tangibility: humanly tangible phenomena, otherwise physically tangible phenomena and intangible phenomena. We will now cover these in turn.

10.4.1 Humanly Tangible Phenomena

Characterisation. A domain phenomenon is *humanly tangible* if it can be sensed by humans: seen, heard, tasted, touched or smelled. ▪

We do not include emotionally experienced individuals (i.e., phenomena) among humanly tangible phenomena. The humanly tangible senses can be measured. Emotions cannot be measured objectively. Humanly tangible individuals are inevitably informal: We can never hope to formalise all aspects, all properties of humanly tangible phenomena.

Example 10.22 *Varieties of Humanly Tangible Phenomena:* We informally present a variety of humanly tangible phenomena: automobile, train, aircraft and ship movement; flow of people, equipment and materials in the health-care sector; flow of people, equipment and materials in manufacturing plants (factories); human heart pulse beats. ▪

[6] We refer to the entry on Domain Characteristics, on page 67, paragraph 3, in [189].

We next give a formal model example, most of whose entities are humanly tangible.

Example 10.23 *Manufacturing:* A metal machining manufacturing plant, i.e., a factory, consists of machines (machine(i)) numbered (i) from 1 to n, and a production line (pl). Each machine, i.e., a lathe, a band saw, a belt sander, a milling machine, a drill press, a grinder, a shear, a notscher, a press brake, etc., accepts metal parts (one, m) from, and delivers metal parts (one, m′) to the production line. Machines perform operations, $o : \mathcal{O}$, (upon) the metal parts, each machine being identified by one operation. The production line, i.e., conveyor belt, has, for each input to a machine, as indexed from 0 to n, and for the common output from machines to the environment, as indexed by $n+1$, a possibly empty set of metal parts. Each machine has a further unexplained state ($\sigma : \Sigma$). In this simple example each operation takes one metal part from the production line (ie, from its input to this machine) and the machine state, and delivers, results in, one processed metal part to the line together with the identity, i.e., index, of the machine to which this part should go.

Formal Presentation: Manufacturing, I

```
value
  n:Nat axiom n>0
type
  Σ                          /* Machine tool state */
  Mat                        /* Material (i.e., metal parts) */
  Idx0n_1 = {|0..n−1|}       /* Input: exterior (0) or interior (1..n−1)  */
  Idx1n = {|1..n|}           /* Machine index */
  Idx1n′ = {|1..n+1|}        /* Output: interior (1..n) or exterior (n+1) */
  ProdLine = Idx0n_1 ~m~> (Idx1n′ ~m~> Mat-set)
  MΣ = Idx1n ~m~> Σ
  MO = Idx1n ~m~> O
  O: Mat → Σ → Σ × (Mat×Idx1n′)
```

When starting up the plant we initialise each machine to a state depending on its index i, i.e., mσ(i), and to an operation, i.e., oω(i). The production line p is likewise initialised to some state pl, which is usually, but not shown, empty outputs and nonempty inputs from the exterior to machines.

Formal Presentation: Manufacturing, II

```
value
  mσ:MΣ
  oω:MO
  pl:ProdLine
```

```
line: Unit → Unit
line() ≡ ‖ { machine(i)(oω(i))(mσ(i))|i:Idx1n} ‖ p(pl)
```

The conveyor belt, i.e., the "production line", is "mimicked", i.e., simulated, by a set of channels: One set, pm[j:0..n,i:1..n+1], from the line to the machines, such that results from machine m[j], or the exterior (index 0), sent to machine m[i] or the exterior (index n+1), are conveyed on channel pm[j,i] . Another set, mp[i:1..n,j:1..n+1], from the machines to the line, such that results from machine m[i], sent to machine m[j] or the exterior (index n+1), is conveyed on channel mp[i,j]

Formal Presentation: Manufacturing, III

```
channel
  {pm[j,i]|j:Idx0n,i:Idx1n'} : (Mat×Idx1n')
  {mp[i,j]|i:Idx1n,j:Idx1n'} : (Mat×Idx1n)
```

A machine now iterates between fetching parts from the line, m=pm[j,i]?, processing a part, o(m) (= (σ',(m',k))), delivering the processed part to the line, k]!m', and "back, all over again", machine(i)(o)(σ'').

Formal Presentation: Manufacturing, IV

```
value
  machine: i:Idx1n → 𝒪 → Σ →
                  in {pm[j,i]|j:Idx0n} out {mp[i,k]|k:Idx1n'} Unit
  machine(i)(o)(σ) ≡
    let σ'' =
        [] {let m=pm[j,i]? in let (σ',(m',k))=o(m) in mp[i,k]!m';σ'
            | j:Idx0n end end} in
    machine(i)(o)(σ'') end
```

The production line (p) nondeterministically chooses (by itself, i.e., internal choice, ⌈⌉), whether to provide input to a machine, or to accept output from a machine. In the former case the line arbitrarily selects a pair of machines, and/or the exterior, and if there are parts on that "channel" of the line, then it arbitrarily selects a part, updates the line and sends that part to the machine. If there are no parts, then it continues being a production line! In the latter

case the production line (p) is willing to accept output from any machine, and waits until such output is provided, if ever, and joins it to the conveyor belt.

Formal Presentation: Manufacturing, V

```
value
  p: ProdLin → in {pm[i,j]|j:Idx0n} out {mp[i,j]|j:Idx1n′} Unit
  p(pℓ) ≡
    let i,j:Nat • i ∈ dom pℓ∧ j ∈ dom pℓ(i) in
    if (pℓ(i))(j) ≠{}
      then
        let m:Mat • m ∈ (pℓ(i))(j),
            update = [j↦(pℓ(i))(j)\{m}],
            pℓ′ = pℓ†[i+ pℓ(i)†update]
        in pm[i,j] ! m ; p(pℓ′) end
      else p(pℓ) end
    ⌈⌉
    ⌈⌉ {let m = mp[i,j]?,
            update = [j↦(pℓ(i))(j)∪{m}]
        in pℓ†[i↦pℓ(i)†update] end|i:Idx0n,j:Idx1n′}
    end
```

Please convince yourself that most entities mentioned above are humanly tangible. ▪

10.4.2 Otherwise Physically Tangible Phenomena

All humanly tangible phenomena are — to some degree — physically tangible. Chemical phenomena are usually sensible by physical apparatuses. Also physically tangible individuals are inevitably informal: We can thus never hope to formalise all aspects, all properties of otherwise physically tangible phenomena. Generally physically (incl. humanly) tangible phenomena are said to be manifest.

Characterisation. *Otherwise physically tangible phenomena* are those that can be measured by such measurement instruments as mechanical, electromechanical, electrical, electronic, chemical, electrochemical, and otherwise — that is instruments that measure physical, chemical, biological, and like quantities. ▪

Example 10.24 *Instrument to Computer "Hook-ups":* For the application domain of software for medical electronic instruments monitoring and controlling blood pressure, certain aspects of lung conditions (through X-ray instruments) and heart behaviour (through electrocardiogram instruments) are

physically tangible. For the application domain of nuclear physics electron spin is physically tangible. ▪

10.4.3 Intangible Phenomena

Intangible phenomena relate strongly to intellectual concepts. We distinguish between two kinds of intangible phenomena: imprecise or informalisable concepts, and precise, formalisable concepts.

Characterisation. *Intangible phenomena* are those phenomena which are not humanly or physically tangible. ▪

Example 10.25 *Varieties of Intangible Phenomena:* (i) Human emotions and human moods are intangible — and cannot be made objectively precise, let alone formalised. (ii) The mathematical numbers, truth values, sets, functions, etc., are intangible, but most can be formalised, and all can be made sufficiently precise. (iii) The pragmatics and semantics of text being manipulated by the word processor process are intangible. (iv) The pragmatics and semantics of a program parse tree traversed by the compilation process are intangible. (v) The pragmatics and semantics of database files and records which perhaps reflect external world information (insofar as the problem context in which we understand them as "data residing within the computer") are intangible. Items (ii–v) are all intangible but can be fully formalised. ▪

Since any definition, and hence any formalisation, designates an intangible phenomenon, and since these volumes are otherwise full of definitions and formalisations, we shall not further exemplify the notion of intangible phenomena.

10.4.4 Discussion

Perhaps we should have brought in this section on tangibility much earlier in these volumes. Certainly some worries and difficulties we might have had in expressing what can be described and how to express those descriptions could have been made didactically and hence pedagogically simpler by another sequence of presentation of the topics of these volumes. Be that as it may. Now we have discussed the important idea of tangibility, and we have seen its relation to describability and formalisation.

The reason why we have brought in the material on tangibility and intangibility here, and not earlier, is that the concepts of tangibility and intangibility are attributes of domain entities, and this chapter is about domain attributes.

10.5 One, Two, ..., Dimensionality

Our ability to precisely describe individuals in a way that closely fits reality is much hampered by the one-dimensionality of textual language. The problem arises in connection with the description of complex individuals, that is, individuals composed from two or more other either simple or complex individuals.

Example 10.26 *Spatial GUI Window Objects:* A graphical user interface (GUI) can be thought of as a rectangular, hence a geometric figure. Upon the GUI is superimposed a number of so-called windows. A window can also be thought of as a rectangular, hence a geometric figure. One window may partially or fully overlap another window. Given a sequence of two or more windows $< w_1, w_2, \ldots, w_n >$, let it be the case that w_i overlaps w_j, for some $i < j$. It cannot also be the case that w_j overlaps w_i, for the same i, j as just mentioned. It therefore seems that a GUI window is three-dimensional: two for the simple window itself, and an additional dimension for the stacking or overlapping of windows. ∎

In describing we normally introduce defined terms. Let a definition consist of one or more simple definitions, each of which is just a pair of the term being defined and the description text defining that term. Definitions usually consist of several simple definitions which are ordered sequentially.

We introduce some definitional terms. When an earlier (or a later) description text (of some simple definition) refers to a term (simply) defined later (respectively earlier) we speak of a forward (backward) reference. When a term is defined immediately in terms of itself we speak of an immediate recursive reference.[7] Such a definition may spread over several simple definitions. When a term is nonrecursively defined in terms of constituent terms, these eventually denote smaller classes of individuals, and we therefore speak of descending references. Recursive terms are therefore not descending references. Similarly we can speak of ascending references.

Example 10.27 *Abstract Trees:* We assume labels to be a defined notion. A root is a label. A branch has a label. A (finite) tree consists of a root and a (finite) set of zero, one or more distinctly labelled branches. A branch is a tree.

The sentences of the above definition cannot be reordered to only have (immediate) recursive and descending references. ∎

We leave it to the reader to formalise Examples 10.26 and 10.27.

[7] When a term is defined in terms of itself, but via some intervening (simple) definitions we also speak of a recursive reference, but now it is not an immediate recursive reference.

10.5.1 Zero Dimensionality

The simplest phenomena are the zero-dimensional ones, which we refer to as atomic. We model them by atomic data types: numbers, Booleans, characters and further unexplained sorts, and by simple operations on these zero-dimensional values.

10.5.2 One Dimensionality

The next simplest phenomena are the one-dimensional ones. They are typically understandable as linear sequences.

Characterisation. A complex individual is a *one-dimensional* individual if all relations between adjacent properly contained (i.e., constituent) individuals can be described in terms of only descending references, and thus either only forward or only backward references. ∎

Example 10.28 *Varieties of One-Dimensional Phenomena:* (i) Text to be manipulated by a simple word processor is one-dimensional: Symbols form the atomic elements and are either characters, digits, blanks, carriage returns or other. Text is a linear sequence of symbols. (ii) A train journey can usually be considered a one-dimensional individual: A train journey consists of a linear sequence of station visits. A station visit is here further undefined. ∎

Observe that we chose to make use of a backward reference only among defined terms for the first subexample (i), but a forward reference only among defined terms for the second subexample (ii). Of course, we cannot prevent a writer from confusing readers by unnecessarily using both forward and backward references when only one direction is needed.

Principle. *One-Dimensional Phenomena:* Do not expect that interesting phenomena or concepts are one-dimensional. When a phenomenon or a concept is one-dimensional then describe it as simply as possible: with only forward, or with only backward references. Use backward references if the phenomenon or concept is unfamiliar to the reader of your description; use forward otherwise. Investigate the concept likewise: atomic parts first, if unfamiliar; then their composition. ∎

Forward references only correspond to the notion of hierarchical description (Vol. 2, Chap. 2). Correspondingly, backward references only correspond to the notion of compositional description (also covered in Vol. 2, Chap. 2).

Techniques. *One-Dimensional Phenomena:* We model the type of interest as a simple set type, a Cartesian type or a sequence type of simpler type (scalar or such, but not recursive types), and with appropriate observer and possibly also generator functions. ∎

We treat vector, matrix, tensor, i.e., arbitrary dimension array, concepts all as one-dimensional since they all enjoy a simple notion of (multiple) adjacencies.

10.5.3 Multidimensionality

Computer and computing science is distinguished from ordinary mathematics by dealing with highly structured, multidimensional phenomena whose proper constituents are themselves such phenomena.

Example 10.29 *Mathematical Expressions:* Programs of programming languages are multidimensional individuals, as are, in general, any natural language sentences. Mathematical, not necessarily computing, notions such as (names of) constants (numerals, etc.), (names of) variables (identifiers) and (pre-, in- and suffix) operators are further undefined zero- or one-dimensional terms. Expressions are either constants or variables, or are multidimensional pre-, in-, suffix- or distributed fix composites. An infix composite consists of a linear sequence of an expression, an infix operator and an expression. ∎

Observe that we cannot reorder the above text to avoid having recursive references.

Characterisation. A complex individual is a *multidimensional phenomenon* if some relations between adjacent properly contained (i.e., constituent) individuals can only be described by both forward and backward references, and/or with recursive references. ∎

Example 10.30 *A Variety of Multidimensional Phenomena:* (i) Hyperlinked word processor texts: First, we have simple one-dimensional word processor texts. Then we associate with some (arbitrarily chosen) subsequences of symbols the concept of a node. Then we may associate two or more nodes with each other through a notion of link. The links are like edges in a graph of the nodes, but are allowed to connect, not just two, but three or more nodes. Linked nodes may designate overlapping text segments. The above can be analysed to constitute four-dimensional phenomena.

(ii) A network of roads with bridges (over roads, rail lines, rivers, lakes, seas and canals), and tunnels (through mountains, under roads, rail lines, rivers, seas and canals), of rail lines (etc.), of rivers and of canals (etc.), typically form a three-dimensional phenomenon. ∎

Principles. Expect that "interesting" phenomena are *multidimensional.* Analyse them carefully. Seek utter simplicity, even though the model remains multidimensional. ∎

Techniques. We typically model *multidimensionality* in terms of possibly recursive types over graphs and with associated observer and generator functions. ∎

10.5.4 Discussion

The real issue is that of multidimensionality. And the derived, main issue is that of diagramming, in two and higher dimensions.

The problems, as we see them, are as follows: On one hand, we have the linear texts of narratives and formal specifications. On the other hand, we have a plethora of diagrammatic notations. Some of the latter have well-founded semantic theories, and hence constitute formal specification languages, viz.: Petri nets (Vol. 2, Chap. 12) [196, 273, 293–295] and statecharts (Vol. 2, Chap. 14) [144, 145, 147, 148, 150].

But some of the latter do not have well-founded theories, viz.: UML's class diagrams (Vol. 2, Chap. 10) [44,193,264,303]: UML class diagrams do not have precise syntax and semantics. They have no refinement relation: When is one diagram an implementation (decomposition) of another diagram? And they have no logic: One cannot reason over diagrams.

Yet diagrams have a tremendous appeal. Many proposals have been put forward to *combine* formal specification languages with diagrammatic notations: the combination of `VDM-SL` with `UML`-like features [92–94]; combination of `Z` with `Petri Nets`; and so on. Research is only now picking up on these *unifying theory* [171] and *integration of formal methods* [16, 43, 52, 132] challenges, while cognitive and other scientists study the topic of *reasoning with diagrams* [10].

10.6 Discussion

We have overviewed and given some depth to a treatment of a number of domain attributes: continuous, discrete, hybrid and chaotic attributes; statics and varieties of dynamic attributes; tangibles and intangibles; and zero, one and multidimensionality attributes.

For some attributes we have given only informal, yet terse, narrative descriptions, while for others we have given both those and formal descriptions. There are other attributes, but these are covered in Vols. 1 and 2.

When confronted with some domain phenomena to be described, the developer analyses these, to decide, along one dimension of analysis, whether they possess the kind of attributes covered in this section, and, if so, in which combination. Then the developer uses the hints given in enunciated principles and techniques in order to describe them, i.e., model them — whether informally or formally. The reader should, however, not forget that the universe of discourse for domain attributes can also be requirements and software design — not just an application domain!

10.7 Bibliographical Notes

The main bibliographical reference is to Michael Jackson's delightful [189].

10.8 Exercises

10.8.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples. We refer also to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

10.8.2 The Exercises

The use of the term 'describe' means to rough sketch and/or terminologise, and to narrate. If you are studying this volume in its formal version, then the term describe additionally means formalise.

Exercise 10.1 *Continuous, Discrete and Chaotic Phenomena.* For the fixed topic, selected by you, identify and describe:

- one or more continuous,
- one or more discrete and, possibly,
- one or more chaotic

phenomena.

Exercise 10.2 *Static Phenomena.* For the fixed topic, selected by you, identify and describe some static entities.

Exercise 10.3 *Dynamic Inert Phenomena.* For the fixed topic, selected by you, identify and describe some dynamic inert entities.

Exercise 10.4 *Dynamic Active Phenomena.* For the fixed topic, selected by you, identify and describe some dynamic active entities: Autonomous, biddable and/or programmable; preferably at least one of each.

Exercise 10.5 *Dynamic Reactive Phenomena.* For the fixed topic, selected by you, identify and describe some dynamic reactive entities.

Exercise 10.6 *Tangible and Intangible Phenomena.* For the fixed topic, selected by you, identify and describe some tangible, and some intangible phenomena.

Exercise 10.7 *Dimensional Phenomena.* For the fixed topic, selected by you, identify and describe some dimensional phenomena. Preferably at least one 0-, one 1-, and some 2- or more dimensional phenomena.

11

Domain Facets

- The **prerequisite** for studying this chapter is that you, as a domain engineer, need to know: *which are the constituents of a proper model of a domain?*
- The **aims** are to introduce the concept that a proper domain description is made up from most of the following constituent descriptions, i.e., facets: domain-facilitating business processes, domain intrinsics, domain support technologies, domain management and organisation, domain rules and regulations, domain scripts, human behaviour, etc., and to present principles, techniques and tools for the description of these facets.
- The **objective** is to ensure that you will become a thoroughly professional domain engineer.
- The **treatment** is from systematic to formal.

11.1 Introduction

Let us remind ourselves of what it is all about. Software development is all about getting software to the market, software that can and will be sold. Hence it must be software whose use pleases people, software which solves problems, that is, software which fits, hand in glove, the application domain in which it is to serve.

Therefore describing the domain is important. If we cannot describe the domain, then we are not trustworthy. We simply cannot be trusted to develop software for that domain. Describing the domain is thus of utmost importance. And hence it is of primary importance to know and to practice what a description consists of.

This chapter is all about that: to identify the various facets of a domain that are describable, and, hence, most likely, are parts of a proper domain description. So, in this chapter we will identify those facets, and we will present principles, techniques and tools for their proper description.

The present chapter constitutes a first high point of the present volume, because in this chapter we present principles and techniques of software development that are not otherwise available in any other textbook on software engineering. So take your time to become thoroughly familiar with the contents of the present chapter.

Characterisation. 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. ▪

In this section we identify a number of *domain facets* and we survey principles and techniques for modelling, relative to identified domain stakeholder classes, each of the identified facets. So far we have been able to identify the following facets:

- (i) *intrinsics*,
- (ii) *support technology*,
- (iii) *management and organisation*,
- (iv) *rules and regulations* including
- (v) *script* facets, and
- (vi) *human behaviour*.

We enlarge upon the above enumeration using the following brief characterisations:

- (i) **Domain intrinsics:** *That which is common to all facets* (Sect. 11.3).
- (ii) **Domain support technologies:** *That in terms of which several other facets (intrinsics, business processes, management and organisation, and rules and regulations) are implemented* (Sect. 11.4).
- (iii) **Domain management and organisation:** *That which primarily determines and constrains communication between enterprise stakeholders* (Sect. 11.5).
- (iv–v) **Domain rules, regulations and scripts:** *That which guides the work of enterprise stakeholders, their interaction, and the interaction with non-enterprise stakeholders* (Sects. 11.6–11.7).
- (vi) **Domain human behaviour:** *The way in which domain stakeholders dispatch their actions and interactions wrt. the enterprise: dutifully, forgetfully, sloppily, yes, even criminally* (Sect. 11.8).

To help us identify parts of the above facets we suggest that rough sketch descriptions first be made of what we shall call the domain business process facilitators:

- **Domain business process facilitators:** *Those processes — carried out primarily by people — in terms of which the intrinsics (and so on) are implemented* (Sect. 11.2).

11.1.1 Separation of Concerns

We shall now treat each of these facets in some detail. For each we venture to express some specification pattern that most closely captures the essence of the facet. Separating the treatment of each of these (and possibly other) facets reflects the following principle:

Principle. *Separation of Facets:* When possible, one should identify distinguishable facets and, when appropriate, i.e., if feasible and pleasing, treat them separately. ▪

We believe that the facets we shall present can be treated separately in most developments — but not necessarily always. *Separation or not* is a matter of development as well as of presentation style.

11.1.2 Discussion of the Separation Principle

The separation, in more generality, of computing systems development into the triptych of domain engineering, requirements engineering and machine (hardware + software) design is also a result of separation of concerns. So are the separations of domain requirements, interface requirements and machine requirements (within requirements engineering), as well as the separation of software architecture and component and module design.

11.1.3 Structure of Chapter

Before we cover each of the facets individually (Sects. 11.3–11.8) we cover the concept of business process facilitators (Sect. 11.2). The material of Sect. 11.2, in addition to helping the domain describer to identify the various facets of a domain, also covers the important notion of business processes. Describing business processes is not only the responsibility of a software developer, but also of managers in any business enterprise. Before having, even superficially, understood current business processes how could a business manager mandate the reengineering of these processes? Section 11.2 therefore also serves as a prerequisite for the section on business process reengineering (a domain requirements facet, Sect. 19.3).

11.2 Domain Facilitators: Business Processes

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

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

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

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

11.2.1 Business Processes

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

Example 11.1 *Some Business Processes:*

(i) A Business Plan Business Process: The board of any company instructs its chief executive officer (CEO) to formulate revised business plans.[1] Briefly, a business plan is a plan for how the company strategically, tactically and, to some extent, operationally wishes to conduct its business: what it strives for,

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

productwise, imagewise, market-share-wise, financially, etc. The CEO develops a business plan in consultation with executive layers of (i.e., with strategic) management. Strategic management (in-between) discusses the plan (which the CEO wishes to submit to the Board) with tactical management, etc. Once generally agreed upon, the CEO submits the plan to the Board.

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

Example 11.2 *Some More Business Processes:* The University of California at Irvine (UCI), had their Administrative and Business Services department suggest, as a learning example, the description of a number of business processes. The "learning" had to do, actually, with business process reengineering (BPR). So we really should bring the below example into Sect. 19.3 instead of here! We quote from their home page [357]:

- **Human Resources:** "Examine the hiring business process of the University, including the applicant process. Special emphasis should be given to simplifying the process, identifying those parts where there is no value added — i.e., where those parts of the process which one considers *simplifying "away"* add no value. Increase speed of response to applicant and units, and reduce process costs while achieving high quality."
- **Renovation:** "Review the campus' remodelling and alterations business process, and develop recommendations to improve Facilities Management services to UCI departments for small projects (under $50,000) and minor capital projects (up to $250,000). Special emphasis should be given to simplifying the process, identifying those parts where there is no value added

to the customer's product; to increase speed and flexibility of response; and to reduce process costs while achieving high quality."

- **Procurement:** "Review the campus procurement business process and develop recommendations/solutions for process improvement. The redesigned process should provide "hassle-free" purchasing, give a quick response time to the purchaser, be economical in terms of all costs, be reasonably error-free and be compliant with (US) Federal procurement standards."
- **Travel:** "Study the travel business process from the beginning stage when a faculty/staff member identifies the need to travel to the time when reimbursement is received. Analyze and redesign the process through a six step program based on the following business process improvement (BPI) principles: (i) simplify the process, (ii) identify those parts where there is no value added to the customer, increase (iii) speed and (iv) flexibility of response, (v) improve clarity for responsibilities and (vi) reduce process costs while meeting customer expectations from travel services. The redesign should reflect customer needs, service, economy of operation and be in compliance with applicable regulations."
- **Accounts payable:** "Redesign the accounts payable business process to meet the following functional objectives (in addition to BPI measures): Payment for goods and services must assure that vendors receive remittance in a timely manner for all goods and services provided to the University. Significantly improve the operation's ability to serve campus customers while maintaining financial solvency and adequate internal controls."
- **Parking:** "Review how parking permits[2] are sold to students, faculty and staff with the intent of omitting unnecessary steps and redundant data collection. The redesigned process should achieve a dramatic reduction in time spent by people standing in line to purchase a permit, and reduce administrative time (and cost) in recording and tracking permit sales."

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

Characterisation. By *business process engineering* we understand the identification of which business processes should be subject to precise description, describing these and securing their general adoption (acceptance) in the business, and enacting these business process descriptions. ▪

[2] We here assume that the company is a very large company with extensive, but still limited, parking facilities.

Example 11.3 *Example Business Process Engineering:*

(i) *Business plans:* We assume, about our example company, that — up to a certain time — there was no set procedure wrt. the creation, etc., of business plans. As the company grows, a need is felt for "stricter" procedures wrt. business plans. Therefore the CEO and/or the board drafts the business plan very implicitly hinted at in Example 11.1 (i). The last two sentences, above, portray an example business process engineering.

(ii) *Purchase regulations:* We assume, about our example company, that — up to a certain time — there was no set procedure wrt. purchase of equipment. As the company grows, a need is felt for "stricter" procedures wrt. procurement. Therefore some (say, operations) manager drafts the purchase process roughly sketched in Example 11.1 item (ii). The previous two sentences portray an example business process engineering. ▪

11.2.2 Overall Principles

We summarise:

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

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

Tools. *Business Processes:* Further techniques and the basic tools for describing business processes include: (1) `RSL/CSP` definition of processes, where one suitably defines their *in*put/*out*put signatures, associated *channel* names and *type*s, and their process definition bodies;[4] (2) Petri nets;[5] (3) message

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

[4] `RSL/CSP` [168, 301, 311] was covered in detail in Vol. 1, Chap. 21.

[5] Petri Nets [196, 273, 293–295] were covered in detail in Vol. 2, Chap. 12.

and live sequence charts for the definition of interaction between behaviours;[6] (4) statecharts for the definition of highly complex, typically interwoven behaviours;[7] and (5) the usual, full complement of RSL's *type*, function *value*, and *axiom* constructs and their abstract techniques for modelling entities and functions. ∎

11.2.3 Informal and Formal Examples

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

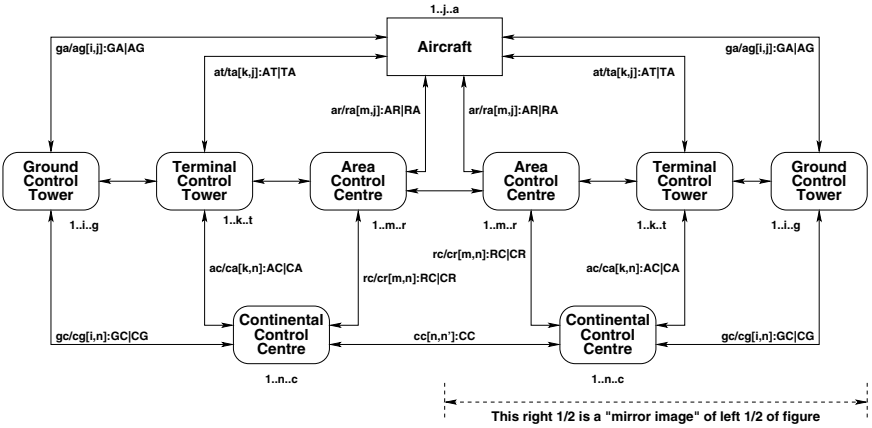


Fig. 11.1. An air traffic behavioural system abstraction

Example 11.4 *Air Traffic Business Processes:* The main business process behaviours of an air traffic system are the following: (i) the aircraft, (ii) the ground control towers, (iii) the terminal control towers, (iv) the area control centres and (v) the continental control centres (Fig. 11.1).

We describe each of these behaviours separately:

[6] Message [182–184] and live sequence charts [73, 149, 203] were covered in detail in Vol. 2, Chap. 13.

[7] Statecharts [144, 145, 147, 148, 150] were covered in detail in Vol. 2, Chap. 14.

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

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

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

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

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

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

Example 11.5 *Freight Logistics Business Processes:* The main business process behaviours of a freight logistics system are the following: (i) the senders of

freight, (ii) the logistics firms which plan and coordinate freight transport, (iii) the transport companies on whose conveyors freight is being transported, (iv) the hubs between which freight conveyors "ply their trade", (v) the conveyors themselves and (vi) the receivers of freight (Fig. 11.2). A detailed description for each of the freight logistics business process behaviours listed above should now follow. We leave this as an exercise to the reader to complete. ∎

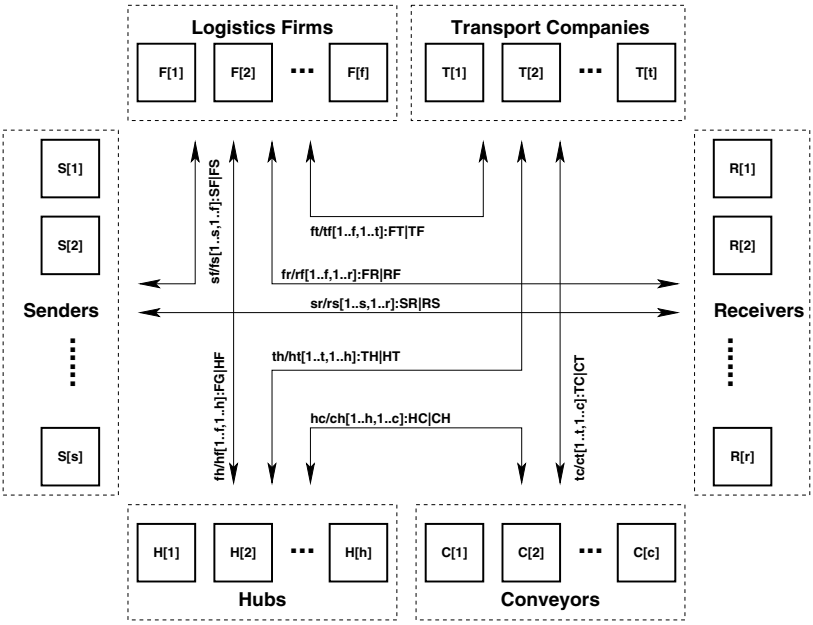


Fig. 11.2. A freight logistics behavioural system abstraction

Example 11.6 *Harbour Business Processes:* The main business process behaviours of a harbour system are the following: (i) the ships who seek harbour to unload and load cargo at a harbour quay, (ii) the harbourmaster who allocates and schedules ships to quays, (iii) the quays at which ships berth and unload and load cargo (to and from a container area) and (iv) the container area which temporarily stores ("houses") containers (Fig. 11.3). There may be other parts of a harbour: a holding area for ships to wait before being allowed to properly enter the harbour and be berthed at a buoy or a quay, or for ships to rest before proceeding; as well as buoys at which ships may be anchored while unloading and loading. We shall assume that the reader can properly complete an appropriate, realistic harbour domain.

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

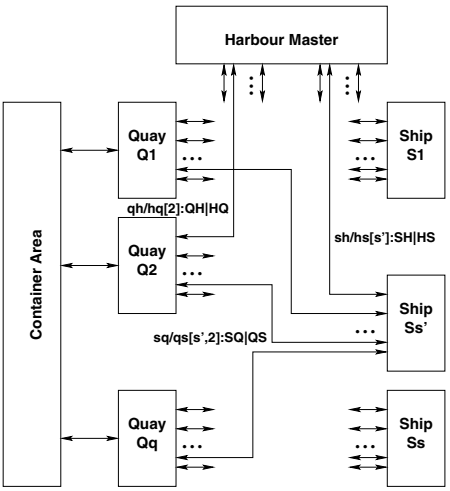


Fig. 11.3. A harbour behavioural system abstraction

Example 11.7 *Financial Service Industry Business Processes:* The main business process behaviours of a financial service system are the following: (i) clients, (ii) banks, (iii) securities instrument brokers and traders, (iv) portfolio managers, (v) (the, or a, or several) stock exchange(s), (vi) stock incorporated enterprises and (vii) the financial service industry "watchdog". We rough-sketch the behaviour of a number of business processes of the financial service industry.

(i) Clients engage in a number of business processes: (i.1) they open, deposit into, withdraw from, obtain statements about, transfer sums between and close demand/deposit, mortgage and other accounts; (i.2) they request brokers to buy or sell, or to withdraw buy/sell orders for securities instruments (bonds, stocks, futures, etc.); and (i.3) they arrange with portfolio managers to look after their bank and securities instrument assets, and occasionally they reinstruct portfolio managers in those respects.

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

(iii) Securities instrument brokers and traders engage with clients, portfolio managers and the stock exchange(s) in exchanges related to client transactions

with brokers and traders, and, for traders, as well as with the stock exchange(s) on their own behalf, as clients.

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

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

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

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

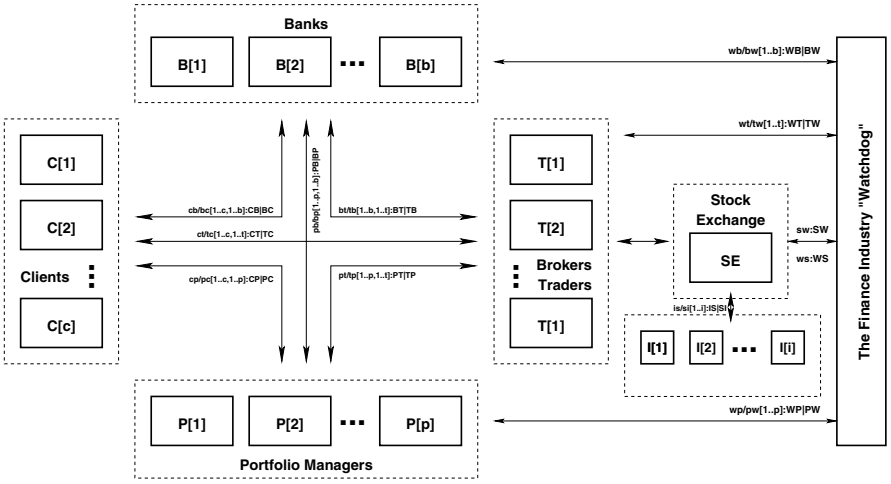


Fig. 11.4. A financial behavioural system abstraction

11.2.4 Discussion

The reader is to be properly warned.

An essence of the examples is not the specific diagrams shown, but that one can indeed draw such behavioural rough sketches. These can include square or rounded boxes designating behaviours; single- or, as here shown, double-ended arrows, designating the possibility of typed communication of messages (say over channels); the (entity) typing of these messages; and so on.

Another essence of the examples is hence that there is a diagrammatic language of behaviours, and that this language has textual counterparts — say in the form of CSP or RSL/CSP. Other diagrammatic forms might be chosen, depending on properties not revealed in the above examples. These other forms might be Petri nets, message or live sequence charts, or, for example, statecharts.

Furthermore, the examples are sketchy, but they provide an immediate, constructive start to the arduous task of carefully and painstakingly describing a domain.

In all examples we have sketched the suggested arrays of channels and their types (as sorts). These are just suggestions. Interactions between behaviours are then modelled in terms of messages communicated over these channels. But such models are just that: there is no obligation on the part of any, subsequent software design to implement channels as something anywhere similar to channels!

The reader should understand that to describe domains fully satisfactorily requires at least the full complement of principles, techniques and tools covered in all chapters of Vols. 1 and 2, as well as in all the chapters up to and including all of the present chapter in this volume!

11.2.5 Summary

The purpose of first rough-sketching a number, not necessarily all, identifiable business processes is to use these descriptions to identify

- entities,
- functions,
- events and
- behaviours,

as well as to classify these into their "facethood":

- intrinsics,
- support technologies,
- management and organisation,
- rules and regulations,
- scripts and
- human behaviour.

11.2.6 Reminder

We remind the reader of the principle stated at the outset of this section on domain business process facets.

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

A main reason for initially describing the business processes of a domain is to discover, identify and capture entities, functions, events and behaviours of that domain. Another good reason is to get the process of description started — somewhere!

11.3 Domain Intrinsics

Railways, although they have many "players and actors" revolve around some core notions: the rail net and trains on these. Overlapping groups of players and actors (i.e., stakeholders), hence different perspectives, in general, have a core of common entities and phenomena. We refer to this core as *the intrinsics of the domain.*

Principles. *Domain Intrinsics:* From the outset of describing a domain: Analyse it with respect to its intrinsic phenomena and concepts. Focus on describing these first. Make sure that the descriptions of subsequently described domain facets are subordinated descriptions of the domain facets. ∎

Principle. *Describing the Domain Intrinsics Facets:* So from the outset of describing a domain analyse it with respect to its intrinsic phenomena and concepts. Focus on describing these first, and make sure that the descriptions of all other (subsequently described) domain facets are subordinated descriptions of the domain intrinsics. ∎

11.3.1 Overall Principles

Each stakeholder group typically has its view of a domain. Different stakeholder groups may thus have different views of their — otherwise shared — domain. In developing a description of the domain intrinsics we must first develop one description per stakeholder group. Then, in some step of development, reconcile possible domain description inconsistencies and conflicts. To do so systematically we first need to form a basis, the intrinsics, which is common to all subsequent facets.

Characterisation. By domain *intrinsics* we shall understand 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. ∎

In the next many examples we show typical fragments of rough-sketch or narrative descriptions — such as the software developer has to construct when creating a domain description.

Example 11.8 *Railway Net Intrinsics:* We narrate and formalise three railway net intrinsics.

- From the view of *potential train passengers* a railway net consists of lines, stations and trains. A line connects exactly two distinct stations.
- From the view of *actual train passengers* a railway net — in addition to the above — allows for several lines between any pair of stations and, within stations, provides for one or more platform tracks from which to embark or alight a train.
- From the view of *train operating staff* a railway net — in addition to the above — has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path (through a unit) is a pair of connectors of that unit. A state of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in either one of a number of states of its state space.

Formal Presentation: Railway Net Intrinsics

A summary formalisation of the three narrated railway net intrinsics could be:

- *Potential train passengers:*

```
scheme N0 =
  class
    type
      N, L, S, Sn, Ln
    value
      obs_Ls: N → L-set, obs_Ss: N → S-set
      obs_Ln: L → Ln, obs_Sn: S → Sn
      obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
    axiom
      ...
  end
```

N, L, S, Sn and Ln designate nets, lines, stations, station names and line names. One can observe lines and stations from nets, line and station names from lines and stations, pair sets of station names from lines, and lines names (of lines) into and out from a station from stations. Axioms ensure proper graph properties of these concepts.

- *Actual train passengers:*

scheme N1 = **extend** N0 **with**
  class
    type
      Tr, Trn
    value
      obs_Trs: S → Tr-**set**, obs_Trn: Tr → Trn
    axiom
      ...
  end

The only additions are that of track and track name sorts, related observer functions and axioms.

- *Train operating staff:*

scheme N2 = **extend** N1 **with**
  class
    type
      U, C
      P′ = U × (C×C)
      P = {| p:P′ • **let** (u,(c,c′))=p **in** (c,c′)∈ ∪ obs_Ω(u) **end** |}
      Σ = P-**set**
      Ω = Σ-**set**
    value
      obs_Us: (N|L|S) → U-**set**
      obs_Cs: U → C-**set**
      obs_Σ: U → Σ
      obs_Ω: U → Ω
    axiom
      ...
  end

Unit and connector sorts have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers.

The reader is invited to compare the three narrative descriptions with the three formal descriptions, line by line. ▪

Different stakeholder perspectives, not only of intrinsics, as here, but of any facet, leads to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide with the name of a "similar" phenomenon of another perspective, that is, of another model, and so on. If the intention is that the "same" names cover comparable phenomena, then the developer must state the comparison relation.

Example 11.9 *Comparable Intrinsics:* We refer to Example 11.8. We claim that the concept of nets, lines and stations in the three models of Example 11.8 must relate. The simplest possible relationships are to let the third model be the common "unifier" and to mandate

- that the model of nets, lines and stations of the *potential train passengers* formalisation is that of nets, lines and stations of the *train operating staff* model; and
- that the model of nets, lines, stations and tracks of the *actual train passengers* formalisation is that of nets, lines, stations of the *train operating staff* model.

Thus the third model is seen as the definitive model for the stakeholder views initially expressed. ▪

In general the relationships to be expressed between different stakeholder models require more elaborate expressions. To express these formally, in RSL, we make use of RSL's *scheme* facility. We refer to Vol. 2, Chap. 10 (Modularisation) in which we cover the *scheme* concept of RSL (Sect. 10.2 (RSL Classes, Objects and Schemes) of that volume). More elaborate stakeholder schemes can be expressed by *extending* basic (i.e., intrinsic) schemes *with* additional types, values and axioms. The *hiding* facility of schemes can likewise be used to express different, but commensurate models.

The comparison relations are in this case quite simple, namely those of *conservative algebra inclusions*. One algebra is conservatively included in another algebra if all the entities and operations (etcetera) of the former are included in the latter, and hence if all theorems true of the former algebra hold in the latter.

In the above description such things as lines, stations and units, including their particular kind (linear, switch, etc.) are phenomena, that is, they can be pointed to. Such things as connectors and paths could be considered either phenomena or concepts. Unit states and unit state spaces, including the idea of open and closed units, will here be considered concepts. The above example is only indicative. Much care must be taken to ensure that a description is consistent and complete. Care must also be taken to not describe phenomena or concepts that more properly belong to some other facets, as covered next. Identifying and describing intrinsics is also an art!

Example 11.10 *Intrinsics of Switches:* The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch ($^{c|}Y^{c/}_{c}$) has three connectors: $\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$ (Fig. 11.5). So we have that a possible state space of such a switch could be ω_{g_s} :

$\{\{\},$
$\{(c, c_|)\}, \{(c_|, c)\}, \{(c, c_|), (c_|, c)\},$
$\{(c, c_/)\}, \{(c_/, c)\}, \{(c, c_/), (c_/, c)\}, \{(c_/, c), (c_|, c)\},$
$\{(c, c_|), (c_|, c), (c_/, c)\}, \{(c, c_/), (c_/, c), (c_|, c)\}, \{(c_/, c), (c, c_|)\}, \{(c, c_/), (c_|, c)\}\}$

The above models a general switch ideally. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$. Nothing is said about how a state is determined: who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (i.e., invisible, intangible) signals up or down the rail, away from the switch. ∎

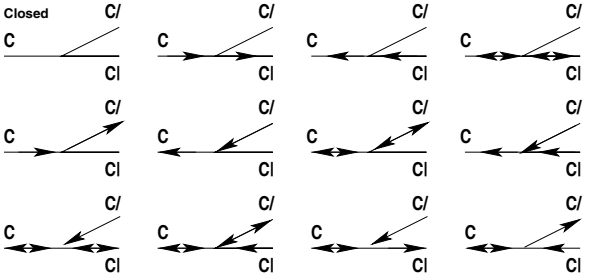


Fig. 11.5. Possible states of a rail switch

11.3.2 Conceptual Versus Actual Intrinsics

In order to bring an otherwise seemingly complicated domain across to the reader, one may decide to present it piecemeal:[8] First, one presents the very basics, the fewest number of inescapable entities, functions and behaviours. Then, in a step of enrichment, one adds a few more (intrinsic) entities, functions and behaviours. And so forth. In a final step one adds the last (intrinsic) entities, functions and behaviours. In order to develop what initially may seem to be a complicated domain, one may decide to develop it piecemeal: We basically do as for the presentation steps: Steps of enrichments — from a big lie, via increasingly smaller lies, till one reaches a truth!

[8] That seemingly complicated domain may seem very complicated, containing hundreds of entities, functions and behaviours. Instead of presenting all the entities, functions, events and behaviours in one "fell swoop", one presents them in stages: first, around seven such (entities, functions, events and behaviours), then seven more, etc.

Example 11.11 *Conceptual Intrinsics: Freight Transport:* The very essence of freight transport is: Entities: Senders, freight, "the system of transport", and receivers. Functions: submitting an item of freight for transport, and receiving an item of freight having been transported. Behaviour: Being transported.

Formal Presentation: Freight Transport

type
  Sndr, Frei, Rcvr
value
  submit: Sndr × Frei → System → System
  receiv: Rcvr → System → System × Frei
  transport: System → System

Observe that we have said nothing, really, about "the system of transport. ▪

Example 11.12 *Actual Intrinsics: Freight Logistics:* We now elaborate on "the system of transport" alluded to in Example 11.11. The system entities are: harbours, bills of lading, ships and ship routes (from harbours to harbours). We assume that there is no need to detail what are harbours, ships and ship routes. A bill of lading is a document, say attached to a piece of freight, which stipulates properties of the freight (sender, receiver, origin of transport, destination of transport and route of transport: sequence of harbours and ships, sailing times, etc.). The system functions are: submit a piece of freight to a harbour (of origin) indicating a receiver and a harbour of destination, and obtaining a bill of lading; load a piece of freight from a harbour to a ship, as prescribed by that freight's bill of lading; unload a piece of freight from a ship to a harbour, as prescribed by that freight's bill of lading; fetching, by a receiver, a piece of freight from a destination harbour, as prescribed by that freight's bill of lading. A system behaviour could be the sequence of one submission, one or more pairs of loadings and unloadings, ended by one fetch. The above behaviour has abstracted "away" any notion of sailings, i.e., of actual movement!

Formal Presentation: Freight Logistics

type
  Sndr, Sndr_Na, Frei, Rcvr, Rcvr_Na,
  Harb, H_Na, Ship, S_Na, System, BoL
  Dest = H_Na
value
  obs_Harbs: System → Harb-**set**
  obs_HNa: Harb → H_Na

obs_Route : BoL → (H_Na × S_Na)*
obs_Dest : BoL → HNa
obs_RcvrNa : BoL → Rcvr_Na
obs_RcvrNa : Rcvr → Rcvr_Na

submit: Sndr × Frei × Dest → System → BoL
load: Frei × BoL × Ship × Harb → Ship × Harb
unload: BoL × Ship × Harb → Ship × Harb × Frei
receiv: Rcvr → Harb → System → Frei × BoL
transp: System → System

The formalisation, as does the narrative, only rough-sketches some intrinsics of freight logistics. ▪

We leave the two versions, the virtual and the "more realistic", further undefined. Both descriptions were kept in the form of rough sketches. The latter can take being further refined, i.e., made more precise.

11.3.3 Methodological Consequences

Principles. In any modelling one first forms and describes *intrinsic* facets. ▪

Techniques. The *intrinsics* model of a domain is a partial specification. As such, it involves the use of well-nigh all description principles. Typically we resort to property-oriented models, i.e., sorts and axioms. ▪

11.3.4 Discussion

Thus the intrinsics become part of every one of the next facets. From an algebraic semantics point of view these latter are extensions of the above. We have presented a story of intrinsics as truthfully as we could. To decide on what is intrinsics and what is not is an art — it is a matter of choice, hence of style. There is no clear-cut criterion according to which a line of separation between intrinsics and nonintrinsics can be drawn.

11.3.5 Utter Barebones Intrinsics

It was implied above that an absolute barebones intrinsics of railways was the atomic trains and the rail net abstracted to atomic lines and atomic stations. Similarly one could claim that an absolute barebones intrinsics of a hospital system was the atomic patients, atomic medical staff and atomic beds. Without the beds the first two kinds of entities would pass only for a physician's

office. And similarly one could claim that an absolute barebones intrinsics for air traffic would be the aircraft, the airports and the air space. And so on.

The reason we bring this concept of *utter barebones intrinsics* up is threefold. First, the domain engineer must "think very hard" in trying to isolate, identify and capture the, or an utter barebones intrinsics of a domain. Secondly, the "more frugal" the domain engineer has been in selecting the utter barebones entities, functions, events and behaviours, the more time that domain engineer has to care about properly extending that utter barebones intrinsics with the remaining domain facets covered next. Thirdly, by "forcibly" trying to isolate an utter barebone intrinsics the domain engineer is actually trying to establish a scientific basis for the domain. The domain describer is more of a researcher than an engineer. This is basically untrodden land: few have tried to formulate domain descriptions let alone intrinsics, and very few, if any may have attempted to identify the utter barebones of a domain. We claim that it is a prerequisite for good domain descriptions to have tried to discover utter barebones intrinsics.

11.3.6 Reminder

We remind the reader of the principle stated at the outset of this section on domain intrinsics:

Principle. *Describing the Domain Intrinsics Facets:* So from the outset of describing a domain analyse it with respect to its intrinsic phenomena and concepts. Focus on describing these first, and make sure that the descriptions of all other (subsequently described) domain facets are subordinated descriptions of the domain intrinsics. ▪

11.4 Domain Support Technologies

Technology is meant to support human activities. Usually technology replaces human actions one to one, i.e., rather directly. (That is, for one human action kind there is usually a substitute technology.) In other instances technology radically transforms the ways in which things are done. Hence:

Principle. *Describing the Domain Support Technologies Facets:* When describing a domain analyse it with respect to its support technology phenomena and concepts, focus on possibly describing these separately, and make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain support technologies. ▪

11.4.1 Overall Principles

In Example 11.8, we implied that a switch may take on a number of states: linking, into paths, suitable pairs of connectors, or none. But how such states came about was abstracted (away).

Characterisation. By domain *support technology* we shall understand ways and means of implementing certain observed phenomena. ▪

The above characterisation is deliberately loose. It is so, so that we are not, later, constrained by a too tight characterisation. Therefore it is important to illustrate the idea, so as to aid the reader's intuition, and thus enable proper identification and description of support technologies.

Example 11.13 *Railway Support Technology:* We give a rough sketch description of possible rail unit switch technologies.

(i) In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.

(ii) With the advent of reasonably reliable mechanics, pulleys and levers[9] (and steel wires), switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).

(iii) This partial mechanical technology then emerged into electromechanics, and cabin tower staff was "reduced" to pushing buttons.

(iv) Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another).[10] ▪

It must be stressed that Example 11.13 is just a rough sketch. In a proper narrative description the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electromechanics and the human operator interface (buttons, lights, sounds, etc.).

An aspect of supporting technology includes recording the state-behaviour in response to external stimuli. We give an example.

Example 11.14 *Probabilistic Rail Switch Unit State Transitions:* Figure 11.6 indicates a way of formalising this aspect of a supporting technology.

[9] For pulley see: http://www.walter-fendt.de/ph11e/pulleysystem.htm. For lever see: http://www.edhelper.com/ReadingComprehension_24_90.html.

[10] In Vol. 2, Chap. 12, Petri nets, in Sect. 12.3.4 we exemplified this concept of interlocking by specifying a software design based on place transition nets. See also: http://irfca.org/faq/faq-signal4.html.

Figure 11.6 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and resettings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd. ∎

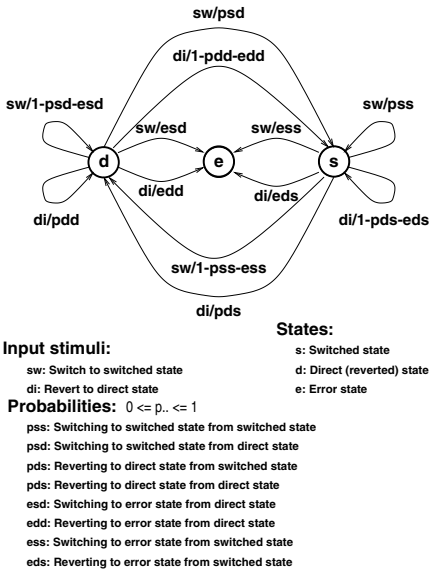


Fig. 11.6. Probabilistic state switching

Another example shows another aspect of support technology: Namely that the technology must guarantee certain of its own behaviours, so that software designed to interface with this technology, together with the technology, meets dependability requirements.

Example 11.15 *Railway Optical Gates:* Train traffic (itf:iTF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) trains (tn:TN).

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

We need to express quality criteria that any optical gate technology should satisfy — relative to a necessary and sufficient description of a closeness predicate. The following axiom does that:

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

Since units change state with time, n:N, the railway net, needs to be part of any model of traffic. We have defined railway nets in Example 11.8 (Sect. 11.3.1).

Formal Presentation: Railway Optical Gate Technology Requirements

```
type
  T, TN
  P = U*
  NetTraffic == net:N  trf:(TN ~m~> P)
  iTF = T → NetTraffic
  sTF = T ~m~> NetTraffic
  oG = iTF ~~> sTF
value
  [close] c: NetTraffic × TN × NetTraffic ~~> Bool
axiom
  ∀ itt:iTF, og:OG • let stt = og(itt) in
    ∀ t:T • t ∈ dom stt •
      t ∈ 𝒟 itt ∧ ∀ Tn:TN • tn ∈ dom trf(itt(t))
        ⇒ tn ∈ dom trf(stt(t)) ∧ c(itt(t),tn,stt(t)) end
```

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

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

The next example shows another aspect of the technology support facet. Example 11.15 relates any support technology to an intrinsics whose entity values that support technology was supposed to monitor. The next example, i.e., Example 11.16, again shows the relativeness of support technologies, as did Example 11.13.

Example 11.16 *Air Traffic Control:* We first refer to Example 11.4. Then we make the following remarks: The particular decomposition of air traffic control into the domain described, the ground, terminal, area and continental (monitoring and) control centres, represents but one composition of technologies. The pragmatics, i.e., the assumptions underlying that combined ground,

terminal, area and continental control centre support technology is that all monitoring and control was to take place from the ground. Future technologies, easily implementable today, facilitate the following alternative "sum total" technologies: Most, if not all, of the human guidance that today takes place at these control centres can be automated and physically moved either to fixed space-positioned satellites, or to each aircraft itself. Intermediate support technologies shall then feature solutions that are intermediary to the present and the future support technologies. ∎

11.4.2 Methodological Consequences

Techniques. The *support technologies* model of a domain is a partial specification, hence all the usual abstraction and modelling principles, techniques and tools apply. More specifically, support technologies (st:ST) "implement" intrinsic contexts and states: $\theta_i : \Theta_i$ in terms of "actual" contexts and states: $\theta_a : \Theta_a$:

type
  Θ_i, Θ_a
  ST = $\Theta_i \to \Theta_a$
axiom
  ∀ sts:ST-**set**, st:ST • st ∈ sts ⇒ ∀ θ_i:Θ_i, ∃ θ_a:Θ_a • st(θ_i) = θ_a

The formal requirements can be narrated: Let Θ_i and Θ_a designate the spaces of intrinsic and actual-world configurations (contexts and states).[11] For each intrinsic configuration model — that we know is support technology assisted — there exists a support technology solution, that is, a total function from all intrinsic configurations to corresponding actual configurations. If we are not convinced that there is such a function then there is little hope that we can trust this technology. ∎

Support technology is not a refinement, but an extension. Support technology typically introduces considerations of technology accuracy, reliability, fault tolerance, availability, accessability, safety, and so on. Axioms characterise members of the set of support technologies (sts). An example axiom was given in the optical gate example (Example 11.15). We shall have much (more) to say about support technologies, and the above dependability (etc.) issues, as we — much later in these volumes — move into machine requirements.

Principles. The *support technology* principle is relative to all other domain facets. It expresses that one must first describe essential intrinsics. Then it expresses that support technology is any means of implementing concrete instantiations of some intrinsics, of some management and organisation, and/or of some rules and regulations, and so on. ∎

[11] The concept of configurations, in terms of contexts and states, was treated in detail in Vol. 2, Chap. 4.

Generally the principle states that one must always be on the look out for and inspire new support technologies. The most abstract form of the principle is: *What is a support technology one day becomes part of the domain intrinsics a future day.*

11.4.3 Discussion

Skakkebæk et al. [337] exemplify the use of the duration calculus [381, 382] in describing supporting technologies that help achieve safe operation of a road-rail level crossing. This was exemplified very extensively in Vol. 2, Chap. 15, Sect. 15.3.6 (the road-rail level crossing example).

Chapters 12–15 of Vol. 2 cover a somewhat extensive variety of principles, techniques and tools for formally modelling support technologies. The support technology descriptions reappear in the requirements definitions: as projected, instantiated, extended and initialised (see Chap. 19). In the domain description we only record our understanding of aspects of support technology failures. In the requirements definition we then follow up and make decisions as to which kinds of breakdowns the computing system, the machine, is to handle, and what is to be achieved by such handling.

11.4.4 Reminder

We remind the reader of the principle stated at the outset of this section on domain support technologies:

Principle. *Describing the Domain Support Technologies Facets:* When describing a domain analyse it with respect to its support technology phenomena and concepts, focus on possibly describing these separately, and make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain support technologies. ▪

11.5 Domain Management and Organisation

It is a basic characteristic of human-made systems that they are managed by humans and that their management and the managed are structured in organisational structures. This section is about how we model this facet.

Principle. *Describing the Domain Management and Organisation Facets:* When describing a domain analyse it with respect to its management and organisation phenomena and concepts. Focus on possibly describing these separately, and make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain management and organisation. ▪

11.5.1 Overall Principles

Activities of some (application) domains are made up by the actions of many people. It is therefore common to organise these into levels of management and many groups of "floor", i.e., nonmanagement staff.

Railway systems are usually characterised by highly structured management organisations, and rules and regulations set up by upper echelons of management to be followed by lower levels and by ground staff and users.

Example 11.17 *Train Monitoring, I:* In China, as an example, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations ("up and down the lines"). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net. ▪

Characterisation. By domain *management* we shall understand such people (such decisions) (i) who (which) determine, formulate and thus set standards (cf. rules and regulations, Sect. 11.6) 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. ▪

In Example 9.2 (Chap. 9, Sect. 9.3.1) we illustrated the distinctions indicated in the above characterisation of management (item (i)) between strategies, tactics and operations.

Characterisation. By domain *organisation* we shall understand the structuring of management and nonmanagement staff levels; the allocation of strategic, tactical and operational concerns to within management and nonmanagement staff levels; and hence the "lines of command": who does what, and who reports to whom, administratively and functionally. ▪

Example 11.18 *Railway Management and Organisation: Train Monitoring, II:* We single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a

manager, that a train is not following its schedule, based on information monitored by nonmanagement staff, then that manager directs that staff: (i) to suggest a new schedule for the train in question, as well as for possibly affected other trains, (ii) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon, by the managers at respective stations, (iii) and to enact that new schedule.[12] A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts. ∎

The above, albeit rough-sketch description, illustrated the following management and organisation issues: There is a set of lowest-level (as here: train traffic scheduling and rescheduling) supervisors and their staff. They are organised into one such group (as here: per station). There is a middle-level (as here: regional train traffic scheduling and rescheduling) manager (possibly with some small staff), organised with one such per suitable (as here: railway) region. The guidelines issued jointly by local and regional (...) supervisors and managers imply an organisational structuring of lines of information provision and command.

11.5.2 A Conceptual Analysis, I

People staff enterprises, the components of infrastructures with which we are concerned, i.e., for which we develop software. The larger these enterprises — these infrastructure components — the more need there is for management and organisation. The role of management is roughly, for our purposes, twofold: first, to perform strategic, tactical and operational work, to set strategic, tactical and operational policies (cf. Example 9.2) — and to see to it that they are followed. The role of management is, second, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution.

Policy setting should help nonmanagement staff operate normal situations — those for which no management interference is thus needed. And management "backstops" problems: management takes these problems off the shoulders of nonmanagement staff.

To help management and staff know who's in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff has to turn to others for assistance they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams — the usually hierarchical box and arrow/line diagrams.

[12] That enactment may possibly imply the movement of several trains incident upon several stations: the one at which the manager is located, as well as possibly at neighbouring stations.

11.5.3 Methodological Consequences, I+II

Techniques. The *management and organisation* model of a domain is a partial specification; hence all the usual abstraction and modelling principles, techniques and tools apply. More specifically, management is a set of predicates, observer and generator functions which either parameterise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions. ▪

Organisation is thus a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modelled as sets (of recursively invoked sets) of equations. This was illustrated in Example 9.2.

11.5.4 Conceptual Analysis, II

To relate classical organigrams to formal descriptions we first show such an organigram (Fig. 11.7), and then we show schematic processes which — for a rather simple scenario — model managers and the managed!

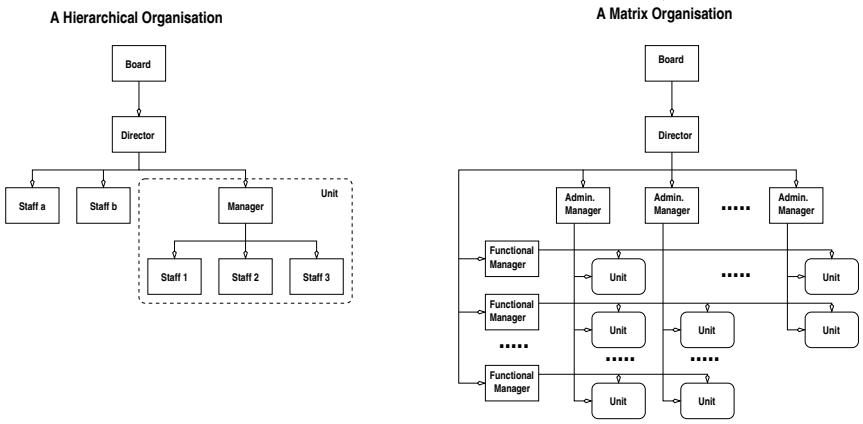


Fig. 11.7. Organisational structures

Based on such a diagram, and modelling only one neighbouring group of a manager and the staff working for that manager we get a system in which one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modelled by ψ. Each staff, stf(i) operates in a context and a state modelled by $s\sigma(i)$.

Formal Presentation: Conceptual Model of a Manager-Staff Relation, I

type

```
  Msg, Ψ, Σ, Sx
  SΣ = Sx -m-> Σ
channel
  { ms[i]:Msg | i:Sx }
value
  sσ:SΣ, ψ:Ψ

  sys: Unit → Unit
  sys() ≡ || { stf(i)(sσ(i)) | i:Sx } || mgr(ψ)
```

In this system the manager, mgr, (1) either broadcasts messages, msg, to all staff via message channel ms[i]. The manager's concoction, mgr_out(ψ), of the message, msg, has changed the manager state. Or (2) is willing to receive messages, msg, from whichever staff i the manager sends a message. Receipt of the message changes, mgr_in(i,msg)(ψ), the manager state. In both cases the manager resumes work as from the new state. The manager chooses — in this model — which of the two things (1 or 2) to do by a so-called nondeterministic internal choice ($\sqcap$).

Formal Presentation: Conceptual Model of a Manager-Staff Relation, II

```
  mgr: Ψ → in,out { ms[i] | i:Sx } Unit
  mgr(ψ) ≡
(1) (let (ψ',msg) = mgr_out(ψ) in
       || { ms[i]!msg | i:Sx } ; mgr(ψ') end)
    ⌈⌉
(2) (let ψ' = [] {let msg = ms[i]? in
       mgr_in(i,msg)(ψ) end | i:Sx } in mgr(ψ') end)

  mgr_out: Ψ → Ψ × MSG,
  mgr_in: Sx × MSG → Ψ → Ψ
```

And in this system, staff i, stf(i), (1) either is willing to receive a message, msg, from the manager, and then to change, stf_in(msg)(σ), state accordingly, or (2) to concoct, stf_out(σ), a message, msg (thus changing state) for the manager, and send it ms[i]!msg. In both cases the staff resumes work as from the new state. The staff member chooses — in this model — which of the two "things" (1 or 2) to do by a nondeterministic internal choice ($\sqcap$).

Formal Presentation: Conceptual Model of a Manager-Staff Relation, III

```
  stf: i:Sx → Σ → in,out ms[i] Unit
  stf(i)(σ) ≡
(1) (let msg = ms[i]? in stf(i)(stf_in(msg)(σ)) end)
```

```
    ⫿
(2) (let (σ′,msg) = stf_out(σ) in ms[i]!msg; stf(i)(σ′) end)

    stf_in: MSG → Σ → Σ,
    stf_out: Σ → Σ × MSG
```

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process nondeterministically, external choice, "alternates" between "broadcast"-issuing orders to staff and receiving individual messages from staff. Staff processes likewise nondeterministically, external choice, alternate between receiving orders from management and issuing individual messages to management.

The conceptual example also illustrates modelling stakeholder behaviours as interacting (here CSP-like [168, 301, 311]) processes.[13]

11.5.5 Methodological Consequences, III

The *strategic, tactical and operations resource management* example of Example 9.2 (Sect. 9.3.1) illustrated another management and organisation description pattern. It is based on a set of, in this case, recursive equations. Any way of solving these equations, finding a suitable fix point, or an approximation thereof, including just choosing and imposing an arbitrary "solution", reflects some management communication. The syntactic ordering of the equations — in this case a linear passing of enterprise results from upper equations onto lower equations — reflects some organisation.

Principles. The *management and organisation* principle expresses that relations between resources, and decisions to acquire and dispose resources, to deschedule, reschedule and schedule resources, to deallocate, reallocate and allocate resources and to deactivate, reactivate and activate resources, are the prerogatives of well-functioning management, reflect a functioning organisation and imply invocation of procedures that are modelled as actions that "set up" and "take down" contexts and change states. As such, these principles tell us which subproblems of development to tackle. ∎

Techniques. *Management and Organisation:* We have already, under techniques for modelling stakeholder and stakeholder perspectives, mentioned some of the techniques (cf. Sect. 9.3.1). Two extremes were shown: Earlier we modelled individual management groups by their respective functions (strm, trm, orm), and their interaction (i.e., organisation) by solutions to a set of recursive equations! Presently we modelled management and organisation, especially the latter, by communicating sequential behaviours. ∎

[13] We covered the use of CSP in Vol. 1, Chap. 21.

11.5.6 Discussion

The domain models of management and organisation eventually find their way into requirements and, hence, the software design — for those cases in which the requirements are about computing support of management and its organisation. Support in the solution of the recursive equations of the earlier stakeholder example (Example 9.2 Resource Management) may be offered in the form of constraint-satisfaction solvers [15]. These may partially handle logic characterisations of the strategic and tactical management functions. They might then do so in the form of computerised support of message passing between the various management groups (of, for example, that stakeholder example), as well as of the generic example of the present part.

11.5.7 Reminder

We remind the reader of the principle stated at the outset of this section on domain management and organisation:

Principle. *Describing the Domain Management and Organisation Facets:* When describing a domain analyse it with respect to its management and organisation phenomena and concepts. Focus on possibly describing these separately, and make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain management and organisation. ▪

11.6 Domain Rules and Regulations

Railway systems, for example, are characterised by large varieties of rules for appropriate behaviour of: trains, train dispatch, monitoring and control, supporting technology, and hence of humans at all levels. This is also true for most other systems that we might care to consider.

When rules are broken regulations take effect: Humans may be disciplined, and activities of the domain may be adjusted.

Principle. *Describing the Domain Rules and Regulations Facets:* When describing a domain analyse it with respect to its rules and regulations phenomena and concepts. Focus on possibly describing these separately, and make sure that the descriptions of other domain facets are commensurate with possibly multiple, alternative descriptions of domain rules and regulations. ▪

11.6.1 Overall Principles

Earlier, when we dealt with management and organisation, it was hinted that management may issue certain guidelines. We now look at a special class of these.

Characterisation. By a domain *rule* we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their function. ▪

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

Rules are like one part of a law: *Thou shalt!* Regulations are like another part of a law: *If you break this law "thou" can expect the following punishment!*

Rules and regulations are set by enterprises, by equipment manufacturers, by enterprise associations, by [government] regulatory agencies, and by society (the latter in the form of laws). Adherence to rules is likewise monitored by these or similar institutions. Enforcement of (i.e., the imposition of what is specified in) regulations is similarly ensured by these or similar institutions.

Example 11.19 *Trains at Stations:*

- Rule: In China the arrival and departure of trains at, respectively from, railway stations is subject to the following rule:
  In any three-minute interval at most one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

▪

Example 11.20 *Trains Along Lines:*

- Rule: In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:
  There must be at least one free sector (i.e., without a train) between any two trains along a line.[14]
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

▪

It is, as for all other domain facets, crucially important that rules and regulations are captured and precisely described — as we often shall find that requirements of software either assume these rules to hold, or expect such rules to be enforced.[15]

11.6.2 Methodological Consequences

Techniques. *Rules and Regulations:* At a metalevel, 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: 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.

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

Formal Explication: Conceptual Model of Rules, 1

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

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

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

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,

[14] In Vol. 2, Chap. 14, Sect. 14.4.1, an example, Automatic Line Blocking, illustrates how one might implement this rule.

[15] As, for example, in Sect. 14.4.1 of Vol. 2, Chap. 14.

which is a pair. This pair consists of a predicate, pre_reg:Pre_REG, where Pre_REG = $(\Theta \times \Theta) \rightarrow$ **Bool**, and a domain configuration-changing function, act_reg:Act_REG, where Act_REG = $\Theta \rightarrow \Theta$, that is, both involving current and next domain configurations. The two kinds of functions express: If the predicate holds, then the action can be applied.

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

Formal Explication: Conceptual Model of Regulations, 2

```
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
```

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.

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, $\sim$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.

Formal Explication: Conceptual Model of Rules and Regulations, 3

```
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))
```

$$\Rightarrow \exists\ \mathrm{n}\theta{:}\Theta \bullet \mathrm{act\_reg}(\theta){=}\mathrm{n}\theta \wedge \mathrm{se\_rul}(\theta,\mathrm{n}\theta)$$

end

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. ▪

We have given an outline of the basic conditions under which a set of rules and regulations must be designed. Whether they are, in actual life, designed, by people, and to be interpreted and followed by people, as described here is not for us to decide. Such concerns are the prerogatives of business process reengineering and domain requirements (Sects. 19.3 and 19.4).

11.6.3 Rules and Regulation Languages

We have outlined the basic properties any set of rules and regulations must imply in a properly functioning organisation. The axioms prescribed above are abstract. They also apply, inter alia, to natural language expressions of rules and regulations.

It would be nice if rules and regulations could be formalised. Then, given an appropriate model of the domain, one might be able to analyse the consistency and completeness of rules and regulations with respect to the domain model.

It is inside the scope, but outside the span of this book to bring in — as of 2006 — research material on this subject. In other words: Expect it to come, one day, probably couched in terms of some modal logics of knowledge and belief, and/promise and commitment, etc. We refer to the nice book by Fagin, Halpern, Moses and Vardi: *Reasoning About Knowledge* [98].

Essentially, the issues are: first, to design and use languages (one or more, Rul, Reg), with proper, possibly modal constructs, for expressing rules and regulations. Second, we need to compile such expressions of rules and regulations. Finally, we need to let a computer check "all the time" whether stimuli (whether human or otherwise generated) might cause transitions that may result in violations of the rules.

11.6.4 Principles and Techniques

Principle. *Rules and Regulations:* Domains are governed by rules and regulations: by laws of nature or edicts by humans. Laws of nature can be part of intrinsics, or can be modelled as rules and regulations constraining the intrinsics. Edicts by humans usually change, but are normally considered part of an irregularly changing context, not a recurrently changing state. Modelling techniques follow these principles. ▪

Techniques. *Rules and regulations*, in the domain, are therefore domain-modelled by abstract or concrete syntaxes of syntactic rules, by abstract types of denotations and by semantics definitions, usually in the form of axioms or denotation-ascribing functions. Such rules and regulations modelling must allow for conflicts between rule and regulation interpretations: that rules are interpreted to state that a next configuration is not valid, while a regulation (applicability) predicate does not hold. Stimuli, without here going into details, may be modelled by nondeterministic external events, i.e., CSP-like inputs. ∎

11.6.5 Reminder

We remind the reader of the principle stated at the outset of this section on domain rules and regulations:

Principle. *Describing the Domain Rules and Regulations Facets:* When describing a domain analyse it with respect to its rules and regulations phenomena and concepts. Focus on possibly describing these separately, and make sure that the descriptions of other domain facets are commensurate with possibly multiple, alternative descriptions of domain rules and regulations. ∎

11.7 Domain Scripts

Usually rules and regulations form a contract between levels of staff in an enterprise. We may call these intrainstitutional rules and regulations. Rules that pertain to contracts between, say, a private enterprise and its customers, or a government and its citizens, often need be far more stringently phrased than intrainstitutional rules and regulations. We may call such rules legal rules and regulations. Legal rules and regulations often need be scripted.

Principle. *Describing the Domain Script Facets:* When describing a domain analyse it with respect to its script phenomena and concepts. Focus on possibly describing these separately, and make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain scripts. ∎

11.7.1 The Description of Scripts

Characterisation. By a domain *script* we shall understand the structured, almost, if not outright, formally expressed, wording of a rule or a regulation that has legally binding power, that is, which may be contested in a court of law. ∎

Scripts are like programs. They are expected to prescribe step-by-step actions to be applied in order to determine whether a rule should be applied, and, if so, exactly how it should be applied.

Example 11.21 *A Casually Described Bank Script, I:* We deviate, momentarily, from our line of railway examples, to exemplify one from banking. Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborate on the theme of (bank) scripts.

The problem area is that of how repayments of mortgage loans are to be calculated. At any one time a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, then the following calculations shall take place: (i) the interest on the balance of the loan since the most recent repayment, (ii) the handling fee, normally considered fixed, (iii) the effective repayment — being the difference between the repayment and the sum of the interest and the handling fee — and the new balance, being the difference between the old balance and the effective repayment.

We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts.

(i) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (ii) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (iii) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. ∎

The idea about scripts is that they can somehow be objectively enforced: that they can be precisely understood and consistently carried out by all stakeholders, eventually leading to computerisation. But they are, at all times, part of the domain.

In the next example we systematically describe a bank, informally and formally. The formal description is in the classical style of semantics. Each formal description is followed by an informal, almost rough-sketch description. You may consider the latter to be in some casual script language. Example 11.23 then attempts a formalisation of the rough-sketch scripts into a "bank-friendly" script.

Example 11.22 *Bank Scripts, II:* Without much informal explanation, i.e., narrative, we define a small bank, small in the sense of offering but a few services. One can open and close demand/deposit accounts. One can obtain and close mortgage loans, i.e., obtain loans. One can deposit into and withdraw from demand/deposit accounts. And one can make payments on the loan. In

this example we illustrate informal rough-sketch scripts while also formalising these scripts.

In the following we first give the formal specification, then a rough-sketch script. You may prefer to read the pairs, formal specification and rough-sketch script, in the reverse order.

Bank State

Formal Presentation: Bank State

type
C, A, M
AY′ = **Real**, AY = {| ay:AY′ • 0<ay≤10 |}
MI′ = **Real**, MI = {| mi:MI′ • 0<mi≤10 |}
Bank′ = A_Register × Accounts × M_Register × Loans
Bank = {| β:Bank′ • wf_Bank(β)|}
A_Register = C $\overrightarrow{m}$ A-**set**
Accounts = A $\overrightarrow{m}$ Balance
M_Register = C $\overrightarrow{m}$ M-**set**
Loans = M $\overrightarrow{m}$ (Loan × Date)
Loan,Balance = P
P = **Nat**

There are clients (c:C), account numbers (a:A), mortgage number (m:M), account yields (ay:AY), and mortgage interest rates (mi:MI). The bank registers, by client, all accounts (ρ:A_Register) and all mortgages (μ:M_Register). To each account number there is a balance (α:Accounts). To each mortgage number there is a loan (ℓ:Loans). To each loan is attached the last date that interest was paid on the loan.

State Well-formedness

Formal Presentation: State Well-formedness

value
ay:AY, mi:MI

wf_Bank: Bank → **Bool**
wf_Bank(ρ,α,μ,ℓ) ≡ ∪ **rng** ρ = **dom** α ∧ ∪ **rng** μ = **dom** ℓ
axiom
ai<mi

We assume a fixed yield, ai, on demand/deposit accounts, and a fixed interest, mi, on loans. A bank is well-formed if all accounts named in the

accounts register are indeed accounts, and all loans named in the mortgage register are indeed mortgages. No accounts and no loans exist unless they are registered.

Client Transactions

Formal Presentation: Syntax of Client Transactions

```
type
  Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
  OpA == mkOA(c:C)
  CloA == mkCA(c:C,a:A)
  Dep == mkD(c:C,a:A,p:P)
  Wdr == mkW(c:C,a:A,p:P)
  OpM == mkOM(c:C,p:P)
  Pay == mkPM(c:C,a:A,m:M,p:P)
  CloM == mkCM(c:C,m:M,p:P)
  Reply = A | M | P | OkNok
  OkNok == ok | notok
```

The client can issue the following commands: Open Account, Close Account, Deposit monies (p:P), Withdraw monies (p:P), Obtain loans (of size p:P) and Pay installations on loans (by transferring monies from an account). Loans can be Closed when paid down.

Open Account Transaction

Formal Presentation: Semantics of Open Account Transaction

```
value
  int_Cmd: Cmd → Bank → Bank × Reply

  int_Cmd(mkOA(c))(ρ,α,μ,ℓ) ≡
    let a:A • a ∉ dom α in
    let as = if c ∈ dom ρ then ρ(c) else {} end ∪ {a} in
    let ρ′ = ρ † [c↦as],
        α′ = α ∪ [a↦0] in
    ((ρ′,α′,μ,ℓ),a) end end end
```

When opening an account the new account number is registered and the new account set to 0. The client obtains the account number.

Close Account Transaction

Formal Presentation: Semantics of Close Account Transaction

```
int_Cmd(mkCA(c,a))(ρ,α,μ,ℓ) ≡
  let ρ′ = ρ † [c↦ρ(c)\{a}],
      α′ = α \ {a} in
  ((ρ′,α′,μ,ℓ),α(a)) end
  pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

When closing an account the account number is deregistered, the account is deleted, and its balance is paid to the client. It is checked that the client is a bona fide client and presents a bona fide account number. The well-formedness condition on banks secures that if an account number is registered then there is also an account of that number.

Deposit Transaction

Formal Presentation: Semantics of Deposit Transaction

```
int_Cmd(mkD(c,a,p))(ρ,α,μ,ℓ) ≡
  let α′ = α † [a↦α(a)+p] in
  ((ρ,α′,μ,ℓ),ok) end
  pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

When depositing into an account that account is increased by the amount deposited. It is checked that the client is a bona fide client and presents a bona fide account number.

Withdraw Transaction

Withdrawing monies can only occur if the amount is not larger than that deposited in the named account. Otherwise the amount, p:P, is subtracted from the named account. It is checked that the client is a bona fide client and presents a bona fide account number.

Formal Presentation: Semantics of Withdraw Transaction

```
int_Cmd(mkW(c,a,p))(ρ,α,μ,ℓ) ≡
  if α(a)≥p
    then
      let α′ = α † [a↦α(a)−p] in
      ((ρ,α′,μ,ℓ),p) end
    else
      ((ρ,α,μ,ℓ),nok)
```

```
    end
    pre c ∈ dom ρ ∧ a ∈ dom α
```

Open Mortgage Account Transaction

Formal Presentation: Semantics of Open Mortgage Account Transaction

```
int_Cmd(mkOM(c,p))(ρ,α,μ,ℓ) ≡
    let m:M • m ∉ dom ℓin
    let ms = if c ∈ dom μ then μ(c) else {} end ∪ {m} in
    let mu′ = μ † [c↦ms],
        α′ = α † [a_ℓ↦α(a_ℓ)−p],
        ℓ′ = ℓ ∪ [m↦p] in
    ((ρ,α′,μ′,ℓ′),m) end end end
```

To obtain a loan, p:P, is to open a new mortgage account with that loan (p:P) as its initial balance. The mortgage number is registered and given to the client. The loan amount, p, is taken from a specially designated bank capital acount, a_ℓ. The bank well-formedness condition should be made to reflect the existence of this account.

Close Mortgage Account Transaction

Formal Presentation: Semantics of Close Mortgage Account Transaction

```
int_Cmd(mkCM(c,m))(ρ,α,μ,ℓ) ≡
    if ℓ(m) = 0
        then
            let μ′ = ρ † [c↦μ(c) \ {m}],
                ℓ′ = ℓ \ {m} in
            ((ρ,α,μ′,ℓ′),ok) end
        else
            ((ρ,α,μ,ℓ),nok)
    end
    pre c ∈ dom μ ∧ m ∈ μ(c)
```

One can only close a mortgage account if it has been paid down (to 0 balance). If so, the loan is deregistered, the account removed and the client given an OK. If not paid down the bank state does not change, but the client is given a NOT OK. It is checked that the client is a bona fide loan client and presents a bona fide mortgage account number.

Loan Payment Transaction

Formal Presentation: Semantics of Loan Payment Transaction

To pay off a loan is to pay the interest on the loan since the last time interest was paid. That is, interest, i, is calculated on the balance, b, of the loan for the period $d' - d$, at the rate of mi. (We omit defining the interest computation.) The payment, p, is taken from the client's demand/deposit account, a; i is paid into a bank (interest earning account) a_i and the loan is diminished with the difference $p-i$. It is checked that the client is a bona fide loan client and presents a bona fide mortgage account number. The bank well-formedness condition should be made to reflect the existence of account a_i.

int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) ≡
  let (b,d) = ℓ(m) **in**
  if α(a)≥p
    then
      let i = interest(mi,b,d′−d),
        ℓ' = ℓ † [m↦ℓ(m)−(p−i)]
        α' = α † [a↦α(a)−p,a_i↦α(a_i)+i] **in**
      ((ρ,α',μ,ℓ'),ok) **end**
    else
      ((ρ,α',μ,ℓ),nok)
  end end
  pre c ∈ **dom** μ ∧ m ∈ μ(c)

This ends the first stage of the development of a script language. ∎

Example 11.22 gave the formal description of banking transactions and their informal, rough-sketch script counterparts. We now "derive", without much further ado, pseudo-formal "bank-friendly" scripts.

Example 11.23 *Bank Scripts, III:* From each of the informal/formal bank script descriptions we systematically "derive" a script in a possible bank script language. The derivation, for example, for how we get from the formal descriptions of the individual transactions to the scripts in the "formal" bank script language is not formalised. In this example we simply propose possible scripts in the formal bank script language.

Open Account Transaction

Formal Presentation: Open Account Transaction

```
value
  int_Cmd(mkOA(c))(ρ,α,μ,ℓ) ≡
    let a:A • a ∉ dom α in
    let as = if c ∈ dom ρ then ρ(c) else {} end ∪ {a} in
    let ρ′ = ρ † [c↦as],
        α′ = α ∪ [a↦0] in
    ((ρ′,α′,μ,ℓ),a) end end end
```

Derived Bank Script: Open Account Transaction

```
routine open_account(c in "client",a out "account") ≡
    do
      register c with new account a ;
      return account number a to client c
    end
```

Close Account Transaction

Formal Presentation: Close Account Transaction

```
  int_Cmd(mkCA(c,a))(ρ,α,μ,ℓ) ≡
    let ρ′ = ρ † [c↦ρ(c)\{a}],
        α′ = α \ {a} in
    ((ρ′,α′,μ,ℓ),α(a)) end
    pre c ∈ dom ρ ∧ a ∈ ρ(c)
```

Derived Bank Script: Close Account Transaction

```
routine close_account(c in "client",a in "account" out "monies") ≡
    do
      check that account client c is registered ;
      check that account a is registered with client c ;
      if
        checks fail
          then
            return NOT OK to client c
          else
            do
              return account balance a to client c ;
```

```
            delete account a
        end
    fi
end
```

Deposit Transaction

Formal Presentation: Deposit Transaction

int_Cmd(mkD(c,a,p))(ρ,α,μ,ℓ) $\equiv$
  let α' = α † [a$\mapsto\alpha$(a)+p] **in**
  ((ρ,α',μ,ℓ),ok) **end**
  pre c $\in$ **dom** ρ $\wedge$ a $\in$ ρ(c)

Derived Bank Script: Deposit Transaction

```
routine deposit(c in "client",a in "account",ma in "monies") ≡
    do
        check that account client c is registered ;
        check that account a is registered with client c ;
        if
            checks fail
                then
                    return NOT OK to client c
                else
                    do
                        add ma to account a ;
                        return OK to client c
                    end
        fi
    end
```

Withdraw Transaction

Formal Presentation: Withdraw Transaction

int_Cmd(mkW(c,a,p))(ρ,α,μ,ℓ) $\equiv$
  if α(a)$\geq$p
    then
      let α' = α † [a$\mapsto\alpha$(a)$-$p] **in**
      ((ρ,α',μ,ℓ),p) **end**
    else

```
            ((ρ,α,μ,ℓ),nok)
      end
      pre c ∈ dom ρ ∧ a ∈ dom α
```

Derived Bank Script: Withdraw Transaction

```
routine withdraw(c in "client",a in "account",
                                  ma in "amount" out "monies") ≡
      do
         check that account client c is registered ;
         check that account a is registered with client c ;
         check that account a has ma or more balance;
         if
            checks fail
               then
                  return NOT OK to client c
               else
                  do
                     subtract ma from account a ;
                     return ma to client c
                  end
         fi
      end
```

Obtain Loan Transaction

Formal Presentation: Obtain Loan Transaction

```
int_Cmd(mkOM(c,p))(ρ,α,μ,ℓ) ≡
   let m:M • m ∉ dom ℓin
   let ms = if c ∈ dom μ then μ(c) else {} end ∪ {m} in
   let mu′ = μ † [c↦ms],
       α′ = α † [a_ℓ↦α(a_ℓ)−p],
       ℓ′ = ℓ ∪ [m↦p] in
   ((ρ,α′,μ′,ℓ′),m) end end end
```

Derived Bank Script: Obtain Loan Transaction

```
routine get_loan(c in "client",p in "amount",m out "loan number") ≡
      do
         register c with loan m amount p;
         subtract p from account bank's loan capital
```

```
      return loan number m to client c
   end
```

Close Loan Transaction

Formal Presentation: Close Loan Transaction

```
int_Cmd(mkCM(c,m))(ρ,α,μ,ℓ) ≡
   if ℓ(m) = 0
      then
         let μ' = ρ † [c↦μ(c)\{m}],
             ℓ' = ℓ \ {m} in
         ((ρ,α,μ',ℓ'),ok) end
      else
         ((ρ,α,μ,ℓ),nok)
   end
   pre c ∈ dom μ ∧ m ∈ μ(c)
```

Derived Bank Script: Close Loan Transaction

```
routine close_loan(c in "client",m in "loan number") ≡
   do
      check that loan client c is registered ;
      check that loan m is registered with client c ;
      check that loan m has 0 balance;
      if
         checks fail
            then
               return NOT OK to client c
            else
               do
                  close loan m
                  return OK to client c
               end
      fi
   end
```

Loan Payment Transaction

Formal Presentation: Loan Payment Transaction

```
int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) ≡
  let (b,d) = ℓ(m) in
  if α(a)≥p
    then
      let i = interest(mi,b,d′−d),
          ℓ′ = ℓ † [m↦ℓ(m)−(p−i)]
          α′ = α † [a↦α(a)−p,a_i↦α(a_i)+i] in
      ((ρ,α′,μ,ℓ′),ok) end
    else
      ((ρ,α′,μ,ℓ),nok)
  end end
  pre c ∈ dom μ ∧ m ∈ μ(c)
```

Derived Bank Script: Loan Payment Transaction

```
routine pay_loan(c in "client",m in "loan number",p in "amount") ≡
    do
      check that loan client c is registered ;
      check that loan m is registered with client c ;
      check that account a is registered with client c ;
      check that account a has p or more balance ;
      if
        checks fail
          then
            return NOT OK to client c
          else
            do
              compute interest i for loan m on date d ;
              subtract p−i from loan m ;
              subtract p from account a ;
              add i to account bank's interest
              return OK to client c ;
            end
      fi
    end
```

This ends the second stage of the development of a script language. ∎

From the sketch attempts of bank-friendly scripts we establish, in the next example, a syntax for the bank-friendly script language.

Example 11.24 *Bank Scripts, IV:* We now examine the proposed scripts. Our objective is to design a syntax for the language of bank scripts. First, we list the statements as they appear in Example 11.23, except for the first two statements.

Routine Headers

We first list all routine "headers":

open_account(c **in** ″`client`″,a **out** ″`account`″)
close_account(c **in** ″`client`″,a **in** ″`account`″ **out** ″`monies`″)
deposit(c **in** ″`client`″,a **in** ″`account`″,ma **in** ″`monies`″)
withdraw(c **in** ″`client`″,a **in** ″`account`″,ma **in** ″`amount`″ **out** ″`monies`″)
get_loan(c **in** ″`client`″,p **in** ″`amount`″,m **out** ″`loan number`″)
close_loan(c **in** ″`client`″,m **in** ″`loan number`″)
pay_loan(c **in** ″`client`″,m **in** ″`loan number`″,p **in** ″`amount`″)

We then schematise a routine "header":

routine name(v1 io ″`t`″,v2 io ″`t2`″,...,vn io ″`tn`″) ≡

where:

io = **in** | **out**

and:

ti is any text

Example Statements

do stmt_list ***end***
if test_expr ***then*** stmt ***else*** stmt ***fi***

register c ***with new account*** a
register c ***with loan*** m ***amount*** p

add p ***to account*** a
subtract p ***from account*** a
subtract p−i ***from loan*** m
add i ***to account bank's interest***
subtract p ***from account bank's loan capital***

add p ***to account bank's loan capital***
compute interest i ***for loan*** m ***on date*** d

delete account a
close loan m

return ret_expr ***to client*** c
check that check_expr

The interest variable i is a ***local*** variable. The date variable d is an "oracle" (see below), but will be treated as a ***local*** variable.

Example Expressions

test_expr:

checks fail

ret_expr:

account number a
account balance a
NOT OK
OK
p
loan number m

check_expr:

account client c ***is registered***
account a ***is registered with client*** c
account a ***has*** p ***or more balance***
loan client c ***is registered***
loan m ***is registered with client*** c
loan m ***has*** 0 ***balance***

Abstract Syntax for Syntactic Types

We analyse the above concrete schemas (i.e., examples). Our aim is to find a reasonably simple syntax that allows the generation of the scripts of Example 11.23. After some experimentation we settle on the syntax shown next.

Formal Presentation: Bank Script Language Syntax

type

```
RN, V, C, A, M, P, I, D

Routine = Header × Clause

   Header == mkH(rn:RN,vdm:(V ⇒m (IOL × Text)))
   IOL == in | out | local

Clause = DoEnd | IfThEl | Return | RegA | RegL | Check
         | Add | Sub | 2Sub | DelA | DelM | ComI | RetE |

   DoEnd == mkDE(cl:Clause*)
   IfThEl == mkITE(tex:Test_Expr,cl:Clause,cl:Clause)

   Return == mkR(rex:Ret_Expr,c:V)
   RegA == mkRA(c:V,a:V)
   RegL == mkRL(c:V,m:V,p:V)
   Chk = mkC(cex:Chk_Expr)
   Add == mkA(p:V,t:(V|BA))
   Sub == mkS(p:V,t:(V|BA))
   2Sub == mk2S(p:V,i:V,t:(AN|MN|BA))
      AN == mkAN(a:V)
      MN == mkMN(m:V)
      BA == bank_i | bank_c
   DelA == mkDA(c:V,a:V)
   DelM == mkDM(c:V,m:V)
   Comp == mkCP(m:V,fn:Fn,argl:(V|D)*)

      Fn == interest | ...

Test_Expr = mkTE()

Chk_Expr == CisAReg(c:V) | AisReg(a:V,c:V) | AhasP(a:V,p:V)
            |  CisMReg(c:V) | MisReg(m:V,c:V) | Mhas0(m:V)

RetE == mkAN(a:V)|mkAB(a:V)|ok|nok|mkP(p:V)|mkMN(m:V)
```

■

Finally, in the next example, we establish a formal semantics of the bank-friendly script language. The reader is asked to compare the semantic types of the bank-friendly script language of Example 11.25 with the semantic types of Example 11.22.

Example 11.25 *Bank Scripts, V:*

Formal Presentation: Semantics of Bank Script Language

We now give semantics to the bank script language of Example 11.24.

Semantic Types Abstract Syntax

type
V, C, A, M, P, I
type
AY′ = **Real**, AY = {| ay:AY′ • 0<ay≤10 |}
MI′ = **Real**, MI = {| mi:MI′ • 0<mi≤10 |}
Bank′ = A_Register × Accounts × M_Register × Loans
Bank = {| β:Bank′ • wf_Bank(β)|}
A_Register = C $\overrightarrow{m}$ A-**set**
Accounts = A $\overrightarrow{m}$ Balance
M_Register = C $\overrightarrow{m}$ M-**set**
Loans = M $\overrightarrow{m}$ (Loan × Date)
Loan,Balance = P
P = **Nat**
Σ = (V $\overrightarrow{m}$ (C|A|M|P|I)) $\bigcup$ (Fn $\overrightarrow{m}$ FCT)
FCT = (...|Date)* → Bank → (P|...)
value
a_ℓ,a_i:A
axiom
∀ (ρ,α,μ,ℓ):B {a_ℓ,a_i} ⊆ **dom** α

The only difference between the above semantics types and those of Example 11.23 is the Σ state. The purpose of this auxiliary bank state component is to provide (i) a binding between the (always fixed) formal parameters of the script routines and the actual arguments given by the bank client or bank clerk when invoking any one of the routines, and (ii) a binding of a variety of "primitive", fixed, banking functions, FCT, named Fn, like computing the interest on loans, etc.

Semantic Functions

channel
k:(C|A|M|P|**Text**), d:Date

There is, in this simplifying example, one channel, k, between the bank and the client. It transfers text messages from the bank to the client, and client names (c:C), client account numbers (a:A), client mortgage numbers (m:M), and amount requests and monies (p:P) from the client to the bank. There

is also a "magic", a demonic channel, d, which connects the bank to a date "oracle".

```
value
  date: Date → out d  Unit
  date(da) ≡ (d!da ; date(da+Δ))
```

Each routine has a header and a clause. The purpose of the header is to initialise the auxiliary state component σ to appropriate bindings of formal routine parameters to actual, client-provided arguments. Once initialised, interpretation of the routine clause can take place.

```
  int_Routine: Routine → Bank → out k  Bank×Σ
  int_Routine(hdr,cla)(β) ≡
    let σ = initialise(hdr)([ ]) in
    Int_Clause(cla)(σ)(true)(β) end
```

For each formal parameter used in the body, i.e., in the clause, of the routine, there is a formal parameter definition in the header, and only for such. We have not expressed the syntactic well-formedness condition — but leave it as an exercise to the reader. And for each such formal parameter of the header a binding has now to be initially established. Some define input arguments, some define local variables and the rest define, i.e., name, output results. For each input argument the meaning of the header therefore specifies that an interaction is to take place, with the environment, as here designated by channel k, in order to obtain the actual value of that argument.

```
  initialise: Header → Σ → out,in k  Σ
  initialise(hdr)(σ) ≡
    if hdr = [ ]
      then σ
      else
        let v:V • v ∈ dom hdr in
        let (iol,txt) = hdr(v) in
        let σ′ =
          case iol of
            in → k!txt ; σ ∪ [v ↦ k?],
            _  → σ ∪ [v ↦ undefined]
          end in
        initialise(hdr\{v})(σ′)
    end end end end
```

In general, a clause is interpreted in a configuration consisting of three parts: (i) the local, auxiliary state, $\sigma : \Sigma$, which binds routine formal parameters to their values; (ii) the current 'check' state, tf:Check, which records the "sum

total", i.e., the conjunction status of the check commands so far interpreted, i.e., initially tf = **true**; and (iii) the proper bank state, β:Bank, exactly as also defined and used in Example 11.23. The result of interpreting a clause is a configuration: (Σ×Check×Bank).

```
type
  Check = Bool
value
  Int_Clause: Clause→Σ→Check→Bank→out k,in d (Σ×Check×Bank)
```

A **do ... end** clause is interpreted by interpreting each of the clauses within the clauses in the **do ... end** clause list, and in their order of appearance. The result of a check clause is "anded" (conjoined) to the current tf:Check status.

```
  Int_Clause(mkDE(cll))(σ)(tf)(β) ≡
    if cll = ⟨⟩
      then (σ,tf,β)
      else
        let (σ′,tf′,β′) = Int_Clause(hd cll)(σ)(tf)(β) in
        Int_Clause(mkDE(tl cll))(σ′)(tf∧tf′)(β′)
    end end
```

if ... then ... else fi clauses only test the current check status (and propagate this status).

```
  Int_Clause(mkITE(tex,ccl,acl))(σ)(tf)(β) ≡
    if tf
      then
        Int_Clause(ccl)(σ)(true)(β)
      else
        Int_Clause(acl)(σ)(false)(β)
    end
```

Interpretation of a **return** clause does not change the configuration "state". It only leads to an output, to the environment, via channel k, of a return value, and as otherwise directed by any of the six return expressions (rex).

```
  Int_Clause(mkRet(rex))(σ)(tf)(ρ,α,μ,ℓ) ≡
    k!(case rex of
        mkAN(a)
          → "Your new account number:" σ(a),
        mkAB(a)
          → "Your account balance paid out:" α(a),
        mkP(p)
```

```
            → "Monies withdrawn:" σ(p),
        mkMN(m)
            → "Your loan number:" σ(m),
        OK
            → "Transaction was successful",
        NOK
            → "Transaction was not successful"
    end);
(σ,true,(ρ,α,μ,ℓ))
```

Interpretation of a **register account** clause is as you would expect from Example 11.23 — anything else would "destroy" the whole purpose of having a bank script. That purpose is, of course, to effect basically the same as the not yet "script-ised" semantics of Example 11.23.

```
Int_Clause(mkRA(c,a))(σ)(tf)(ρ,α,μ,ℓ) ≡
  let av:A • av ∉ dom α in
  let σ′ = σ † [a ↦ av],
      as = if c ∈ dom ρ then ρ(c) else {} end,
      ρ′ = ρ † [c ↦ as ∪ {av}],
      α′ = α ∪ [av ↦ 0] in
  (σ′,tf,(ρ′,α′,μ,ℓ))
  end end
```

The same holds for the **register loan** clause (as for the **register account** clause).

```
Int_Clause(mkRL(c,m,p))(σ)(tf)(ρ,α,μ,ℓ) ≡
  let mv:M • mv ∉ dom ℓin
  let σ′ = σ † [m ↦ mv],
      ms = if c ∈ dom μ then μ(c) else {} end,
      μ′ = μ † [c ↦ ms ∪ {mv}],
      ℓ′ = ℓ ∪ [mv ↦ p] in
  (σ′,tf,(ρ,α,μ′,ℓ′))
  end end
```

It can be a bit hard to remember the "meaning" of the mnemonics, so we repeat them here in another form:

- CisAReg: Client named in c is registered:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\ \rho$.
- AisReg: Client named in c has account named in a:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\rho \wedge \sigma(\sigma(\mathsf{a})) \in \rho(\sigma(\mathsf{c}))$.
- AhasP: Account named in a has at least the balance given in p:
  $\alpha(\sigma(\mathsf{a})) \geq \sigma(\mathsf{p})$.

- CisMReg: Client named in c has a mortgage:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\ \mu$.
- MisReg: Client named in c has mortgage named in m:
  $\sigma(\mathsf{c}) \in \mathbf{dom}\mu \wedge \sigma(\mathsf{m}) \in \mu(\sigma(\mathsf{c}))$.
- Mhas0: Mortgage named in m is paid up fully:
  $\ell(\sigma(\mathsf{m}))=0$.

Then it should be easier to "decipher" the logics:

$$
\begin{array}{l}
\mathrm{Int\_Clause}(\mathrm{mkChk}(\mathrm{cex}))(\sigma)(\mathrm{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad (\sigma,\mathbf{case}\ \mathrm{cex}\ \mathbf{of} \\
\qquad \mathrm{CisAReg(c)} \to \sigma(\mathrm{c}) \in \mathbf{dom}\ \rho, \\
\qquad \mathrm{AisReg(a,c)} \to \sigma(\mathrm{c}) \in \mathbf{dom}\rho \wedge \sigma(\sigma(\mathrm{a})) \in \rho(\sigma(\mathrm{c})), \\
\qquad \mathrm{AhasP(a,p)} \to \alpha(\sigma(\mathrm{a})) \geq \sigma(\mathrm{p}), \\
\qquad \mathrm{CisMReg(c)} \to \sigma(\mathrm{c}) \in \mathbf{dom}\ \mu, \\
\qquad \mathrm{MisReg(m,c)} \to \sigma(\mathrm{c}) \in \mathbf{dom}\mu \wedge \sigma(\mathrm{m}) \in \mu(\sigma(\mathrm{c})), \\
\qquad \mathrm{Mhas0(m)} \to \ell(\sigma(\mathrm{m}))=0 \\
\quad \mathbf{end},(\rho,\alpha,\mu,\ell))
\end{array}
$$

There are a number of ways of adding amounts, designated in p, to accounts and mortgages:

- mkAN(a): to account named in a
- mkMN(m): to mortgage named in m
- bank_i: to the bank's own interest account
- bank_c: to the bank's own capital account

$$
\begin{array}{l}
\mathrm{Int\_Clause}(\mathrm{mkA(p,t)})(\sigma)(\mathrm{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad \mathbf{case}\ \mathrm{t}\ \mathbf{of} \\
\qquad \mathrm{mkAN(a)} \to (\sigma,\mathbf{true},(\rho,\alpha\dagger[\mathrm{a} \mapsto \alpha(\sigma(\mathrm{a}))+\sigma(\mathrm{p})],\mu,\ell)) \\
\qquad \mathrm{mkMN(m)} \to (\sigma,\mathbf{true},(\rho,\alpha,\mu,\ell\dagger[\sigma(\mathrm{m}) \mapsto \ell(\sigma(\mathrm{m}))+\sigma(\mathrm{p})])) \\
\qquad \mathrm{bank\_i} \to (\sigma,\mathbf{true},(\rho,\alpha\dagger[\mathrm{a}_i \mapsto \alpha(\mathrm{a}_i)+\sigma(\mathrm{p})],\mu,\ell)) \\
\qquad \mathrm{bank\_c} \to\ (\sigma,\mathbf{true},(\rho,\alpha\dagger[\mathrm{a}_\ell \mapsto \alpha(\mathrm{a}_\ell)+\sigma(\mathrm{p})],\mu,\ell)) \\
\quad \mathbf{end}
\end{array}
$$

The case, as above for adding, also holds for subtraction.

$$
\begin{array}{l}
\mathrm{Int\_Clause}(\mathrm{mkS(p,t)})(\sigma)(\mathrm{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad \mathbf{case}\ \mathrm{t}\ \mathbf{of} \\
\qquad \mathrm{mkAN(a)} \to (\sigma,\mathbf{true},(\rho,\alpha\dagger[\sigma(\mathrm{a}) \mapsto \alpha(\sigma(\mathrm{a}))-\sigma(\mathrm{p})],\mu,\ell)) \\
\qquad \mathrm{mkMN(m)} \to (\sigma,\mathbf{true},(\rho,\alpha,\mu,\ell\dagger[\sigma(\mathrm{m}) \mapsto \ell(\sigma(\mathrm{m}))-\sigma(\mathrm{p})])) \\
\qquad \mathrm{bank\_i} \to (\sigma,\mathbf{true},(\rho,\alpha\dagger[\mathrm{a}_i \mapsto \alpha(\mathrm{a}_i)-\sigma(\mathrm{p})],\mu,\ell)) \\
\qquad \mathrm{bank\_c} \to\ (\sigma,\mathbf{true},(\rho,\alpha\dagger[\mathrm{a}_\ell \mapsto \alpha(\mathrm{a}_\ell)-\sigma(\mathrm{p})],\mu,\ell)) \\
\quad \mathbf{end}
\end{array}
$$

And it holds as for subtraction, but subtracting two amounts, of values designated in p and i.

```
Int_Clause(mk2S(p,i,t))(σ)(tf)(ρ,α,μ,ℓ) ≡
  let pi = σ(p)−σ(i) in
  case t of
    mkAN(a) → (σ,true,(ρ,α†[σ(a)↦α(σ(a))−pi],μ,ℓ))
    mkMN(m) → (σ,true,(ρ,α,μ,ℓ†[σ(m)↦ℓ(σ(m))−pi]))
    bank_i → (σ,true,(ρ,α†[a_i↦α(a_i)−pi],μ,ℓ))
    bank_c → (σ,true,(ρ,α†[a_ℓ↦α(a_ℓ)−pi],μ,ℓ))
  end end
```

To delete an account is to remove it from both the account register and the accounts.

```
Int_Clause(mkDA(c,a))(σ)(tf)(ρ,α,μ,ℓ) ≡
  (σ\{a},true,(ρ†[σ(c)↦α(σ(c))\{σ(a)}],α\{σ(a)},μ,))
```

Similarly, to delete a mortgage is to remove it from both the mortgage register and the mortgages.

```
Int_Clause(mkDM(c,m))(σ)(tf)(ρ,α,μ,ℓ) ≡
  (σ\{m},true,(ρ,α,μ†σ(c)[↦μ(σ(c))\{σ(m)}],ℓ\{β(m)}))
```

To compute a special function requires a place, i, to put, i.e., to store, the resulting, the yielded, value. It also requires the name, fn, of the function, and the actual argument list, aal, i.e., the list of values to be applied to the named function, fct. As an example we illustrate the "built-in" function of computing the interest on a loan, a mortgage.

```
Int_Clause(mkCP(i,fn,aal))(σ)(tf)(ρ,α,μ,ℓ) ≡
  let fct = σ(fn) in
  let val = case fn of
              "interest" →
                  let ⟨m,d⟩ = aal in fct(⟨μ(σ(m)),d?⟩) end
              ... → ...
            end in
  (σ†[σ(i)↦val],true,(ρ,α,μ,ℓ)) end end
```

This ends the last stage of the development of a script language. ∎

11.7.2 Methodological Consequences

We have already covered techniques for, and principles of describing (i.e., modelling) rules and regulations (Sects. 11.6.2 and 11.6.4). These carry over,

but in stricter forms, to the description (incl. modelling) of scripts. Designing script languages is basically like designing small programming languages. Vol. 2, Chaps. 3, 6–9 and 16–19 outlined a long series of principles, techniques and tools for designing such languages, including specifying their syntax, semantics and pragmatics.

11.7.3 Reminder + More

We remind the reader of the principle stated at the outset of this section on domain scripts.

Principle. *Describing the Domain Script Facets:* When describing a domain analyse it with respect to its script phenomena and concepts. Focus on possibly describing these separately, and make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain scripts. ▪

Techniques. *Domain Scripts:* To properly develop domain scripts, the full force of the semiotic concepts of pragmatics, semantics and syntax, and the techniques of language definition as covered extensively in Vol. 2, apply. ▪

Tools. *Domain Scripts:* Many tools exist for language design and compiler implementation. Some deal with analysis of syntactic and semantic descriptions. Others deal with the automatic generation of lexical scanners, error-correcting parser generators, and yet others with interpreter and compiler generation. We refer to standard textbooks on compiler implementation [6,14,372]. Search the Internet and you will find many references to downloadable compiler construction tools. ▪

11.8 Domain Human Behaviour

Let us consider the staff of any enterprise, any place of work, whether private or public. Some go about doing their job conscientiously: diligently carrying out tasks as expected. Other staff unconsciously sometimes forget: are sometimes a bit sloppy in the dispatch of duties. Yet other staff set themselves lower standards for the pursuit of their assignments: they are slovenly delinquent in completing their work. Finally it may be that some staff are outright criminal in doing their work: They misappropriate funds or steal from the warehouse, etc. A whole spectrum of quality thus characterises human work.

Principles. *Describing the Domain Human Behaviour Facets:* When describing a domain, analyse it with respect to its human behaviour phenomena and concepts. Focus on possibly describing these separately. Make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain human behaviours. ▪

11.8.1 Overall Principles

Characterisation. By domain *human behaviour* we shall understand any of a quality spectrum of carrying out assigned work: from *careful, diligent* and *accurate,* via *sloppy* dispatch, and *delinquent* work, to outright *criminal* pursuit. ▪

In describing a domain it is important to try capture salient features of what it means to be a human worker: being *careful, diligent* and *accurate,* being unintentionally *sloppy,* being intentionally *delinquent,* being outright *criminal* and, if describable, any shade in-between.

How one describes that, and how one, i.e., the software developer, utilises such descriptions are covered in more detail below.

Example 11.26 *Banking — or Programming — Staff Behaviour:* Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments, as illustrated in Example 11.21: We would characterise such a clerk as being *diligent,* etc., if that person carefully follows the mortgage calculation rules, and checks and double-checks that calculations "tally up", or lets others do so. We would characterise a clerk as being *sloppy* if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being *delinquent* if that person systematically forgets these checks. And we would call such a person a *criminal* if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater.

Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments along the lines illustrated in Example 11.21: We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. ▪

Example 11.27 *Shopping — Overall Consumer Behaviour:* A consumers goods market consists of consumers, retailers, wholesalers, producers and delivery services. We focus just on possible consumer behaviours: (i) a consumer inquires, with a retailer, as to availability, price, and delivery terms, of some merchandise. (ii) The retailer responds with zero, one or more offers. (iii) The consumer may decide to ignore the offers, or the consumer may select one of

the offers, or the consumer may order something that was not in the set of offers. (iv) The retailer may confirm an order, whereupon delivery takes place and an invoice is sent. (v) The consumer may decide to return the merchandise unpaid, or even paid! (vi) Or the consumer may keep the merchandise and may ignore the invoice, or may pay it, or may pay some other "fictive" (i.e., nonexisting) invoice. (vii) The consumer may then decide to return the merchandise for repair or for claims.

Formal Presentation: Shopping — Overall Consumer Behaviour

We formalise the above. The .. parts indicate "open" parts of the specification, that is, those parts which we believe can be left schematised without loss of basic understanding on the part of the reader.

```
type
   Σ
   Choice == inq | ord | acc | ret | pay | cla | ign
   CR == Inq(..)|Ord(..)|Acc(..)|Pay(..)|Cla(..)|Ign(..)
   RC == Ofr(..)|Del(..)|Inv(..)|..
channel
   cr:CR, rc:RC
value
   consumer: Σ → out cr in rc  Unit
   consumer(σ) ≡
c0   (let cho == inq⌈ord⌈acc⌈ret⌈pay⌈cla⌈ign in
c1    let σ′ =
c2      case cho of
c3        inq → let (σ″,i) = .. in cr!i ; σ″ end
c4        ord → let order = .. in cr!order end
c5        acc → if .. then let (σ″,a) .. in cr!a ; σ″ end else σ end
c6        ret → if .. then let (σ″,r) = .. in cr!r ; σ″ end else σ end
c7        pay → if .. then let (σ″,p) = .. in cr!c ; σ″ end else σ end
c8        cla → if .. then let (σ″,c) = .. in cr!c ; σ″ end else σ end
c9        ign → σ
c10     end
c11   consumer(σ′) end end)
      ⌈
s1    (let res = rc ? in
s2     let σ′ =
s3       case res of
s4         Ofr(..) → handle_ofr(res)(σ),
s5         Del(..) → handle_del(res)(σ),
s6         Inv(..) → handle_inv(inv)(σ),
s7         .. → ..
s8       end in
s9    consumer(σ′) end end)
```

We explain the above formalisation, or, to put it differently, we narrate in more detail the informal points (i–vii) above.

The consumer function has two internally nondeterministically chosen alternatives. Either the initiative is on the side of the consumer (i.e., 'client' mode, shown using "c" prefixed line labels); or the consumer "passively" awaits response from the retailer (i.e., 'server' mode, shown using "s" prefixed line labels).

(c) As a client the consumer nondeterministically internally, i.e., of her own free will,[16] chooses (c0) between doing any of the actions (c3) inquire about merchandise (..), (c4) order merchandise (..), (c5) accept delivery of merchandise (..) believed to have been delivered (hence the **if .. then .. else .. end**), (c6) return merchandise (..) believed to have been delivered (hence the **if .. then .. else .. end**), (c7) pay for merchandise (..) believed to have been delivered (hence the **if .. then .. else .. end**), (c8) claim refund on supposedly faulty merchandise (..) believed to have been delivered (hence the **if, then, else**), or (c9) ignore whatever goes on! Any of these actions (the last is, in effect, a nonaction) does, indeed, leave a side effect, a remembrance, in the mind of the consumer, hence a state change, from state to state′ ((c1)).

(s) As a server the consumer awaits a response from the retailer. If none is forthcoming, the consumer "deadlocks"! This models that the consumer has gotten "stuck" and stubbornly refuses to take her own initiative, just waits and waits. If a response is forthcoming, it is either (s4) an offer, possibly prompted by an earlier consumer inquiry — but not necessarily. It could be an "own initiative" by the retailer, or (s5) a delivery (etc.), (s6) an invoice (etc.), (s7) or other! In any case, a new state (s2) results. The consumer resumes being a consumer in a new state resulting from either her own initiatives, or from externally prompted actions (c11), resp. (s9). ▪

In the above example we are deliberately leaving many things unspecified (..). The point is that we are not so much interested — in this section — in those (..) things. We are interested in modelling, in describing, the vagaries of consumers. These uncertainties, these unpredictable wanderings, were fully described by the nondeterministic choice (c0) and by the fact that after the outputs (!) the consumer "recursed" being a consumer without awaiting responses from the retailer. It was also shown in our not defining, yet, the handle_xyz(..) clauses.

Example 11.28 *Shopping — Detailed Consumer Behaviour:* We continue Example 11.27. We left some open points in the earlier example. We shall use these to illustrate other aspects of human behaviour, its informal and formal descriptions.

[16] We tacitly assume that such a concept as "free will" exists in connection with consumer behaviour!

We start by singling out the treatment of a consumer-initiated initiative, like making an inquiry (c3).

Formal Presentation: Shopping — Detailed Consumer Behaviour

c3 inq → **let** (σ'',i) = .. **in** cr!i ; σ'' **end**

To (c3) we add the "missing" information about how we form (i.e., "compute") the information (i.e., data) that goes into an inquiry: '..':

Formal Presentation: Shopping — Detailed Consumer Behaviour

c3 inq → **let** (σ'',i) = mki(σ) **in** cr!i ; σ'' **end**

and

value
mki: Σ → Inq Σ

In the formula above we have referred to the action of human "gathering" the information that goes into an inquiry by the cryptic function name mki. To make an inquiry we assume that the consumer refers to whatever sense impressions that person may have, and we model that ("whatever sense impressions that person may have") as part of that person's state. Hence the gathering action operates on the state and updates it with the fact that the person (whose state it is) has contemplated and formed an inquiry. We leave the description of mki open. Leaving it open also leaves it open to interpretation. Anything is allowed that forms an inquiry and possibly changes the state. This "openness" models the vagaries of human behaviour. The case for all other consumer-initiated actions directed at the retailer is similar to that of the inquiry action in respect of acting upon and communicating information. We now treat the case of retailer-initiated interactions. Let us consider the consumer's reaction to a retailer offer response.

Formal Presentation: Shopping — Detailed Consumer Behaviour

s4 Ofr(..) → handle_ofr(res)(σ)

We refer to this reaction by handle_ofr. As for the making of an inquiry (etc.), this action is not being further described, other than saying: It is any action that somehow records, in the consumer's state, i.e., mind, or jotted down on a

piece of paper, say stuck to a kitchen notice board, the fact that approximately "such and such" an offer was received.

Formal Presentation: Shopping — Detailed Consumer Behaviour

```
value
  handle_ofr: Ofr → Σ → Σ
```

No further action is described. In particular, the perhaps expected reaction of the consumer "immediately firing off" an order, or a declination of the offer is not described. Any such possible reaction is modelled by the internal nondeterministic choices of the client actions of the consumer: The consumer may, sooner or later or even never select or choose an order reply. And that order reply may relate, "through" the mko action (c4, not shown), to the Offer response (s4). ∎

11.8.2 Methodological Consequences

Techniques. *Human Behaviour:* (I) We often model the "arbitrariness" of human behaviour by internal nondeterminism. There are two concepts to keep clear of one another: the user choosing to perform an arbitrary action, act_i, from a set Act, of alternative actions, and the interpretation, by the user, or by a system, of that action, b_x, that is, the resulting behaviour.

Formal Explication: Conceptual Model of Human Behaviour, I

```
type
  Act == act_1 | act_2 | ... | act_n | ...
value
  f(...) ≡ ... b_p ⌈⌉ b_s ⌈⌉ b_d ⌈⌉ b_c ...
```

Act denotes a type of action. f defines a function which nondeterministically, under no influence from an, or the, environment (i.e., arbitrarily), selects one of the behaviours b_p, b_s, b_d or b_c. The, possibly deterministic, meaning of each of the alternatives can then be separately described. Proper actions, act_i: some actually perceivable fruitful action, as illustrated in the examples above through the use of the signature-only functions (mk_x and handle_y); and (or versus) action qualities: (i) b_p: professional, (ii) b_s: sloppy, (iii) b_d: delinquent, or (iv) b_c: criminal. We prefer to merge the latter into the former, that is, to assume that the definitions of the actions (mk_x and handle_y) embody both intended actions as well as their quality. ∎

Techniques. *Human Behaviour:* (II) Alternatively we can model human behaviour by the arbitrary selection of elements from sets and of subsets of sets:

Conceptual Model of Human Behaviour, II

```
type
   X
value
   hb_i: X-set ...→... , hb_i(xs,...) ≡ let x:X • x ∈ xs in ... end
   hb_j: X-set ...→... , hb_j(xs,...) ≡ let xs′:X-set • xs′ ⊆ xs in ... end
```

The above shows just fragments of formal descriptions of those parts which reflect human behaviour. Similar, loose descriptions are used when describing faulty supporting technologies, or the "uncertainties" of the intrinsic world. ∎

Techniques. *Human Behaviour (III):* 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:

Formal Explication: Conceptual Model of Human Behaviour, III

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

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. ∎

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. This is the subject of Part V.

11.8.3 Human Behaviour and Knowledge Engineering

We refer to Sect. 4.1.1 for a first, albeit very brief coverage of the concept of knowledge engineering.

Domain engineering aims at making precise our understanding of the entities, functions, events and behaviours of the observable phenomena and the intellectual concepts of the domain. By *knowledge* we shall, in the narrow context of knowledge engineering, understand that which a human (or a machine, i.e., an agent) knows or believes or assumes or commits with respect to (*knowledge*, *beliefs*, *promises* or *commitments* of) another agent. By *knowledge engineering* we shall understand the formulation (whether informal or formal) of such knowledge. Knowledge engineering is thus concerned with understanding relations between two or more agents' knowledge (etcetera) about one another with respect to the following issues: what does an agent know about what another agent knows or believes; which (things) does an agent promise another agent who may then commit or promise other or similar things to yet other agents; and so on. The subject of knowledge engineering is of importance when we model human behaviour but we shall not in this book venture into this very important field of computer and computing science. We refer to the seminal treatise on the subject [98].

11.8.4 Discussion

Please observe the difference between the version of meaning under the rules and regulations facet, Sect. 11.6.2, and the present version. The former reflected the semantics as intended by the stakeholder who issued the rules and regulations. The latter reflects the professional or the sloppy or the delinquent or the criminal semantics as intended by the similarly "qualified" staff which carries out the rule-abiding or rule-violating actions. Please also observe that we do not here exemplify any regulations.

11.8.5 Reminder

We remind the reader of the principle stated at the outset of this section on domain human behaviour:

Principles. *Describing the Domain Human Behaviour Facets:* When describing a domain, analyse it with respect to its human behaviour phenomena and concepts. Focus on possibly describing these separately. Make sure that descriptions of other described domain facets are commensurate with possibly multiple, alternative descriptions of domain human behaviours. ∎

11.9 Other Domain Facets?

We have exemplified and formalised some aspects of human behaviour in the domain. And we have informally and formally described how we model some aspects of some facets (rules and regulations, respectively human behaviour). The latter form some initial contributions to a more proper theory of what we mean by domain facets. The domain facets that we have covered included: intrinsics, support technologies, management and organisation, rules and regulations, domain scripts and human behaviour. The question now is obvious: Are there other domain facets? We refrain, at present, from an answer. But we would be surprised if there were not! In other words, we expect further practice and further exploratory and experimental research to yield additional facets. Thus the reader should be on the look out for whether the facets covered here suffice. More generally we must accept the next principle:

Principle. *Domain Facets:* When modelling, informally or formally, a domain, analyse the domain phenomena with respect to whether one or another, or a combination of currently identified domain facets suffice to model the domain, or whether you, the developer, have to discover, i.e., identify, define and otherwise find a suitable set of one or more principles, techniques and tools with which to model the domain. ▪

11.10 Composition of Domain Models

From the various facet descriptions the domain engineer now has to weave a fabric, and Sect. 11.10.1 is about that. The domain engineer may also have to formalise the full description, and Sect. 11.10.2 is about that.

11.10.1 Collating Domain Facet Descriptions

General

The various domain facets can be described more or less individually. It is a good idea to try identify and describe these separate facets individually — in other words applying the principle of separate concerns. But, in doing so the describer may be repeating some descriptive material unnecessarily. Such duplicate material may differ in details and may thus create inconsistencies as well as doubts in the minds of the readers. But analysing the domain and describing it on a per facet basis may yield insight and lead to discoveries about the domain not otherwise attainable.

A Comprehensive Narrative

Describing the domain on a per facet basis may lead to a fragmented, staccato (abrupt, disjointed) description. To avoid this it may be a good idea to take all the bits and pieces of the various facet descriptions and write them into one whole comprehensive narrative. In merging the various facets into one structured narrative the domain engineer may discover possible inconsistencies — and thus will have an early opportunity to correct such. The possibly revised (for example corrected) "bits and pieces" should not be thrown away. They can serve as possibly clarifying study material.

From Big Lies via Smaller Lies to the Truth

A Golden Rule of Comprehension

Develop your domain understanding — and hence the first round of domain descriptions — by analysing and describing the domain facet-by-facet (including formalisation), then by consolidating this into a more pedagogical and didactical[17] flow of narration (with edited formalisation).

One typical way of structuring a comprehensive narrative, as well as its accompanying formalisations, is to formulate the full narrative as a sequence of narratives. Initially the narratives pretend to cover the entire domain, starting, obviously with some intrinsics. But steps of subsequent narratives enlarge upon the scope, choosing pedagogically further domain aspects — be they of intrinsic, of support technology, of management and operation, or of the nature of some other domain facets. The order chosen is determined by what the writer judges is good didactics and good pedagogics. Many such orders are possible. We can phrase this unfolding of a narrative as follows:

Principles. The principle of *From Big Lies via Smaller Lies to the Truth*. To achieve a smooth, pedagogically and didactically sound presentation of some universe of discourse, start by narrating a suitable lie, call it a big lie, a gross simplification. Proceed by adorning the ("false") narration with smaller lies, that is, with less gross simplifications. In doing this you have to accommodate it so that the smaller lies fit nicely onto the big lie, that is, that you do not have to change anything in your presentation, only, so to speak, "refine" it. Then go on to detail the less gross simplifications, i.e., tell tiny lies while still adhering to the "accommodation principle". Finally you have added so much detail that you have told "the truth", that is, what we abstract of the universe of discourse as our truthful abstraction of that universe. Thus "the limit of all the lies is the truth". ∎

[17] *Pedagogical:* of the art and science of teaching. *Didactic:* intended to convey instruction and information as well as pleasure and entertainment [238].

11.10.2 Technical Issues

We saw, in Sect. 11.3.1, the need for composing intrinsic descriptions from intrinsic description parts. We have now seen, in this chapter, through its coverage of many facets, the need for composing from descriptions of separate facets of a domain a comprehensive and consistent description. As in Sect. 11.3.1, we refer to the use, for example, of RSL's *scheme* facility. We refer to Vol. 2, Chap. 10 (Modularisation) in which we cover the *scheme* concept of RSL (Sect. 10.2 (RSL Classes, Objects and Schemes) of that volume). Non-intrinsic facet schemes can be expressed by *extending* basic (e.g., intrinsic) schemes *with* additional types, values and axioms. The *hiding* facility of schemes can likewise be used to express different, but commensurate models.

11.11 Exercises

11.11.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

11.11.2 The Exercises

Exercise 11.1 *Intrinsics.* For the fixed topic, selected by you, identify and describe

1. some intrinsic entities,
2. some intrinsic functions,
3. some intrinsic events and
4. some intrinsic behaviours.

Exercise 11.2 *Business Processes.* For the fixed topic, selected by you, identify and describe two ("as different as is reasonable") business processes.

Exercise 11.3 *Support Technologies.* For the fixed topic, selected by you, identify and describe two ("as different as is reasonable") support technologies.

Exercise 11.4 *Management and Organisation.*

1. For the fixed topic, selected by you, identify and describe management entities, functions, events and behaviours.
2. Identify and describe a possible organisational structure of your chosen domain.

Exercise 11.5 *Rules and Regulations.* For the fixed topic, selected by you, identify and describe three to four rules and corresponding regulations.

Exercise 11.6 *Scripts.* For the fixed topic, selected by you, identify and describe a possible script language (hint at a syntax, and rough sketch or narrate a semantics).

Exercise 11.7 *Human Behaviour.* For the fixed topic, selected by you, identify and describe:

1. specifically desirable human behaviours, and
2. specifically undesirable human behaviours.

Exercise 11.8 *A Comprehensive Domain Description.* For the fixed topic, selected by you, collate the descriptions that you have produced in answers to Exercises 11.1–11.7 into one comprehensive domain description.

12

Domain Acquisition

- The **prerequisite** for studying this chapter is that you now know what should go into a domain model: Descriptions of its intrinsics, its business processes, its supporting technologies, its management and organisation, its rules and regulations, their scripts, and the variety of human behaviours, and, possibly, many more facets.
- The **aims** are that you will now know how to gather facts about the domain, and that you will then know how to organise those facts for subsequent analysis.
- The **objective** is that you will competently lead and carry out thorough acquisition of domain facts from domain stakeholders.
- The **treatment** is pragmatic and systematic.

12.1 Introduction

Characterisation. *Domain Acquisition:* By *domain acquisition* we shall understand the process of obtaining facts about the domain, that is, of *eliciting* (i.e., capturing) facts from domain stakeholders, of *writing* down descriptions about them, and of roughly *structuring* (i.e., roughly organising, roughly *classifying*) these descriptions. ▪

The term *elicitation* is thus not synonymous with *acquisition*. *Elicitation* stands for part of the acquisition process. The *structuring* (organising or *classifying*) is meant as not being based on any serious, nontrivial analysis — that comes later! It is just an *indexing* for purposes of making easy some process of *search* for *domain description units* having specified *properties*.

Acquisition involves *stakeholders*, and usually many of them. And acquisition involves writing something down, namely: *domain prescription units*.

Elicitation is difficult. So we shall further investigate that concept. But first, let us examine what it is we have to elicit, to extract, to capture, namely *domain facts*.

12.1.1 Domain Facts

Characterisation. By *domain facts* we shall understand what is conceivable of as (manifest) phenomena or (intellectualised) concepts of the domain, whether as properties of entities, or of functions, or of behaviours (processes with events). ▪

12.1.2 Elicitation of Domain Facts

Characterisation. By *elicitation of domain facts* we mean a process of *studying* the domain, that is, of *reading about* the domain. Of *talking with, interviewing,* domain stakeholders. Of yourself, or stakeholders, *recording* (possibly aided by *questionnaires*) your, or their conception — i.e., perspective or views — of the domain, that is, the properties of the entities, the functions, the behaviours, including the events of the domain. These entities, functions, events and behaviours are either *phenomenological,* i.e., are actually manifest, or are *conceptual,* i.e., are "intellectualised", in the domain. ▪

12.1.3 Recording Domain Facts

Characterisation. By *recording domain facts* we shall understand a not necessarily systematic, nor coherent or complete, writing down of "bits and pieces" of rough-sketch descriptions of one or another fragment of a domain, such that each of these bits and pieces forms a *unit of domain description* which is then suitably *indexed* (i.e., *classified,* categorised, named). ▪

Characterisation. Roughly, an *index* of a *unit of description* is a marking as to one or more categories of the name of the phenomena or concept being recorded, and/or the names of the stakeholders, and/or a category name for the stakeholder group, and/or whether the unit describes a phenomenon or a concept, and/or the type of the phenomenon or concept (entity, function, behaviour), and/or the type of the attributes and/or the type of the domain facets. ▪

Characterisation. By *unit of domain description* we shall understand some informal or formal, usually rough-sketch text which starts describing something, or which adds to a description of something, with the unit description being annotated by some initial classification, such that the unit description forms a whole (e.g., a complete sentence). ▪

A unit of description is, deliberately, a loose notion. It reflects the nature of the elicitation process. This process is exploratory, even experimental: The persons involved in the elicitation process, normally, at the outset of elicitation, do not know where the elicitation process will take them, that is, what will evolve! Eventually, a "large body" of units of domain descriptions, that is, of recordings of domain facts, will evolve.

Example 12.1 *Units of Domain Descriptions:* We give three examples of units:

(i) A railway net consists of lines and stations. *Initial (rough) classification:* stakeholder: *Miss AA, passenger;* phenomena/concept type: *phenomena: entity;* attribute: *static;* facet: *intrinsic;* phenomena names: *railway net, line, station.*

(ii) One may perform the following operations involving simple demand/deposit bank accounts: opening, deposit, withdrawal, transfer, obtaining a statement (of past operations) and closing. *Initial (rough) classification:* stakeholder: *Mrs. BB, bank teller;* primary phenomena/concept types: *phenomena: entity;* secondary phenomena/concept types: *phenomena: functions;* attributes: *inert* (for primary and secondary phenomena); facet: *intrinsic* (for primary and secondary phenomena); primary phenomena name: *demand/deposit bank account;* secondary phenomena names: *demand/deposit, open, deposit, withdrawal, transfer, statement, close.*

(iii) An aircraft journey consists of a sequence of two or more airport visits. *Initial (rough) classification:* stakeholder: *Mr. CC, pilot (captain);* phenomenon/concept type: *concept: behaviour;* attribute: *dynamic;* facet: *intrinsic;* phenomena names: *aircraft journey, airport visit, ...* ∎

We observe, from the above examples, that often the unit text is (far) shorter than the classification. And that the classifications need to be refined into primary and secondary phenomena and concept types, attributes and facets.

We refrain from detailing possible forms of units and their classifications. Such a detailing would be appropriate for a specific instantiation of an acquisition process, typically one for which specific tools are then provided. Here it suffices to indicate the issues involved.

12.1.4 Indexing Domain Description Sketches

Characterisation. By an *index* we shall understand a "naming" of phenomena and/or concept names; a stakeholder group and one or more persons of that group; whether the primary (i.e., the defining) and secondary (i.e., the using) ideas are phenomena or concepts; the kind of primary and secondary phenomena or concepts: whether entities, functions, events or behaviours; relevant domain attributes; relevant domain facets: intrinsics, support technology, etc. ∎

Indexing is basically an informal endeavour. Indexing is carried out in order to facilitate the search for some properties: as to the name of entity, function,

event or behaviour; or as to stakeholder group or person; or as to whether a phenomenon or a concept; or as to entity, function, event or behaviour; or as to attribute kind; or as to facet category. That is, we envisage that the domain engineer, in the analysis process, may wish to review several domain description units with respect to some (logical) criteria that involve one or more of the above-listed classification categories.

Characterisation. By *indexing domain sketches* we shall understand a process of equipping, i.e., of annotating, domain description units, with one or more classification indexes, where these distinct indexes cover phenomena or concept names, stakeholder kinds and person names, phenomena or concept kinds and types, domain attributes and domain facets. ∎

12.2 The Acquisition Process

The acquisition process now, typically, proceeds, in one or (usually) more rounds (i.e., cycles) of each of the "steps" as now listed:

(i) *Review of stakeholder index, etc.:* (i.1) Is such a listing of "all relevant" stakeholders established? (i.2) If so, is it adequate? (i.3) If not, then establish it, and review. (i.4) Establishment of contact to, i.e., liaison with, identified stakeholder groups and persons.

(ii) *Study of domain documents:* (ii.1) Gathering such documents from various sources: stakeholders, libraries (books, journals), research centres, the Internet, etc.; (ii.2) evaluation of relevance of such documents; (ii.3) reading of relevant documents; and (ii.4) preparation for the recording of domain description units (see below, item (v)).

(iii) *Casual talks with stakeholders:* (iii.1) Initial, "loosening up" talks, i.e., chats, in person, with stakeholders, (iii.2) for the purposes of establishing "rapport", i.e., confidence, trust, (iii.3) including preparation for the recording of domain description units (see item (v) below).

(iv) *Systematic, questionnaire-based interviews of stakeholders:* (iv.1) Formulation and "printing" of questionnaires (see below); (iv.2) distribution, or personal handing out, in connection with casual talks, of questionnaires; (iv.3) gathering of more or less completed questionnaires; (iv.4) and preparation for the recording of domain description units (see below, item (v)).

(v) *Recording of domain description units:* (v.1) There is the recording, on paper, or electronically, done by either the interviewed and questioned stakeholders, or by the (domain engineer) interviewers; (v.2) and there is the result, on paper, or on some storage medium, of this recording: a document consisting of one or more domain description units.

(vi) *Classification of domain description units:* (vi.1) Each domain description unit is then briefly examined, individually, (vi.2) i.e., separately from the examination of other domain description units, (vi.3) and to each is affixed as

many of the domain description unit attributes as are relevant and as can be ascertained.

(vii) *Review of domain acquisition process:* Once, what is believed to be, at some point in time, "all" domain description units — once all these have been received and indexed, (vii.1) they are examined as to whether those that have been elicited form a necessary and sufficient collection, (vii.2) whether they truly represent what has been said, i.e., are "vetted" for accuracy, (vii.3) or whether some need to be rejected, and/or more may be needed.

This review process takes into account such issues as: Is the collection of description units gathered representative (have all selected stakeholder groups replied with similar or uneven care)? Are the domain description unit texts understandable? The review is not analysing the collection for consistency, conflicts or completeness. That follows the domain acquisition process. We will now treat several of the above "steps" in some detail.

12.2.1 Stakeholder Liaison

We remind the reader of Chap. 9: "Domain Stakeholders".

We can thus assume that a list of all possibly relevant domain stakeholders has been established and reviewed by all parties to a contract of development of a domain description. Now comes the effectuation, the liaison and interaction with what is considered appropriate, identified members of those domain stakeholder groups who are deemed relevant for a successful completion of that development. The contract must secure timely availability of both the domain development engineers and the identified domain stakeholders. Let us refer to the latter as the domain stakeholder representatives. A pair of managers from, respectively, the domain development engineers and the stakeholders must ensure an ongoing, free and timely use of the time resources of the domain development engineers and the identified domain stakeholders. It is that act of ensuring we could call the domain stakeholder liaison.

12.2.2 Elicitation Studies

Well-nigh every application domain has its literature, one, which in one form or another more or less explicitly hints at descriptions of one or another segment of the domain. Please observe the hedges incorporated in this last sentence. By this we mean that it is not always that obvious to find good, written descriptions of every domain.

To give the reader an idea of possible domain description sources, besides the normal technical/scientific journal and book publications, we list a few examples of organisational sources:

- **Air Traffic:** ICAO, the International Civil Aviation Organization[1] publishes journals and reports, organises Air Navigation and other conferences,

[1] www.icao.int/

manuals, etc., on such subjects as airport, air traffic control, air navigation, etc.

- **Railway Systems:** UIC, the International Union of Railways[2] likewise publishes journals, reports, thematic periodicals, thematic bibliographies, follows European railway system legislation, etc.
- **Ports and Harbours:** The International Association of Ports and Harbors' home page[3] lists publications of journals and proceedings from port and harbour conferences, etc. The International Harbour Masters' Association home page[4] provides some input to studies on ports and harbours. The Port Technology International home page[5] is likewise a promising source of information, and its 'Partners Page' lists partners with interests in the well-being of ports and harbours. The home pages of the Hong Kong Marine Department[6] as well as the Marine and Port Authority of Singapore[7] provide further, excellent information on the domain of ports and harbours.
- **Logistics:** The Australian government's department of Transport and Regional Services home page[8] provides interesting references to starting points for studies of the logistics domain. So does the Logistics Management Institute home page.[9] The private, it seems, one-man firm Freightworld home page[10] also seems to be a good reference source. The footnoted home page reference[11] lists an extensive number of transport organisations (etc.) involved, in one form or another, with logistics. Members of these lists can be consulted also in connection with matters other than only logistics, for example, air transport, railways, shipping, etc.
- **Hospitals:** The International Hospital Federation's home page[12] is a good starting point for studying the domain of hospitals. The Virtual Hospital home page[13] is certainly a relevant source. The European standards organisation CENELEC's page[14] gives references to European Standardization of Health Informatics. Finally, it seems that the Health Information Systems Special Interests Group page at the University of Aveiro, Portugal[15] is worth consulting.

[2] www.uic.asso.fr/home/home_en.html
[3] www.iaphworldports.org/top.htm
[4] www.harbourmaster.org
[5] www.porttechnology.org
[6] www.mardep.gov.hk/
[7] www.mpa.gov.sg
[8] www.dotars.gov.au
[9] www.lmi.org
[10] www.freightworld.com
[11] www.transettlements.com/current resource pages/organizations.htm
[12] www.hospitalmanagement.net
[13] www.vh.org — partially outphased
[14] www.centc251.org
[15] www.ieeta.pt/sias

In other words: Internet-based search engines are primary instruments for information acquisition. However your patience may be taxed in searching for appropriate information.

12.2.3 Elicitation Interviews

The purpose of elicitation talks was to establish rapport and build confidence between each domain stakeholder group and the domain engineers assigned to capture domain facts from respective domain stakeholder group members. Instead we shall, in this section, more closely examine the processes of interviews and their follow-up.

To repeat, the domain engineer embarks on interviews by first preparing a customised questionnaire, one that is "fitted" to the specific domain at hand. The domain engineer meets one or more representatives of the designated stakeholder group in order to introduce them to the questionnaire. This introduction ("familiarisation") is done by "walking" through the questionnaire with these representatives, explaining terms to them, and by answering questions that might arise during this interview. Then these representatives are left to fill in their own copy of an identical questionnaire, either individually, or as a group. Members of the group are encouraged, before filling out the questionnaire, to contact their domain engineer for resolution of any "metaquestions", that is, questions about how to interpret the questions in the questionnaire. Finally, the questionnaire is believed completed and is returned to the group's domain engineer.

12.2.4 Elicitation Questionnaires

But how is such a questionnaire formulated? And, is there only one round of questionnaire interviews? We can right away suggest that there should be as many rounds of questionnaire interviews as the domain engineer thinks necessary to cover the various domain stakeholder groups' perspectives in necessary and sufficient depth. That is, we cannot usually plan for how many rounds are needed. For an unfamiliar domain, one that has not been described before, domain acquisition is a research undertaking — and for such research we cannot beforehand determine the number of "rounds" needed. But we can give some general guidelines for how to formulate a questionnaire, and, after that, we will give an example of such a questionnaire.

General Guidelines: Questionnaire Structure and Contents

We know, now, what a domain description should look like, what it should contain. Namely, at the least, it should contain an enumeration of domain stakeholders: their names and categories. It should contain descriptions within the chosen span and scope of the domain with respect to the following aspects: a set of description units that covers each applicable *facet* (intrinsics,

support technologies, management and organisation, rules and regulations, human behaviour, and scripts), i.e., a set of description units which emphasise the various *domain attributes* (temporal and spatial attributes, attributes of continuity, discreteness and chaos, static and dynamic attributes, tangibility attributes, dimensionality attributes, etc.). Finally, it should contain a set of description units which, for each facet, attribute, and "thing", provide the type and some details of whether the "thing" is a *phenomenon* or a *concept*, and specifically of what *kind* the phenomenon or concept is: entity, function, event or behaviour. So our questions should be such as to tempt the stakeholders to convey these *facets*, *attributes* and *phenomena* or *concept* kinds to the domain engineer.

Special Guidelines: Questionnaire Structure and Contents

When outlining the questions concerning facets, attributes, and phenomena or concept kinds, the domain questionnaire must be tailored — "tuned" — with examples that are pertinent to the domain. In our next example we shall indicate that "tuning" with *italic font*.

Example 12.2 *An Example Questionnaire:* We show, over the next pages, a long example of a questionnaire directed at healthcare workers (nurses, porters, medical doctors, and the like) in a hospital. (0) Dear stakeholder, you are please asked to 'collect your thoughts', i.e., to concentrate on thinking — thoroughly, conceptually — about your work. (0.1) Not your work as you would like it to be, or to become, (0.2) but as you honestly believe it is, as it is.

Please do not idealise it, or paint your own role pessimistically or optimistically, or in any other way express opinions about it. Please discuss it just as it is, objectively and loyally.

(1) First we ask you (see questions (1.1 and 1.2) a little further down) to tell us about what we call the *entities* of your work, the things you can point to or the things that you can otherwise conceive.

> Typical entities of your work are such as (and now we enumerate a long list of such things) *people: patients, nurses, medical doctors, technicians, next-of-kin (of patients), etc.; materials: medicine, bandages, distilled water, etc.; tools: syringes, blood pressure meter, thermometer, etc.; documents: patient medical records, etc.; other facilities: wards (with beds, etc.), operating rooms (with their equipment), etc. And so forth.*

(1.1) Please list, by naming them, the entities that you meet in your daily work, whether frequently, sometimes or seldomly. (1.2) For each entity listed please describe its characteristics, its properties and its use — not whether it is a good entity or a bad one.

(2) Then we ask you (see questions (2.1 and 2.2) a little further down) to tell us about what we call the *functions* of your work, the things you do with entities.

Typical functions of your work are such as (and now we enumerate a long list of such things) *inject, with a syringe, penicillin into a patient; create, edit or copy a patient medical record; transfer a patient from a ward bed to a movable bed, etc.*

(2.1) Please list, by naming them, the functions that you perform in your daily work, whether frequently, sometimes or seldomly. (2.2) For each function listed please describe its characteristics: which things enter into doing the function, and which things result from having performed the function.

(3) Finally, we ask you (see questions (3.1 and 3.2) a little further down) to tell as about what we call the *behaviours* of your work, the procedural sequences of functions involving entities as well as *events* that trigger your work in different directions, etc.

A typical behaviour of your work is, for example, *the process of interviews, analyses, diagnostics, treatment, etc., such as a patient undergoes during hospitalisation.* In more detail: *A patient hospitalisation behaviour starts with admission to the hospital — during which interviews, analyses, etc., takes place. Then it proceeds with referral to a ward, installation in the ward, etc. After, typically, a night, the patient for a surgery treatment is prepared for operation, conveyed to the operating room, made ready, the operation is performed, and the patient is put under observation in a special "wake-up" (i.e., intensive care) ward.* And so on.

(3.1) Please list, by naming them, the behaviours that you observe, or participate in, in your daily work, whether frequently, sometimes or seldomly. (3.2) For each behaviour listed please describe its characteristics.

(4) Now we go back to item (1) and ask you to provide further details. What we want you to do is to tell us more about the entities you have listed and whose facets you have described. If during doing this you are reminded of entities that you encounter in your daily work, but which you forgot to first list, then please add them to your original list. Now for each entity please consider the following, what we call *facets:* intrinsics, business process, support technologies, management and organisation, rules and regulations, human behaviour, etc.

Here the domain engineer explains, by using respective examples already understood from the domain, the concept of these facets.

(4.1) For each entity of your list please go into detail and describe whether you consider it an intrinsic facet, a business process facet, a support technology facet, a management and organisation facet, a rules and regulations facet, a script facet, or a human behaviour facet.

(5) Now we go back to item (4) and ask you to provide further details. What we want you to do is to tell us even more about the entities you have listed and whose facets you have described. If during doing this you are reminded of entities that you encounter in your daily work, but which you forgot to first list, then please add them to you original list. Now for each entity please consider

the following, what we call *attributes:* temporal and spatial attributes, continuous, discrete and chaotic attributes, static and dynamic attributes, tangibility attributes, dimensionality attributes, etc.

Here the domain engineer explains, by using respective examples already understood from the domain, the concepts of these attributes.

(5.1) For each entity of your list please go into detail and describe which attributes you think it enjoys: temporality and spatiality, continuity, discreteness and chaoticness, whether it is static or dynamic, and then which kind of dynamics it enjoys, its tangibility, and its dimensionality.

(6) Now we go back to item (3), behaviours, and ask you to provide further details. These details are concerned with such things as whether the ordering of some functions being performed can be changed, which events might occur and the reaction expected, and what flow of people, material, information and control such events may designate.

(6.1) For each behaviour that you have described, please review whether actions of behaviours may occur in any order, or do occur in specific order; and please name and describe what you consider important events: the people, materials, information and control (stimuli) which partake in the events. ∎

From domain to domain, and from case to case (i.e., contract to contract) the domain engineers may extend or shorten the above exemplified questionnaire.

12.2.5 Elicitation Reports

The domain engineer now assembles an elicitation report. Roughly it consists of all the reports received from solicited stakeholders, together with the domain engineer's "quick" assessment of these reports: their trustworthiness, as well as their indexing. We expect an elicitation report to be both electronically available, and available in paper form.

12.3 Discussion

12.3.1 Concept and Process Review

We have outlined important concepts of and steps in the domain acquisition process: the concepts of domain description units and their indices; the identification of and liaison with domain stakeholder groups and representatives; and the concept and formulation of domain acquisition questionnaires — as based on what we know of how a domain model is structured and what it must contain. We also include in this list the process of domain engineer studies of the domain; the process of domain stakeholder interviews and domain stakeholder completion of questionnaires — cum writing domain description units; the process of indexing; and the process of evaluating whether a domain acquisition should be ended or continued.

12.3.2 Process Iteration

Domain acquisition, especially for "untrodden" domains, i.e., for new, unfamiliar domains, is an art. It hinges on being research-oriented. As already indicated it will certainly evolve in iterations, and hopefully these can be quickly ascertained as to whether they are converging rapidly enough, or not — or even diverging.

12.3.3 Delineation: Acquisition and Analysis

We have decided here to present the domain acquisition process as separate from the domain analysis process. As two processes they interact: The latter may result in a call for resumption of the former. One might therefore, as the term "resumption" indicates, treat them as coroutines.

To us, the essential difference is: The acquisition process is very stakeholder-intensive, and is centred around rough-sketching description units. On the other hand, the analysis process is not stakeholder-intensive, and is centred around analysing description documents, and in concept formation.

12.3.4 Principles, Techniques and Tools

We summarise:

Principles. The principle of *domain acquisition* is that of providing material which helps one to uncover inconsistencies and conflicts in human perceptions of the domain, and to discover such concepts as can be claimed to underlie the domain. ∎

This is truly a daunting task. It is, by its very nature, a task requiring training, more in science than in engineering. The principle, as formulated, also underlies the work of a natural science researcher when exploring nature.

Commensurate with that, we formulate the next point.

Techniques. The techniques of *domain acquisition*, besides all those clerical ones mentioned earlier in this chapter — (i) review of and liaising with stakeholders; (ii) identifying, collecting and reading up on domain literature; (iii) casual, explorative talks with stakeholders; (iv) formulation of questionnaires and questionnaire-based interviews with stakeholders; (v) recording the results of these interviews; and (vi) collecting and indexing domain description units — are also the (vii) review of these indexed domain description units, i.e., the "vetting" of their contents, so to speak. That is, ensuring (vii.1) that the set of indexed domain description units is necessary and sufficient, and (vii.2) that the set is accurate, and properly reflects bona fide stakeholder perspectives. ∎

Tools. *Domain Acquisition:* Since the domain acquisition process is necessarily an informal one, the tools are: (a) human reading and interviews, (b) processing the recording of domain description units (c) in a way that facilitates storage and retrieval, (d) and which allows for specialised programs (queries) to be (later) expressed and performed with the aim of facilitating domain analysis. ▪

12.4 Exercises

12.4.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

12.4.2 The Exercises

Exercise 12.1 *Domain Description Units.* For the fixed topic, selected by you, you are to suggest some 12 domain description units — such as might have been elicited from 4 different stakeholder groups.

Exercise 12.2 *Domain Description Unit Indexes.* Based on your answer to Exercise 12.1, you are to suggest (the possibility of) some indexing scheme.

Exercise 12.3 *Domain Questionnaire.* For the fixed topic, selected by you,

1. Suggest a short, precise narrative that can form the basis for a domain questionnaire.
2. And then formulate such a domain questionnaire.

13

Domain Analysis and Concept Formation

- The **prerequisite** for studying this chapter is that you have studied the chapters on stakeholders, domain attributes, domain facets and domain acquisition.
- The **aims** are to introduce ideas of concept formation, and to introduce concepts of domain consistency, completeness and conflict.
- The **objective** is to bring you further along the road to becoming a professional software engineer who is versatile in domain engineering.
- The **treatment** is informal, yet systematic.

13.1 Introduction

Characterisation. By *domain analysis* we mean a reading of rough domain acquisition description units, with the aim of forming concepts from these, as well as with the aim of discovering inconsistencies, conflicts and incompletenesses within these domain description units. ∎

The title of the chapter says Domain Analysis and Concept Formation. The above characterisation of the domain analysis mentions both concept formation and inconsistencies, conflicts and incompletenesses. But the latter concepts were not mentioned in the title of the chapter. So what do we mean? We mean that we must get rid of negatives: inconsistencies, conflicts and incompletenesses, while achieving the positive: concept formation. Hence we emphasise the positive in the title. But treat both, in the order, first concept formation, then inconsistencies, conflicts and incompletenesses.

13.2 Concept Formation

We see a spectrum of concepts emerging from proper concept formation. We see this from the simply abstracted concepts, treated in the first section below,

to the cleverly abstracted, "discovery" or breakthrough concepts mentioned further on.

13.2.1 Simply Abstracted Concepts

We exemplify the formation of some concept as based on given description units. So first we give some description units. On the basis of these we can then, in a following example, illustrate the formation of concepts.

Example 13.1 *Some Logistics System Domain Description Units:* We bring some (nonindexed) domain description units gathered when examining a freight logistics domain:

- *Freight can be delivered to trucking depots, railway freight terminals, harbours and air cargo centres.*
- *Freight can be transported by trucks, freight trains, ships* and *Trucks move along highways, trains along railway lines, ships along sea lanes and aircraft along air corridors.*
- *Freight may be transferred between trucks, trains, ships and aircraft at trucking depots, railway freight terminals, harbours, and air cargo centres, and between trucks, trains, ships and aircraft at such places where trucking deports coincide with railway freight stations, harbours and air cargo centres, etc.*
- *Freight can be picked up from trucking depots, railway freight terminals, harbours and air cargo centres.*

The above description units are construed, i.e., made to order. ▪

Characterisation. *Domain Concept Formation:* By *domain concept formation* we understand the abstraction of domain phenomena into concepts. ▪

Whereas description units may refer to categories of immediately instantiable, that is, designatable (phenomenological), entities, concepts usually refer to classes of such categories.

Example 13.2 *A Logistics System Analysis and Concept Formation:* We bring in first some analysis and then we form concepts (from the above exemplified description units).

- *Trucking depots, railway freight terminals, harbours and air cargo centres are all of the same kind: They (temporarily) store (delivered or transferred) freight for further transport or delivery. Hence we shall abstract them into one class: hubs.*
- *Trucks, trains, ships and aircraft are all of the same kind: They move freight. Hence we shall abstract them into one class: conveyor vehicles.*

- *Highways, railway lines, sea lanes and air corridors are all of the same kind: They are "paths" along which freight can be conveyed. Hence we shall abstract them into one class: routes.*

Now you see how our "made to order" examples of Example 13.1 are being disposed of in this example. ▪

Concept formation usually leads to much simpler descriptions.

Example 13.3 *A Partial Logistics System Narrative:*

- *Freight can be delivered to hubs.*
- *Freight can be transported by conveyor vehicles.*
- *Trucks move along routes.*
- *Freight may be transferred from one conveyor vehicle to another conveyor vehicle at hubs.*
- *Freight can be picked up at hubs.*

Our simple sequence of three sets of construed examples, Examples 13.1–13.3, has been brought to an end. ▪

Concept formation is for the experienced engineer, and it is an art — but some principles and techniques can be taught and learned. Principles and techniques of abstraction apply here — to their fullest — and form the major principles and techniques used in concept formation. As we have covered these principles and techniques of abstraction elsewhere[1] we shall not cover abstraction much here. But we remind the reader that in reading domain descriptions we shall seek out texts wherever a phenomenon [or a concept] can be abstracted (into a [further abstracted] concept), or two or more phenomena or concepts can be "merged", i.e., abstracted into one concept.

Each abstracted concept needs to be carefully defined. Concept formation (i.e., identification) then necessitates a rewriting of domain descriptive texts into such where the concepts replace that which they abstract, as well as the insertion of the defined concepts into a terminology.

13.2.2 Breakthrough Abstracted Concepts

The trouble, in a sense, with the "simply abstracted" concepts is that they really do not introduce anything new! Well, why must things be new? Well, sometimes "breakthrough" concepts may turn our world upside down and simplify matters considerably.

It is obvious that we need some informal examples here.

[1] We cover abstraction principles, techniques and tools in Vols. 1 and 2 of this series of textsbooks on software engineering.

Example 13.4 *Breakthrough Concepts:*

Consider the year 1925, or so. Flying aircraft over or near the Magnetic North Pole (currently, 2006, at 82.7 latitude °north and 114.4 longitude °west) was not possible. If one had asked scientists to devise a gadget to help on this matter they might conceivably well have invented a very involved and tricky sextant to have the aircraft guided by the stars. Instead the inertial guidance concept based on gyroscopes, solved the, problem. That is, an abstraction from conventional physical astronomical gadgets had replaced these by, in a sense, other physical gadgets which one would commonly not associate with astronomy.

Consider the year 1880. Performing surgery on the brain — to see what a problem was and, if necessary, to remove spots of cancer — was not possible. But prospects were looming. Things improved so much that a call for designing suitable instruments for such surgery might have led to very intricate mechanical and manually operated minirobots of saws, scissors and scalpels. Instead, some time later X-ray techniques replaced "all that". That is, the phenomenon of X-rays replaced a phenomenon of a mechanical minirobot of saws, scissors and scalpels. An abstraction from having to open up the skull and instead penetrating it invisibly had taken place.

From our own universe of discourse: Computing and, in particular, software, let us consider function abstraction: that of lifting a function from A into B to become a temporal function from Time into functions from A into B. Or, making the data type X of Y elements into an abstract, parameterised data type X(E) of instantiable E elements, one instance being, for example, Y, another being Z. The idea is also shown in Sect. 11.7 on Scripts, where we can factor out the ready-made, "baked into the banking software system", banking transactions into programs of a separately "programmable" bank script language, which could, perhaps, qualify as a discovery. ∎

But, in fact, we have something far more "revolutionary" in mind when we refer to concepts as possibly being innovative discoveries, i.e., breakthroughs. These types of things are not done, i.e., made, really, in the daily life of a software engineer. We cannot plan for them. We can, however, plan for and expect "delivery" of simply abstracted concepts. The breakthroughs one can hope for. They become corporate treasures, which may not be tradeable, but are of advantage in competitive situations.

13.3 Consistencies, Conflicts and Completeness

Conflicts are one form of inconsistency. "Remaining" inconsistencies can be resolved between domain stakeholders — as aided by the domain engineers. Completeness is a relative issue. We shall cover these three ideas now.

13.3.1 Inconsistencies

Characterisation. By *inconsistency* of a *domain description* we shall understand some pairs (or more) of texts where one text describes one (set of) property (properties), while another text (of the pair or more) describes (describe) an "opposite" property (set of properties), that is, property P and property not P. ▪

Example 13.5 *A Domain Description Inconsistency:* The following two non-indexed description units express an inconsistency: *Between 5 am and 2 pm trains run at 12-minute intervals;* and *Between 11 am and 7 pm trains run at 15-minute intervals.* ▪

The P in the above example is *Between 11 am and 2 pm trains run at 12-minute intervals;* and the not P in above example is *Between 11 am and 2 pm trains do not run at 12-minute intervals.*

Current research, basically with a base in requirements engineering, investigates (and the same researchers also propose engineering solutions to) problems of more or less automatically detecting inconsistencies in sets of domain description (or, rather, requirements prescription) units. Such proposals imply the representation of the de/prescription units in logical form, and the proposals then focus on the particulars of such forms and on what tool support can then be mechanised. We thus alert the reader to look out for viable tools of this type, if and when they become commercially available and supported.

13.3.2 Conflicts

Characterisation. By a *domain description conflict* we shall understand a domain description inconsistency, in which some domain stakeholders strongly adhere to one set of domain descriptions, which are inconsistent with another set of domain descriptions, strongly adhered to by other domain stakeholders — and such that this conflict can only be resolved through negotiation, including possible business process reengineering, between up to several levels of management. ▪

Inconsistencies which can be resolved (i.e., removed) by discussion between domain stakeholders are not conflicts. Conflicts are usually deeply rooted, and are not within the prerogative of the domain engineers to solve.

13.3.3 Incompleteness

Characterisation. By *incompleteness* of a *domain description* we shall understand a description which leaves open some of the values of some entities, some of the function argument/result value relations or some of the process behaviours; or which indicates some alternative possibilities of entity attributes,

function argument/result pairs or behaviour alternatives, without indicating (i.e., describing) all (obvious) such. ∎

Example 13.6 *A Domain Description Incompleteness:* We give a few examples:

Adult passengers pay full fare, children half. An incompleteness could be that there are no domain descriptions for what is meant by passengers and hence by adult and child passengers, or that, say, soldier, student and pensioner transport fees are not covered.

A free rail unit is one which is available for scheduling. A locked train unit is one which has been scheduled but is as yet not occupied by a train. An occupied train unit has a train passing it. An incompleteness could be that there are no descriptions of, say, rail units that have just been passed by a train but are possibly not yet (made) available for scheduling. ∎

Incomplete domain descriptions may be made complete, or left as such. Final incomplete domain descriptions may be a source for expressing domain requirements that "repair" any conceived incompletenesses.

13.3.4 Looseness and Nondeterminism

Characterisation. By a *loose domain description* we shall understand the same as an incomplete domain description, but one in which that looseness has been left so on purpose. ∎

Characterisation. By a *nondeterministic domain description* we shall understand a description which deliberately leaves open such things as function argument/result values, choice amongst alternative behaviours, ordering of events, etc. ∎

Example 13.7 *Nondeterministic Domain Descriptions:* We give some examples:

Persons are not further defined.

Function f when applied to any integer value yields a value 3 or more.

Users issue series of inquiry and order commands in any order. ∎

13.4 From Analysis to Synthesis

Once a reasonably thorough analysis and concept formation has been done, the domain engineer can create a domain description, i.e., a domain model, according to the principles and techniques outlined in earlier sections. In this part of the book, which consists of several chapters on domain engineering, we

have covered principles and techniques of the major issues of domain descriptions, i.e., synthesis, before we covered analysis in this chapter. The reason, to repeat, for covering principles and techniques of some development stages in the reverse order of their actual sequence in domain engineering is that we must first know what should go into, i.e., what can constitute, a domain description before we cover the principles of domain acquisition. And analysis follows acquisition.

13.5 Discussion

13.5.1 General

This chapter is closely related to Chap. 21: Requirements Analysis and Concept Formation. The reader is advised to carefully review this material before reading Chap. 21. After having studied Chap. 21, the reader is encouraged to compare the two chapters: 13 and 21.

Which tool support is available for discovering inconsistencies and incompleteness? To answer that question in the positive, we need to reflect a bit on the form of our description units: If they are formulated in an informal language, a notation without any formal semantics or proof system, then there is only the human brain and informal, but precise reasoning to resort to. If they are formulated in a formal language, a notation with a formal semantics, or a proof system, then model checking and theorem proving may be attempted. We shall, in the present edition of these volumes, not venture further into this important area, other than saying that there do indeed exist tools and techniques for assisting in the discovery of domain inconsistencies and incompleteness.

13.5.2 Principles, Techniques and Tools

We summarise:

Principles. The principle of *domain analysis* applies to domain descriptions and shall ensure that descriptions are consistent, relatively complete and based on the "narrow bridge" principle of pleasing sets of a relatively small set of designations and defined concepts. ∎

Principles. The principle of *concept formation* applies to domain descriptions and shall ensure that descriptions are based on "telling", i.e., pleasing concepts. ∎

Techniques. When applied to informal domain descriptions, including informal description units, the *domain analysis* techniques are likewise informal. When applied to formalised domain descriptions, including informal description units, the preferred techniques focus on 'abstract interpretation': from

static data and control flow analysis, to symbolic "execution" of the formal descriptions texts, followed by human interpretation of the results. ▪

Techniques. The techniques of discovering concepts are, naturally, those of exploration and experimentation, of scepticism, and of conjecturing and refuting. These investigative techniques again require further techniques, usually those of an inquisitive, scientific mind. ▪

The reader will have discerned, in the characterisation of concept formation, that (human) ingenuity is called for. So be it.

Tools. When *domain analysis* is applied to informal domain descriptions, including informal description units, the tools are likewise informal: human brainpower, and good text-processing facilities. When domain analysis is applied to formalised domain descriptions, including informal description units, the tools include those that can perform abstract interpretation, i.e., flow analysis on formal texts. ▪

Tools. The only *concept formation* tool "presently available" is that of human brainpower. ▪

13.6 Bibliographical Notes

The following researchers — who have contributed significantly to the field of requirements engineering — are mentioned in this chapter due to the tool-oriented nature of their work and its relevance also to domain engineering: John Mylopoulos, A. Borgida, L. Chung, S.J. Greenspan, and E. Yu: [126, 127, 253–255, 380]; Joseph A. Goguen, M. Girotka and C. Linde: [123, 124]; and Axel van Lamsweerde, A. Dardenne, R. Darimont, M. Feather, S. Fikas, R. De Landtsheer E. Letier, C. Ponsard and L. Willemet: [76–78, 102, 211, 214, 215, 360–366].

13.7 Exercises

13.7.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

13.7.2 The Exercises

Exercise 13.1 *Domain Inconsistencies.* For the fixed topic, selected by you, try to create 3 or 4 examples of inconsistent domain description units.

Exercise 13.2 *Domain Conflicts.* For the fixed topic, selected by you, try to create, say, two examples of conflicting domain description units.

Exercise 13.3 *Domain Concepts.* For the fixed topic, selected by you, try to create a number, say, four or more domain description units from which, by simple analysis, you can come up with at least two concepts.

14

Domain Verification and Validation

- The **prerequisite** for studying this chapter is that you have a reasonable grasp of the previous stages of domain engineering: from domain acquisition, via analysis and concept formation, to domain description (i.e., domain modelling).
- The **aims** are to briefly introduce the concepts of domain verification (including model checking and testing) and validation, and to cover some of the attendant principles and techniques.
- The **objective** is to complete your education and training so as to become a professional domain engineer.
- The **treatment** is informal.

The Right Domain — The Domain Right [42]

- Domain Validation: Validate to get the right domain.
- Domain Verification: Verify (model check, test) to get the domain right.

14.1 Introduction

Let us first review where we are in the process of describing the domain development process and its method principles and techniques:

(i) First we focused on the core aspects of domain modelling: The "whats" and "hows" of a domain model. We could call this the "production technology". (i.1) We covered the concepts of abstraction of phenomena and concepts in Chap. 5, and (i.2) the attributes and facets of what is being described in domain models in Chaps. 10–11. (i.3) We covered, in between, the issues of stakeholders and their perspectives in Chap. 9. That coverage explained "what" a domain model should contain, the abstractions possible, the facets "mirrored", and — notably — with respect to the stakeholders and the perspectives to be dealt with.

(ii) Then we focused more on "how". In contrast to "production technology" we could call this "how" the "process technology". (ii.1) First, we focused on the process, principles and techniques of domain acquisition, Chap. 12, that which "begins" the domain development work. (ii.2) Then we covered the process, principles and techniques of domain analysis and concept formation in Chap. 13. After domain acquisition, domain analysis and concept formation follows the domain modelling proper. Finally, we focus on domain validation and verification — the topic of this chapter.

The purpose of the above review has been to put the somehow "reverse" ordering of the chapter sections "straight" with respect to the ordering of the domain development processes. We can now summarise the domain development process (even before we have covered the notions of verification and validation). After producing the appropriate informative documents: needs and ideas, concepts, scope and span, synopsis, and contracts, one proceeds to identifying domain stakeholders and establishing liaison with members of domain stakeholder groups. Then we move on to domain acquisition: interviews, studies, questionnaire formulation and domain stakeholders' replies to these, ending wit domain description unit indexing and an elicitation report. This acquisition is followed by domain analysis and concept formation. Then we do the actual domain modelling. And, finally, we perform domain verification and validation.

14.2 Domain Verification

In this chapter (as we shall also do in Chap. 22 on requirements validation and verification) we use the term verification to also cover the concepts of model checking and testing.

Characterisation. By *domain verification* we shall understand a process, and the resulting (analytic) documents, in which some domain descriptions are being analysed in order to ascertain whether what is being described satisfies certain (claimed or otherwise expected) properties. ∎

So what — really — is the difference between domain validation and domain verification?

- In validation we examine the domain model to make sure we are modelling what the domain stakeholders think that domain is: *Validation gets the right domain model.*
- In verification we examine whether our domain model "hangs together," such as the domain engineers want it to be: *Verification gets the domain model right.*

(The above, oft-quoted distinction was, it seems, first reported by Barry Boehm in [42].)

Verification is adjoint to validation: Both validation and verification are needed. Usually verification precedes validation. Verification work typically proceeds as follows: Desired properties of the domain model — properties that do not transpire immediately from the domain description — are formulated, informally or formally. Then "proofs" by "verbal" arguments, or some form of symbolic testing, or formal proofs, or model checking, are performed in order to check that the desired property holds of the domain model.

So verification, to us, includes, rearranging the terms a bit, (i) informal reasoning: (i.1) "proofs" by "verbal" arguments and (i.2) testing; (ii) formal reasoning: (ii.1) formal proofs and (ii.2) model checking. By informal reasoning we shall, however, mean "proofs" by "verbal" arguments.

14.2.1 Informal Reasoning

Characterisation. By *informal reasoning* we shall understand a carefully phrased series of arguments, which, as a whole, convinces an audience of the validity of what is concluded. ▪

Human beings often reason, but are not always careful in doing so. Informal reasoning demands great care.

14.2.2 Testing

Characterisation. By *testing* we shall understand that a domain description is provided with set values for all relevant arguments (the test data), with the description then being evaluated ("executed") for those arguments. The test then results in a "final value" of the description for those arguments. ▪

Such a "final value" may be a complicated quantity. Typical final values could be an execution sequence, or a trace of description points, with a set of variable values for each step in the sequence (i.e., a trace).

In another way of phrasing it: Testing is a systematic search for a counterexample to a claim (or proof) of correctness. Testing, till recently, has basically been a heuristics-based science. An important part of testing is text analysis. If domain description parts have been formalised, then theory-based testing technologies have been or can be developed and can be used for testing. Chapter 29, Sect. 29.5.3, covers testing in more detail.

14.2.3 Formal Proofs

Characterisation. By a *formal proof* we shall understand a given domain description, a statement (a theorem) to be proved and a proof that the domain description satisfies the statement: This proof refers to a proof system for the language in which the domain description is expressed (axioms and inference

rules), and is otherwise a sequence, composed from steps, where each step in the sequence is like a theorem (a lemma), a statement, and where pairs of steps in the proof sequence are related, i.e., are justified, by the axioms and the inference rules. ▪

14.2.4 Model Checking

Characterisation. By *model checking* we shall understand [55] *a method for formally verifying usually concurrent systems, whose usually extremely large, practically speaking infinite state systems, have been reduced to manageable finite-state systems.*

We augment this characterisation by the following: In model checking a somehow executable abstraction of the thing to be checked is programmed. That model is then subject to certain forms of executions in which specified properties are checked. These executions, for example, check whether the model is able to enter certain states or not. ▪

Domain descriptions about such finite-state systems are typically expressed as temporal logic formulas. Efficient symbolic algorithms are used to traverse the (state machine) model defined by the system and to check if the domain description holds or not, i.e., whether the model execution "enters" appropriate states, albeit for a "reduced" set of possible states of systems. Extremely large state-spaces can often be traversed in minutes. Seminal books on model checking are [26, 59, 173].

14.3 Domain Validation

Characterisation. By *domain validation* we shall understand a process, and the resulting (analytic) documents, in which some domain descriptive documents are being coinspected by domain stakeholders and domain engineers, and in which whatever is being described is being positively and/or negatively reviewed with reference to the elicitation report and with respect to whatever the stakeholders might now realise about their domain. This includes pointing out, if necessary, inconsistencies, incompletenesses, conflicts and errors of description that may change the elicitation report. ▪

Domain validation is possibly interwoven with domain verification work, see Sect. 14.2.

14.3.1 The Domain Validation Documents

In order to perform domain validations, the validators need the following (input) documents: (i) the list of domain stakeholders; (ii) the domain acquisition

documents: questionnaire, and the collection of indexed description units; (iii) the rough-sketch, terminology, narrative, and possibly — if produced — the formalisation documents that constitute the domain description proper; and (iv) the domain analysis and concept formation documents. That is, the validators need access to basically all documents produced (so far) in the domain modelling effort.

In order to complete domain validation, the validators produce the following (output) documents: (i) a possibly updated domain stakeholder document; (ii) possibly updated domain acquisition documents; (iii) possibly updated rough sketches, terminology, narrative, and — if relevant — the formalisation documents; (iv) possibly updated domain analysis and concept formation documents; and (v) a domain validation report. We now cover some aspects of the necessarily informal validation process.

14.3.2 The Domain Validation Process

Domain validation proceeds as follows: Domain engineers "sit together" with stakeholders and review, line by line, the domain model, holding it up against the previously elicited domain description units, while then noting down any discrepancies. In doing domain validation, domain stakeholders usually read the informal, yet precise and detailed narrative descriptions. No assumption is made as to their ability to read formalisations. On the contrary: It is assumed that they cannot read formal specifications.

For reasonably large-scale projects the customer may hire professional consultants who can also study the formalisations. This is just like future ship owners hiring Lloyd's Register of Shipping [224][1] to check ship designs in preparation for insurance companies to take on insurance risks.[2]

Domain validation (and verification) ends with a signed domain validation (and verification) report. This report either OKs the domain model, or points out required corrections in the elicitation report, in the domain analysis and concept formation report, and in the domain model.

14.3.3 Domain Development Iterations

Thus domain validation (and verification) can be an iterative process, alternating possibly with further domain verification, further elicitation report work, further domain analysis and concept formation work, and with further domain modelling work. The domain validation process may end with further domain validation (and verification) work.

[1] Or such similar companies as Norwegian Veritas [262], Bureau Veritas [51] or TÜV [356].

[2] The staff of these, and similar design quality assurance companies, are oftentimes very sophisticated software engineers, well-versed in formal software development and verification methods.

14.4 Discussion

14.4.1 General

This chapter is closely related to Chap. 22: Requirements Validation and Verification. The reader is advised to carefully review this material before reading Chap. 22. After having studied Chap. 22, the reader is encouraged to compare the two chapters: 14 and 22.

We have treated aspects of domain validation and verification — in the same chapter since they relate in many ways. And we have used the term verification, primarily to stand for formal proofs, but, secondarily, also for model checks and tests.

14.4.2 Principles, Techniques and Tools

We summarise:

Principle. *Domain Validation:* To ensure that the domain described is the right domain. ▪

Principle. *Domain Verification:* To uncover a domain theory, i.e., to get the domain descriptions right. ▪

Techniques. *Domain Validation:* In summary, human, collaborative document inspection (Sect. 14.3.2). ▪

Techniques. *Domain verification* techniques, based on formal descriptions, include those that enable formal verification (of posed lemmas and theorems), model checking, and tests, while domain verification techniques, based on informal descriptions, basically amount to informal, concise reasoning. ▪

Tools. Since *domain validation* is basically an informal process, the tools are those that support document cross-referencing between domain description units and narrative domain descriptions and domain terminologies, and data mining based on such documents. ▪

Tools. *Domain verification* based on formal descriptions requires such tools as, for example, proof assistants and theorem provers, model checkers, and test generators and tester monitors; whereas domain verification based on informal descriptions basically requires human reasoning. ▪

14.5 Exercises

14.5.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

14.5.2 The Exercises

The first 4 exercises (14.1–14.4) of this chapter are *closed book* exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions.

Exercises 14.6 and 14.5 test the problem solver's ability to lead a group of two or more domain validators, respectively domain verifiers.

Exercise 14.1 *Domain Validation Documents.* Which are the domain development documents that are needed in order to commence proper domain validation, and which are the resulting documents?

Exercise 14.2 *Domain Validation Process.* Outline, in brief, i.e., in a few itemised lines, the domain validation process.

Exercise 14.3 *Domain Verification, Model Checking and Testing.* Explain, in brief, in a few itemised lines, the concepts of formal verification, of model checking and of testing.

Exercise 14.4 *Domain Validation Versus Domain Verification.* Explain in two itemised lines the difference between the objectives of domain validation and domain verification.

Exercise 14.5 *Domain-Specific Verification.* For the fixed topic, selected by you, and on the basis of your solutions to some of Exercises 10.1–10.7, and/or some of Exercises 11.1–11.7, suggest, preferably three or more, issues that may need special attention during domain verification.

Exercise 14.6 *Domain-Specific Validation.* For the fixed topic, selected by you, and on the basis of your solutions to some of Exercises 10.1–10.7, and/or some of Exercises 11.1–11.7, suggest, preferably three or more, issues that may need special attention during domain validation.

15

Towards Domain Theories

- The **prerequisite** for studying this chapter is that you have now studied the chapters on domain engineering.
- The **aims** are to very briefly indicate what a domain theory might be, to suggest how a domain theory might be established, and to outline what uses a domain theory might be put to.
- The **objective** is to help ensure that a number of domain theories indeed be established.
- The **treatment** is discursive.

15.1 Introduction

We cannot design software before we have a reasonable grasp of the requirements put to that software. We cannot express requirements before we have a reasonable grasp of the domain in which that software is to serve:

- An automotive engineer, when designing an automobile transmission system, makes extensive use of basic laws of the theories of mechanics, and would not be hired unless he had a certified, deep knowledge of the laws of mechanics.
- A radio communications engineer, when designing a radio antenna, makes extensive use of the theories relating to Maxwell's Equations, and would not be hired unless she had a certified, deep knowledge of the laws of electromagnetic wave propagation.
- A civil engineer, when designing a bridge, makes extensive use of the theories of structural statics, and would not be hired unless he had a certified, deep knowledge of the laws of structural statics.
- An aerospace engineer, when designing, say, a supersonic aircraft, makes extensive use of the theories of aerodynamics, and would not be hired unless she had a certified, deep knowledge of the laws of aero-, thermo- and hydrodynamics.

- A software engineer is, today, often asked to develop software for such diverse fields as transportation (including railways), healthcare, financial services, production (manufacturing), the e-market (consumers, retailers, wholesalers, producers and distribution), etc., without having any theories about transportation, healthcare, financial services, production, sales, marketing and logistics to refer to. Moreover, the software engineers are not expected to be knowledgeable about any such theories.

Clearly this situation is not acceptable. In this chapter we shall, ever so briefly, outline the idea of 'domain theories'.

15.2 What Is a Domain Theory?

A theory, to us, is a mathematical logic structure consisting of a number of axioms, inference rules and theorems proved from these. A domain theory is a theory, one of whose models is a particularly identified, abstracted or instantiated domain. Typically a domain theory is based on a domain model — expressed in some one or more formal specification languages — with the axioms and inference rules being of those languages, and the (zero, one or more) theorems being (further) statements about properties of the domain model.

Characterisation. By a *domain theory* we we understand a theory of an application domain. ▪

That is: a set of axioms, some induction rules, and a set of lemmas and theorems proved from the axioms and induction rules.

So a domain theory applies to a very specifically instantiated domain: *that hospital over there.* Or a domain theory applies to a generic, i.e., large class: *all hospital systems.* Or a domain theory applies to some small class: *the Danish hospital system.*

15.3 Example Statements of Domain Theories

We state some examples of theorems that could be proven to hold of suitable domain models for the example domains hinted at below.

Example 15.1 *Railways:*

1. *God does not play dice:* Two trains travelling down a railway line, from one station to another, do not instantaneously change position.

2. *Kirchhoff's Law:* Under the assumption that train timetables specify train traffic modulo 24 hours, under the assumption that trains are on time and under the assumption that trains do not suddenly disappear, we have that the number of trains entering a station, in a 24-hour period, minus the number of arriving trains taken out of service (i.e., ending their journey) at that station, plus the number of trains starting their service at that station, equals the number of trains departing that station — for all stations.
3. *No ghost trains:* If at any two times any specific train is part of the traffic at those times, then any such train is also part of the traffic at all times in-between those two times.

We refer to Exercise 15.12. ∎

Example 15.2 *Freight Logistics:* Freight logistics is about freight being sent from one hub to another hub via zero, one or more, but a finite number of hubs, along links between hubs.

4. *Life is like a sewer, what you put into it is what you get out of it (I):* Under the assumption that all freight delivered (in)to a freight logistics system can always be accounted for,[1] we have that the number of freight items en route in a freight logistics system is the difference between the number of freight items entered into the freight logistics system and the number of freight items delivered from the freight logistics system.
5. *Monotonic Progress:* Under the assumption that freight bills of lading are followed punctiliously, and that such waybills indeed specify some form of (economically cheapest, or temporally (i.e., timewise shortest), or distancewise shortest) "straightest" route and that freight is not held up indefinitely at some hub, we have that freight is transported progressively towards its ultimate destination.

We refer to Exercise 15.7. ∎

Example 15.3 *Healthcare:*

6. *No cloning:* A patient, a healthcare worker, or a visitor at a hospital is physically present at one location at most within the hospital at any one time.

We refer to Exercise 15.8. ∎

[1] Missing freight when dropped off a ship, a truck or a freight train, when noted missing, is, indeed, "accounted for".

Example 15.4 *Financial Services:*

7. *No money printing:* Financial transactions between financial institutions (transfers of monies between banks, or to or from insurance companies, stockbrokers, portfolio managers, etc.) do not themselves "generate monies": The sum total of monies within the system is unchanged — money is only "moved".
8. *Life is like a sewer, what you put into it is what you get out of it (II):* The only changes in the sum total of monies of a financial system (of banks, insurance companies, stockbrokers, funds managers, etc.) is when clients residing outside this system deposits or withdraws funds.

We refer to Exercise 15.6. ▪

15.4 Possible Domain Theories

We have already hinted at many possible domain theories. See, for example, Sect. 11.2.3. Alphabetically listed, some are:

- *Air traffic:* The "system" of air space (including the landing and take off runways of airports, and their tarmacs — the latter including some part of the gates), aircraft in that space (entering when taking off, or leaving when landing, or when accidentally falling to the ground), and the configurations of ground, terminal, area and continental control centres, together with all their externally observable events within as well as between these "system" components, could form a domain. Some of these events trigger actions, such as: handling of a start flight request (an event), takeoff, change flight course, landing (preparation for a touchdown event, as well as the immediate actions after that event), etc.
- *Airport:* The "system" of check-in counters, baggage handling (to and from aircraft, and baggage delivery), security checks, possibly passport (or similar) checks, gates, catering, aircraft cleaning service, aircraft fuelling, etc., together with all their externally observable events within as well as between these "system" components, could form a domain. Some of these events trigger actions, such as: check-in, entering (or leaving) a gate, ordering catering, unloading baggage, etc.
- *Financial services:* The system of banks (including a national or federal, etc., bank), insurance companies, stockbrokers and traders, stock exchanges, portfolio managers, and the external clients of these "components" (bank account holders, insurance holders, buyers and sellers of securities instruments, etc.), as well as the externally observable events within as well as between these "system" components and between these and their clients, could form a domain. Some of these events trigger ac-

tions, such as: opening an account, depositing monies, withdrawing monies, transferring monies, buying or selling stocks, etc.

- *Freight logistics:* The system of senders and recipients of freight, of conveyor companies (trucking firms, freight railways, shipping companies, air cargo companies), of routes (road net, rail net, shipping lanes and air lanes), of hubs (truck depots, railway stations, harbours, airports), as well as the externally observable events within as well as between these "system" components and between these and their clients, could form a domain. Some of these events trigger actions, such as: inquiring of, and ordering freight conveyance, delivering or receiving freight (by a sender, respectively a recipient), unloading freight at a hub from a conveyor to the hub, similarly for loading, etc.
- *Railways:* The system of a rail net (lines and stations), of rolling stock, of timetables, of train traffic, passengers, and freight, as well as the externally observable events within as well as between these "system" components and between these and their clients, could form a domain. Some of these events trigger actions, such as: planning changes to the rail net, planning new timetables, scheduling and allocating rolling stock, maintenance, staff rostering, signaling and switching, buying passenger tickets, entering trains, etc.

15.5 How Do We Establish a Theory?

Well, there are several parts to an answer to the question posed in the above section title.

- *Domain engineering:* On one hand, do as prescribed in this part of the book. But do more than we have shown: Establish "interesting" theorems and prove them; interact with stakeholders of the domain.
- *Collaborative science:* On the other hand, gather a consortium of research and domain engineer colleagues around the world interested in the same domain, and get them to work together. Establish some bases for commonality and interfaces for collaboration and exchange of documents. Do not forget to interact with stakeholders of the domain.
- *Social science:* Arrange workshops, publish papers, etc. Present your work. Get it accepted, or, as the case may be, rejected, by the relevant communities: the computing scientists, the possible domain scientist/researchers, and the stakeholders.

For an example of a current attempt to establish a domain theory, "surf" to the following Web site:

- `http://www.railwaydomain.org`

As can be observed from the above, and from the referenced URL, to undertake establishing, researching and developing a domain theory is, in general, a "grand challenge". We refer to Sect. 32.4.2.

15.6 Purpose of a Domain Theory

We repeat reasons for establishing domain models, and hence domain theories, which were first given in Sect. 4.3.

- *to gain understanding,*
- *to get inspiration and to inspire,*
- *to present, educate and train,*
- *to assert and predict* and
- *to implement* — the latter both as a basis for business process reengineering and for software requirements.

One more, perhaps altruistic, reason can be given:

- *To further science* — in the sense of discovering new modelling method principles and techniques, including those that enable a model to be a composition of several models expressed in different formal languages, hence requiring the integration of their formal foundations.

15.7 Summary Principles, Techniques and Tools

We summarise below.

Principles. The principles of *domain theory* are to gain understanding, to get inspiration and to inspire, to present, educate and train, to assert and predict and to implement. ▪

Techniques. The techniques of establishing *domain theories* are those of creating any domain model. They encompass domain acquisition techniques, domain analysis and concept formation techniques, domain modelling techniques, domain validation techniques and domain verification techniques. ▪

Tools. The tools for establishing *domain theories* are those needed when creating any domain model, and encompass all the tools otherwise advised upon in these three volumes on software engineering. ▪

15.8 Bibliographical Notes

We refer to [34, 122, 236, 265, 272, 292, 306, 313] for an example of early papers on the start of a domain theory for railways. We also refer to `www.railwaydomain.org`. It is expected that this international research effort will begin to "build up" during 2006, to develop a domain theory for railway systems.

15.9 Exercises

15.9.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

15.9.2 The Exercises

To provide a setting for the exercises of this chapter we first refer to Sect. 1.7.1's enumerated list of exercises:

1. *What is Administrative FormsProcessing?*
2. *What is an Airport?*
3. *What is Air Traffic?*
4. *What is a Container Harbour?*
5. *What is a Document System?*
6. *What is a Financial Services System?*
7. *What is Freight Logistics?*
8. *What is a Hospital?*
9. *What is a Manufacturing Company?*
10. *What is the Market?*
11. *What is a Metropolitan Area*[2] *Tourism Industry?*
12. *What is a Railway System?*
13. *What is a University?*
14. *What is a Public Administration?*
15. *What is a Ministry of Finance?*

Exercise 15.1 *Towards an 'Administrative Forms Processing' Domain Theory.* We refer to our brief rough-sketch outline item 1, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for an administrative forms processing domain theory.

Exercise 15.2 *Towards an 'Airport' Domain Theory.* We refer to our brief rough-sketch outline item 2, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for an 'Airport' domain theory.

Exercise 15.3 *Towards an 'Air Traffic' Domain Theory.* We refer to our brief rough-sketch outline item 3, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for an 'Air Traffic' domain theory.

Exercise 15.4 *Towards a 'Container Harbour' Domain Theory.* We refer to our brief rough-sketch outline item 4, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Container Harbour' domain theory.

[2] Such cities as Singapore, Macau, Hong Kong, London, New York, Tokyo, Paris, etc., can be said to be metropolitan areas.

Exercise 15.5 *Towards a 'Document System' Domain Theory.* We refer to our brief rough-sketch outline item 5, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Document System' domain theory.

Exercise 15.6 *Towards a 'Financial Services System' Domain Theory.* We refer to our brief rough-sketch outline item 6, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Financial Service System' domain theory.

Exercise 15.7 *Towards a 'Freight Logistics' Domain Theory.* We refer to our brief rough-sketch outline item 7, Sect. 1.7.1. Suggest more theorems (like those stated in Examples 15.1–15.3) for a 'Freight Logistics' domain theory.

Exercise 15.8 *Towards a 'Hospital' Domain Theory.* We refer to our brief rough-sketch outline item 8, Sect. 1.7.1. Suggest more theorems (like those stated in Examples 15.1–15.3) for a 'Hospital' domain theory.

Exercise 15.9 *Towards a 'Manufacturing Company' Domain Theory.* We refer to our brief rough-sketch outline item 9, Sect. 1.7.1. Suggest more theorems (like those stated in Examples 15.1–15.3) for a 'Manufacturing Company' domain theory.

Exercise 15.10 *Towards a 'Market' Domain Theory.* We refer to our brief rough-sketch outline item 10, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Market' domain theory.

Exercise 15.11 *Towards a 'Metropolitan Area Tourism Industry' Domain Theory.* We refer to our brief rough-sketch outline item 11, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Metropolitan Area Tourism Industry' domain theory.

Exercise 15.12 *Towards a 'Railway System' Domain Theory.* We refer to our brief rough-sketch outline item 12, Sect. 1.7.1. Suggest more theorems (like those stated in Examples 15.1–15.3) for a 'Railway System' domain theory.

Exercise 15.13 *Towards a 'University' Domain Theory.* We refer to our brief rough-sketch outline item 13 Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'University' domain theory.

Exercise 15.14 *Towards a 'Public Administration' Domain Theory.* We refer to our brief rough-sketch outline item 14, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Public Administration' domain theory.

Exercise 15.15 *Towards a 'Ministry of Finance' Domain Theory.* We refer to our brief rough-sketch outline item 15, Sect. 1.7.1. Suggest theorems (like those stated in Examples 15.1–15.3) for a 'Ministry of Finance' domain theory.

16

The Domain Engineering Process Model

- The **prerequisite** for studying this chapter is that you have studied at least Chaps. 8 and 11 of this part (Part IV).
- The **aims** are to review how the various stages of domain development fit together, and to conclude which documents thus transpire from proper domain modelling.
- The **objective** is to finalise your education and training in domain engineering.
- The **treatment** is informal and systematic.

16.1 Introduction

Domain development, as introduced and covered in this part (Chaps. 8–15) is basically a new element in software engineering. No previously published software engineering textbooks have mentioned this, to us, important and indispensable phase of software engineering, of software development. Domain engineering, as do requirements engineering and software design, possesses a process model, i.e., a model of how the stages and steps of activities of the phase follow one another, iterate, and can or must be done in sequence or in parallel. In this chapter we review this process model. Each stage and step requires documents, in order to be commenced, or results in documents, once completed. We also review these documents or document parts in this chapter.

16.2 Review of Domain Development

The domain description, cum creation, process is just part of the entire domain development process. Figure 16.1 summarises the processes otherwise covered in Chaps. 8–15.

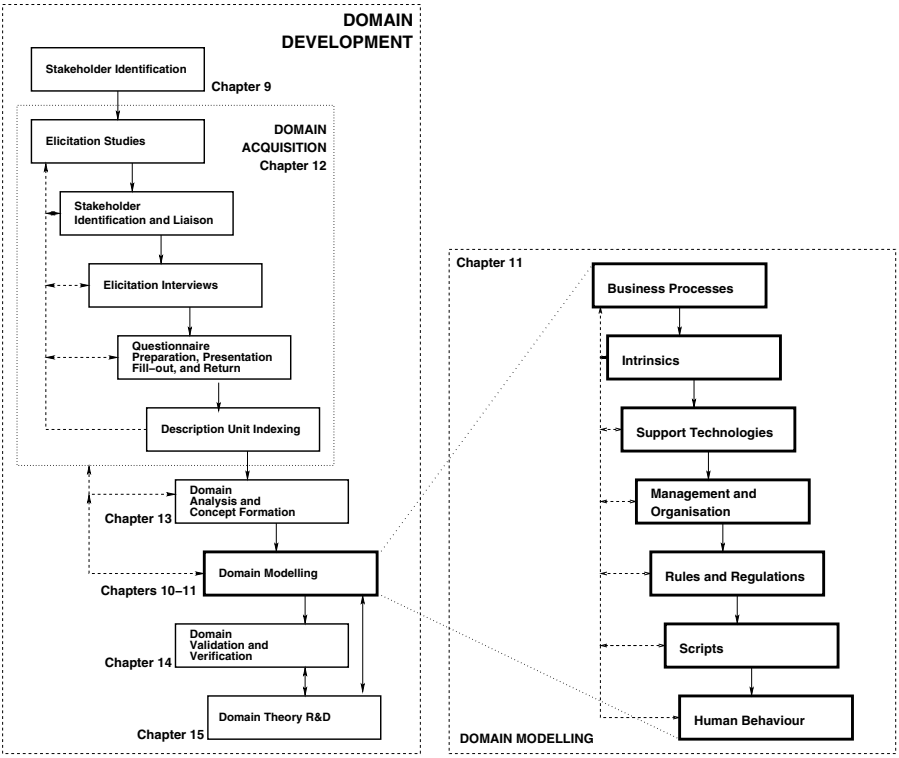


Fig. 16.1. The domain development processes

You are to interpret Fig. 16.1 as follows: The left dashed frame covers all the stages of domain development. It has a time axis, not shown, but — in principle — it starts at the top and propagates to the bottom. The right dashed frame covers just one of the left frame boxes — and does so by detailing its substages or steps. The solid single-arrowheaded lines proceeding in a downward direction designate the ordinary, progressive flow of activities, i.e., they designate a precedence relation between box activities. The dashed double-arrowheaded lines connecting "later" boxes to "earlier" boxes designate iterations between stages. Such iteration is to be very much expected. Iterations are due to "later" discovery of "earlier" "mistakes, errors or omissions".

Sooner — rather than later — iterations stop.

Each box denotes both some development activities and some resulting documents (or document parts).

16.3 Review of Domain Documents

We first list a reasonably complete enumeration of the documents that a proper domain development should produce.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management
      ii. Developers
      iii. Client Staff
      iv. Consultants
2. Descriptions
   (a) Stakeholders
   (b) The Acquisition Process
      i. Studies
      ii. Interviews
      iii. Questionnaires
      iv. Indexed Description Units
   (c) Terminology
   (d) Business Processes
   (e) Facets:
      i. Intrinsics
      ii. Support Technologies
      iii. Management and Organisation
      iv. Rules and Regulations
      v. Scripts
      vi. Human Behaviour
   (f) Consolidated Description
3. Analyses
   (a) Domain Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompletenesses
      iv. Resolutions
   (b) Domain Validation
      i. Stakeholder Walkthroughs
      ii. Resolutions
   (c) Domain Verification
      i. Model Checkings
      ii. Theorems and Proofs
      iii. Test Cases and Tests
   (d) (Towards a) Domain Theory

We then briefly discuss the list.

Chapter 2 covered in detail what is meant by 'Informative Documents' (items 1.a–1.h inclusive) and gave necessary and sufficient advice on what and how to document that kind of information. Chapter 2 also covered in detail what is meant by 'Descriptive Documents' (items 2.a–2.d inclusive), and Chaps. 8–15 gave necessary advice on what and how to document in descriptions. Sufficient advice on formalisation of descriptions is given in Vols. 1 and 2 of this series of textsbooks on software engineering.

Whether you structure the above documents as parts of one (very large) document, or as separately bound documents, say one for 'Information', one for 'Descriptions' and one for 'Analyses' is, in our mind, entirely up to the specific development at hand. What we do advise, though, is to provide your documents, or document parts, with extensive cross-references, say in the form of links and hyperlinks. Everything should be done in order to facilitate browsing and relating, also informally, these many document parts.

16.4 Discussion

We have reached the end of a seemingly long list of different stages of domain development. Each stage is to be considered for possible inclusion in the development actions leading, we hope, to a domain model. Later sections will go into more detailed principles, techniques and tools to be considered by the software cum domain engineer.

Suffice it for now to conclude that if the software cum domain engineer is not aware of the need to identify and classify stakeholders and the many different domain facets, then that domain engineer will have some difficulty in dispatching any domain modelling assignment professionally. We can do better: Domain modelling is not just, partly, the principles, techniques and tools that we shall later detail; it is also, partly, an art.

Part V

REQUIREMENTS ENGINEERING

In the next eight chapters, Chaps. 17–24, we shall cover, in some detail, the principles and techniques of the development stages and steps of the requirements engineering phase:

- Chap. 17, Overview of Requirements Engineering
- Chap. 18, Requirements Stakeholders
- Chap. 19, Requirements Facets
- Chap. 20, Requirements Acquisition
- Chap. 21, Requirements Analysis and Concept Formation
- Chap. 22, Requirements Verification and Validation
- Chap. 23, Requirements Satisfiability and Feasibility
- Chap. 24, The Requirements Engineering Process Model

We refer to Chap. 1 for a comprehensive introduction to the three main phases of software development:

- Domain engineering
- Requirements engineering
- Software design

17

Overview of Requirements Engineering

- The **prerequisites** for studying this chapter are that you are ready to continue the long journey of gaining understanding of the second of the three core phases of software development. You have understood the material of previous chapters, including those on domain engineering, and, preferably, also the (formal) abstraction and modelling principles and techniques of Vols. 1 and 2 of this series of volumes on software engineering.
- The **aims** are to present a capsule view of stages and steps of requirements engineering, and to present a capsule view of the documents that result from requirements engineering.
- The **objective** is to make you feel at ease with the very many stages and steps of requirements development, and the very many parts of resulting documents.
- The **treatment** is informal and systematic.

IEEE Definition of 'Requirements'

By a requirements we understand (cf. IEEE Standard 610.12 [178]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*.

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

Example 17.1 *First 'Requirements' Examples:* We give a few examples of requirements. The examples are very brief, hence they are far from representative of comprehensive requirements prescriptions. The examples below are

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

meant to give some very first hints as to what requirements prescriptions may look like. Take them as rough sketches.

1. **Administrative forms processing:** Office managers shall be able to design forms, aggregations of forms and routines for extracting information from forms and their aggregations.
2. **Airport:** Boarding cards shall be electronic cards that automatically register where in, or near an airport, or in an aircraft the card is located.
3. **Air traffic:** The aircraft tracking system shall alert the terminal control centre operator responsible for handling certain aircraft if any of these deviate significantly from their planned routes.
4. **Container terminal:** The barcode system which registers each and every container subject to unloading or loading shall fail at most once in every 200,000 registrations.
5. **Document system:** Each and every (electronic) document shall contain its entire history: from some original as first created, via all intervening editing and/or copying, etc., including the location, time and person responsible for creation, copying and editing.
6. **Freight logistics:** The freight logistics system, relying on each freight item being suitably equipped with a GPS system responder, is allowed to miss at most one in every 300,000 traced items.
7. **Financial service system:** The stock exchange (system) shall be able to trace all buy and sell orders, as well as all withdrawn such, and all actual transactions, by buyer and seller identification.
8. **Hospital:** The hospitalisation system which to every actual and scheduled patient provides a (flowchart-like) hospitalisation plan, shall be able, at any moment, to estimate and plan for (allocate and schedule) current, immediate and longer-term resource needs: beds, staff (of all categories), medicine, food and beverages, and operating theatres.
9. **Manufacturing company:** For each production cell its current, immediate and longer-term uses, supply of production parts, preventive maintenance schedules, as well as staffing, shall be computable (hence displayable) at any moment.
10. **Market:** Retailer orders with wholesalers, and wholesaler orders with producers (i.e., distributors) shall be automatically issued subject to precisely stated script constraints, and as prompted by "low stock" of certain composites of merchandise.
11. **Metropolitan area tourism:** The MetaTourism system shall enable any suitably equipped (home PC + special GPS, display screen + software controlled mobile phone) person to plan and execute a sequence of visits to places (hotels, restaurants, shops, museums, etc.) and the transport between these.
12. **Railways:** The train monitoring and control system, RaCoSy, being required, shall be able to monitor trains and, if needed, reschedule train traffic, and to do so continuously, and thus to set signals, switches and

train speeds accordingly, and to inform all relevant stakeholders (passengers, train driver, and line and station staff) of any such changes.

The above examples were presented, at this early stage, just to give you a first "feel" for what we are talking about ▪

The "Golden Rule" of Requirements Engineering

Principle. *Requirements Engineering:* Prescribe only those requirements that can be objectively shown to hold for the designed software. ▪

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

An "Ideal Rule" of Requirements Engineering

Principle. *Requirements Engineering:* When prescribing (including formalising) requirements, also formulate tests (theorems, properties for model checking) whose actualisation should show adherence to the requirements. ▪

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

Example 17.1 gave 12 examples of requirements. They all illustrated the need for having a precise description of underlying domains.

Example 17.2 *Analysis of First 'Requirements' Examples:* We analyse the examples of Example 17.1. Our analysis merely consists in listing the domain-specific terms that need to have been precisely described in a prior domain description:

1. **Administrative forms processing:** (i) office managers, (ii) design, (iii) forms, (iv) aggregations of forms, (v) routines (scripts) for extracting information from forms and their aggregations.
2. **Airport:** (i) boarding cards, (ii) where (i.e., airport and aircraft locations).
3. **Air traffic:** (i) terminal control centre operator, (ii) responsible for handling certain aircraft, (iii) aircraft, (iv) deviate significantly, (v) planned route.
4. **Container terminal:** (i) bar-code system, (ii) register, (iii) container, (iv) unloading, (v) loading, (vi) registration.

5. **Document system:** (i) document (ii) [document] history, (iii) original, (iv) created, (v) editing, (vi) copying, (vii) location, (viii) time, (ix) person, (x) responsible.
6. **Freight logistics:** (i) freight logistics system, (ii) freight item, (iii) GPS system responder, (iv) trace.
7. **Financial service system:** (i) stock exchange, (ii) trace, (iii) buy order, (iv) sell order, (v) withdrawals, (vi) actual transactions, (vii) buyer and seller identification.
8. **Hospital:** (i) hospitalisation system, (ii) actual patient, (iii) scheduled patient, (iv) hospitalisation plan, (v) allocate and schedule resources, (vi) current, immediate and longer-term resources, (vii) bed, (viii) staff, (ix) medicine, (x) food and beverages, (xi) operating theatre.
9. **Manufacturing company:** (i) production cell, (ii) current, immediate and longer-term use, (iii) use, (iv) supply, (v) production part, (vi) preventive maintenance schedules, (vii) staffing.
10. **Market:** (i) Retailer, (ii) orders, (iii) wholesaler, (iv) producer, (v) distributors, (vi) ordering ("issued"), (vii) ordering constraint, (viii) "low stock", (ix) composite of merchandise.
11. **Metropolitan area tourism:** (i) person (i.e., potential or actual tourist), (ii) plan, (iii) execute, (iv) visit, (v) place, (vi) hotels, (vii) restaurant, (viii) shop, (ix) museum, etc. (...), (x) transport.
12. **Railways:** (i) monitor train, (ii) reschedule, (iii) train traffic, (iv) set, (v) signal, (vi) switch, (vii) train speed, (viii) inform, (ix) relevant stakeholders, (x) passenger, (xi) train driver, (xii) line staff, (xiii) station staff, (xiv) change.

The above examples were presented, at this early stage, to let you see why we need a precise domain description. ∎

17.1 Introduction

To express requirements is a crucial aspect of overall software development. If we get it even "slightly wrong", the resulting software may be "deadly wrong". The "pitfalls" are legion.[2]

Principle. *Requirements Adequacy:* Make sure that requirements cover what users expect. ∎

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

[2] That is, many, numerous, basically "uncountable".

have been involved in the requirements acquisition process, and that possible conflicts and other inconsistencies have been obviated.

Principle. *Requirements Implementability:* Make sure that requirements are implementable. ▪

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

Principle. *Requirements Verifiability and Validability:* Make sure that requirements are verifiable and can be validated. ▪

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

17.1.1 Further Characterisation of 'Requirement'

From Sect. 1.2.3 we repeat — slightly edited:

Characterisation. By *requirements* we shall understand a document which prescribes desired properties of a machine: (i) what entities the machine shall "maintain", and what the machine shall (must; not should) offer of (ii) functions and of (iii) behaviours (iv) while also expressing which events the machine shall "handle". ▪

17.1.2 The "Machine"

By a machine that "maintains" entities we shall mean: a machine which, "between" users' use of that machine, "keeps" the data that represents these entities. Also from Sect. 1.2.3 we repeat:

Characterisation. By *machine* we shall understand a, or the, combination of hardware and software that is the target for, or result of the required computing systems development. ▪

So this, then, is a main objective of requirements development: to start towards the design of the hardware + software for the computing system.

Principle. *Requirements:* To specify the machine. ▪

When we express requirements and wish to "convert" such requirements to a realisation, i.e., an implementation, then we find that some requirements (parts) imply certain properties to hold of the hardware on which the software to be developed is to "run", and, obviously, that remaining — probably the larger parts of the — requirements imply certain properties to hold of that software. So we find that although we may believe that our job is software engineering, important parts of our job are to also "design the machine"!

We shall keep this in mind, and later treat the above implications in Part VI: "Computing Systems Design".

17.2 Why Requirements, and for What?

Some questions now come to mind:

Why do we wish to express requirements? On what basis do we express requirements? How are requirements expressed? How do we gather requirements? From whom do we gather requirements? How might we know whether we have the right requirements? How are we sure that what we have expressed as requirements are feasible, i.e., implementable desiderata?

These and other questions will be answered in this chapter.

17.2.1 Why Requirements?

Before we can design the software for the hardware — that we also have to "design" (i.e., configure) — we must know what that software + hardware, i.e., the machine, shall do. Expression of that 'what' is that which we call the requirements. We take as a dogma, i.e., as a metarequirement, or as a requirement to software development itself, that we must somehow understand these requirements reasonably well, before we start the software design itself.

17.2.2 Requirements for What?

So, summarising, requirements express properties, some parts of which are to be implemented by hardware, and some parts of which are to be implemented by software, such that the 'whole' implements all of the requirements. That is, requirements express properties of entities, functions, and behaviours that one wishes a (or the) machine to exhibit — and events that the machine needs to handle.

17.2.3 What Does 'Implements' Mean?

What do we mean when we say that a computing systems design, $\mathcal{S}$, implements the requirements, $\mathcal{R}$? It shall mean that one can argue — can reason, can prove, can check, and can test — that under assumptions, $\mathcal{D}$, about the

domain, the design $\mathcal{S}$ has the functions, entities and behaviours expressed in the requirements $\mathcal{R}$.

We can express this mathematically:

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

where we read $\models$ as "models".

17.3 Getting Started on Requirements Development

Let us "reset" our thinking about requirements. Somehow we have to get started. Example 17.1 showed just an incomplete glimpse. So what do we do? How do we get started?

17.3.1 Initial Informative Documentation

We first refer to Chap. 2's coverage of informative documents. We refer to the informative "Current Situation", "Needs and Ideas", "Concepts and Facilities", "Scope and Span", and "Design Brief" document parts. According to our "dogma" (on documentation, especially informative documentation) we must somehow gather our thoughts — we being the requirements development possible partners and stakeholders — around these topics.

We must find out what in the current situation somehow generates, in the minds of some of the stakeholders, some needs and ideas concerning computing. We must also find out which computing concepts and facilities they lead onto, and what scope and span these needs, ideas, concepts, and facilities thereby help set. Finally, we need to determine what design brief may then transpire from all this.

Example 17.3 *Informative Requirements Document: Document System Domain:* We continue our line of examples: Examples 17.1 and 17.2, focusing now on the document system domain.

- **Current situation:** The context is that of public administration. The *current situation*, as perceived, is that there is almost no control as to (i) where the manifest, i.e., the paper, documents are, in other words, their current location; (ii) which are originals, which are copies and which are edited versions of originals or copies; and (iii) which persons created, edited and/or copied individual documents, i.e., are responsible for these documents and possibly their distribution (confidentiality, whereabouts).
- **Needs and ideas:** There is therefore perceived a *need* for bringing order into this domain. The *idea* is to do so by gradually switching to a paperless, fully electronic document regime.

- **Concepts and facilities:** More specifically each document produced, copied and/or edited, is thought to be electronic, to be provided with reference to location, time and (the) person(s) involved in the creation, editing, copying and possible "destruction" (shredding or deletion) of (electronic) documents.
- **Scope and Span:** Thus the *scope* is that of a public administration's entire document handling, while the *span* focuses on the computerised support of document creation, distribution (hence copying), editing, destruction and tracing.
- **Design brief:** Based on an existing domain description for the, or a, *document system domain,* there is to be developed a *requirements prescription* for the computerisation of parts of that domain, and as follows:
  - ⋆ The desired (i.e., required) machine is to support the coexistence of manifest paper, i.e., old, documents and electronic, i.e., new, documents.
  - ⋆ No electronic document shall ever by copied onto paper.
  - ⋆ Old paper documents may be scanned into electronic form, and only if all such copies and edited versions from the same original are so scanned and thus moved to the electronic document system.
  - ⋆ Otherwise the electronic document system shall support the creation of original documents, the editing and copying of documents — resulting in documents (edited versions, respectively copies of "prior" documents).
  - ⋆ The sum total of all documents shall have each document traceable "back", via all intermediary documents (edited versions and/or copies), to respective originals.
  - ⋆ Each document, and each stage in any trace, shall record the location, the time and the person(s) involved in the creation, the editing, or the copying, whichever is relevant.

  You are, based on the detailed domain description, and in collaboration with relevant stakeholders, to acquire requirements, to analyse these, to develop a requirements prescription, to verify, where needed, to validate, and to evaluate satisfiability and feasibility of the requirements prescription.

▪

With the "informative bits and pieces" being settled, a first beginning has been made. The developers' minds have been focused. It is time for the developer, possibly before requirements acquisition, to try sketch a first draft requirements prescription.

17.3.2 Requirements Eurekas

But how do the ideas, concepts and facilities that are recorded in the informative documentation of a requirements development form an initial albeit very rudimentary set of requirements, how do these ideas, first arise? We shall refer to these "arisals" as eurekas, as, *Oh, I've seen it!* We now discuss the arisal of these eurekas.

Initial Eureka of Requirements

The idea, concepts and facilities part of the informative documents are the first places in the documentation of the requirements phase in which specific requirements appear. How did they get there?

Well think of a situation in which there is nothing "there"! (There may not even be a domain description!) Now play out the following alternative scenarios. A client, that is, a potential user of computing, has a problem (the "current situation") and deduces from that some "needs", and hence comes up with the "idea", perhaps even some "concepts and facilities" — all aimed at solving the problem. The client decides, as perhaps already implied in the ideas, concepts and facilities, to contact a software house. Or a developer, a software house, before approaching a potential client, discerns that some such clients are in a current situation involving some needs, ideas, concepts and facilities that all lead up to and entail requirements for software. The software house contacts the, or some such client. We could call those scenarios for the initial sources of the eureka.

Ongoing Eureka of Requirements

A client and a software house engages in a dialogue whose purpose it is to "come up with" requirements. How is that dialogue to be managed and organised, that is, monitored and controlled? Either there is a plan or there is no plan — and we assume that both parties are interested in there being a good plan. Either a plan makes logical sense or it does not really make logical sense— and we assume that both parties are interested in an objective plan. The aim of this part of this volume is to provide such a logical, objective plan for requirements engineering.

A Systematic Source of Requirements Eurekas

The pivotal axis around which our logical and objective plan for requirements engineering evolves is the existence of a domain description. If an appropriate domain description does not exist, then we assume that sufficient parts of a domain description are developed together with, that is, at the same time as the requirements prescription is developed. So the domain description is to be the "standard" source of requirements "eurekas"! Literally speaking the

client and the developers, that is, the requirements engineers, read through the domain description. For every described phenomenon or concept, whether an entity, a function, an event or a behaviour, a number of questions are asked. Is this domain phenomenon or concept part of the requirements also? (If yes then it is projected onto the requirements.) If so, is the selected domain phenomenon or concept too nondeterminate? Must the machine reflect the projected phenomenon or concept less nondeterministically? For any projected phenomenon or concept is it too generically described and must it be more specific, that is, instantiated? And so on. The above questioning and answering process thus takes a domain description and turns it increasingly into a (domain) requirement prescription.

Placement of Initial Requirements Eurekas

So the process of developing requirements starts with some initial eurekas. The very first ones are recorded in the informative documentation as part of the ideas, concepts and facilities. A first more comprehensive presentation of these and, perhaps a few more, are then recorded in the informative documentation's synopsis part. Finally the bulk of requirements, including repeating the initial requirements eurekas, usually in more clarified and refined form, are to be placed in the second part of the documentation of the results of the requirements development phase — the requirements prescription part — usually first as a reasonably comprehensive rough sketch, followed by a very much more systematic presentation. The rest of this part, this chapter and Chaps. 18–25, deals with this "systematics".

17.3.3 Pragmatic Prescriptive Documentation

We now refer to Chap. 2's coverage of descriptive, here prescriptive, documents. We, in particular, refer to the concept of rough sketches. A first, good step of development of a requirements prescription, based on the design brief, and possibly based also on first attempts of requirements acquisition, is to write a reasonably extensive rough-sketch requirements prescription. We shall refer to such a document as a requirements pragmatics.

Example 17.4 *A Requirements Pragmatics: The Administrative Forms Handling Domain:* We choose this time another of our example cases: that of administrative document handling. A rough sketch — which assumes some domain description of administrative forms handling — may be as follows: the documents of our administrative forms handling system are of three kinds: **T**emplates of forms and aggregations, **F**orms, i.e., partially or fully filled-in template forms, and **A**ggregations, i.e., partially or fully computed aggregation templates. We refer to these as **TFA**.

TFA shall support the following functions: the design of uniquely identified form templates and their handling in a reservoir of commonly, or selectively

available form templates; the design of uniquely identified aggregation templates, and their handling in a reservoir of commonly, or selectively available aggregation templates; the filling in of form templates (to create uniquely identified forms); the aggregating of forms and aggregations, according to some aggregation template (to create uniquely identified aggregations); and the distribution of templates, forms and aggregations. Form filling is usually a human action. Aggregation is usually a computerised function.

A form template has a unique form identifier, and usually prescribes named and typed template fields. Some template fields are atomic, i.e., consist of no template subfields, and other template fields are like form templates, i.e., are composite.

An aggregation template has a unique aggregation identifier, and usually prescribes from which number of forms, identified by their form template identifiers, and from which number of aggregations, identified by their aggregation template identifiers, the aggregation is to be computed. The aggregation template then prescribes which, more specific ("spreadsheet"-like) computation rules are to be involved in the aggregation.

And so forth! ∎

With a pragmatics (i.e., rough sketch) of what might evolve into a reasonable and proper requirements narrative, a beginning has been made. The developers' minds have been focused. Planning can begin.

17.3.4 Planning Requirements Development

Once you know what it takes to construct a full requirements documentation, that is, once you have been through all the stages and steps, you will be in a reasonable position to also plan requirements development for future projects. The purpose of the examples of this section (i.e., Examples 17.1–17.4) has been to make a number of claims, i.e., a number of "dogmas", plausible. The next section will now overview the stages of requirements development.

17.4 On Domains, Requirements and the Machine

One way of looking at the process of developing software from requirements based on domain models is informally illustrated in Fig. 17.1. There is the given domain, shown as a curved corner box in order to indicate that the domain is not sharply delineated and cannot be fully formalised. The domain engineer (DE) creates a domain model (DM) from an understanding of the domain. Based on the domain model the requirements engineer (RE) transforms the domain model into and creates the requirements model (RM). Based on the requirements model the software designer (SD) transforms the requirements model into and creates software (S).

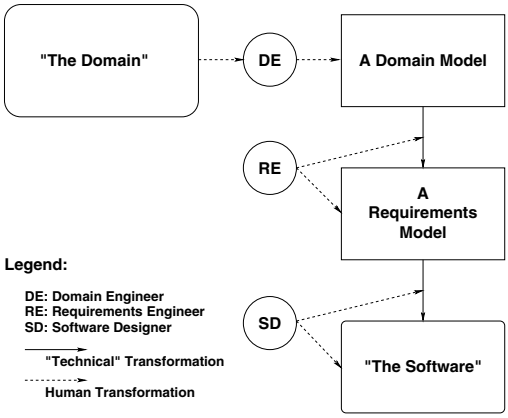


Fig. 17.1. A picture of a development process

Another way of looking at the process of developing software from requirements based on domain models is informally illustrated in Fig. 17.2. Assume that we have a domain (D) and a domain model (DM). We may then say (or claim) that the domain model is a model of the domain (among many possible). Assume similarly that we have some software (S) and a requirements model (RM). We may then say (or claim) that the requirements model is a model of the software (among many possible). In the first case (D, DM) we may visualise the situation as someone, i.e., the DM, standing where DM is placed on Fig. 17.2, and looking at the D.

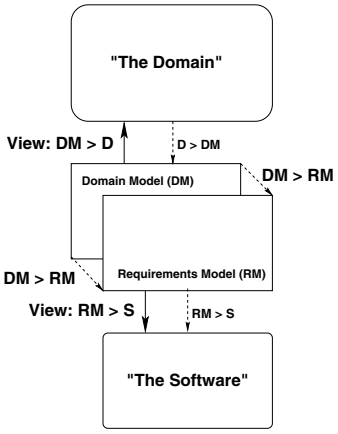


Fig. 17.2. Another picture of the development process

In the second case (RM, S) we may visualise the situation as someone, i.e., the RM, standing where RM is placed on Fig. 17.2, and looking at the S.

Now the requirements model, as we shall see in this part of the book, is more or less derived from the domain model. That is: The two models, the domain model and the requirements model have very many things in common, but one is a model of some actual world ("out there"), whereas the other is (to be) a model of some virtual world ("in there") effected by the software.

Now to some sort of conclusion of this gedankenexperiment. The domain description models some domain. The requirements prescription models some software. The transformation from a domain description to a requirements prescription is really one of turning around 180^{o}: From considering properties of a, or the domain to considering properties of the desired software, yet the two models are very similar! Keep this in mind: although one is "massaging" domain descriptions into requirements, one is really focusing on the software — not how it performs, but what properties any performance of the software ought have.

17.5 Overview: Requirements Engineering Stages

Requirements engineering starts with *stakeholder identification*, which is covered in Chap. 18. Requirements engineering then goes on with *requirements acquisition* in Chap. 20. Then we move on to *requirements analysis and concept formation* in Chap. 21. Once a set of consistent and a set of relatively complete requirements have been gathered and analysed, proper *requirements facet modelling* can take place. Requirements facet modelling is a major undertaking and its result forms a main result of requirements engineering. This is covered in Chap. 19. During requirements modelling we may usually find that *requirements verification* may be needed, Chap. 22, Sect. 22.2. At the end of requirements modelling we shall perform a *requirements validation*, which serves to make sure that the requirements development phase has achieved the right requirements, covered in Chap. 22, Sect. 22.3. A final stage of *requirements satisfiability and feasibility* is needed to complete a full and proper requirements development, see Chap. 23. Some of the satisfiability and feasibility study may be performed "in line" with the acquisition and/or analysis of requirements (Chaps. 20 or 21), or "in line" with the modelling of requirements (Chap. 19).

Please observe that we first present the domain (facet) modelling principles, techniques and tools in Chap. 19 before presenting the ("prior") domain acquisition (Chap. 20) and domain analysis and concept formation techniques (Chap. 21). The reason is simple: we, i.e., you, the practicing requirements engineer, must be thoroughly familiar with *"what kinds of 'things' go into the requirement model"* (documents) before we ask stakeholders.

17.6 The Requirements Document

We say: the requirements document. But we may as well mean the set of requirements documents.

17.6.1 A Preview of Things to Come

The aim of requirements engineering is to create informative, descriptive and analytic documents about and constituting the requirements. Therefore it is important to always keep in mind what a possible contents listing could be of such a complete set of documents. We shall therefore outline, in "capsule" form, what a possible, and, to us, desirable structure could be of such a set of requirements documents. The aim of Part V is then to present the principles, techniques and tools for creating, i.e., developing, such sets of requirements documents.

17.6.2 Contents of a Requirements Document

We bring in, so far without comments, a schematic, "sample" contents listing of a possible, complete requirements documentation.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas (Eurekas, I)
   (e) Concepts and Facilities (Eurekas, II)
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis (Eurekas, III)
   (j) Standards Compliance
   (k) Contracts, with Design Brief
   (l) The Teams
      i. Management
      ii. Developers
      iii. Client Staff
      iv. Consultants
2. Prescriptions
   (a) Stakeholders
   (b) The Acquisition Process
      i. Studies
      ii. Interviews
      iii. Questionnaires
      iv. Indexed Description Units
   (c) Rough Sketches (Eurekas, IV)
   (d) Terminology
   (e) Facets:
      i. BPR
         - Sanctity of Intrinsics
         - Support Technology
         - Management and Organisation
         - Rules and Regulations
         - Human Behaviour
         - Scripting
      ii. Domain Requirements
         - Projection
         - Determination
         - Instantiation
         - Extension
         - Fitting
      iii. Interface Requirements
         - Shared Phenomena and Concept Identification
         - Shared Data Initialisation
         - Shared Data Refreshment
         - Man-Machine Dialogue
         - Physiological Interface

- Machine-Machine Dialogue

iv. Machine Requirements
- Performance
  - ⋆ Storage
  - ⋆ Time
  - ⋆ Software Size
- Dependability
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Reliability
  - ⋆ Robustness
  - ⋆ Safety
  - ⋆ Security
- Maintenance
  - ⋆ Adaptive
  - ⋆ Corrective
  - ⋆ Perfective
  - ⋆ Preventive
- Platform (P)
  - ⋆ Development P
  - ⋆ Demonstration P
  - ⋆ Execution P
  - ⋆ Maintenance P
- Documentation Requirements
- Other Requirements

v. Full Requirements Facets Documentation

3. Analyses
   (a) Requirements Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompletenesses
      iv. Resolutions
   (b) Requirements Validation
      i. Stakeholder Walkthroughs
      ii. Resolutions
   (c) Requirements Verification
      i. Theorem Proofs
      ii. Model Checks
      iii. Test Cases and Tests
   (d) Requirements Theory
   (e) Satisfiability and Feasibility
      i. Satisfaction: correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
      ii. Feasibility: technical, economic, BPR

17.6.3 Comments on Requirements Documents

The requirements document contents listing is but an example. Other forms could be thought of. We shall comment on those later, in Sect. 24.5.

17.7 The Structure of the Rest of the Part

In the next chapters, we do not present the principles and techniques for carrying out the requirements engineering stages and steps in the same order as their preferred approach. In order to do requirements acquisition we must first know, we claim, what makes up a properly structured and "contented" requirements description.

So we treat the four "cornerstones" of a requirements model first (Chap. 19). Then we treat requirements acquisition (Chap. 20), followed by requirements analysis and concept formation (Chap. 21). Finally, we treat requirements validation (Chap. 22), and ideas on studying the satisfiability and feasibility of requirements (Chap. 23). We start with discussing the concept of requirements stakeholders (Chap. 18).

17.8 Bibliographical Notes

As for the entire phase of domain engineering, our approach to requirements engineering possesses some rather novel features. That is, we bring in new principles and techniques into requirements engineering: These methodological concepts are not covered elsewhere in today's available literature on requirements engineering [121, 275, 284, 338, 369].

17.9 Exercises

17.9.1 A Preamble

The exercises of this chapter are a bit "loose" in that not much detailed substance about requirements engineering has been said so far, in this chapter on requirements engineering. And thus we really cannot ask for detailed, objective answers. Most of the exercises below are slightly edited repeats of exercises of Sect. 8.12. From there the term domain has basically been replaced by the term requirements. And, since many of the basic issues of the questions of the present chapter, i.e., the questions below, are similar to those of domain engineering, you are therefore asked to venture your guesses as answers.

17.9.2 The Exercises

The exercises of this chapter are all closed book exercises.

Exercise 17.1 *Why Requirements Engineering?* Without consulting the chapter text, try to recapitulate, in a few lines of informal text, how this chapter motivates 'Why Requirements Engineering?'.

Exercise 17.2 *Machine.* Without consulting the chapter text, characterise what is meant by 'the machine'.

Exercise 17.3 *A Main Objective of Requirements Development.* Without consulting the chapter text, express very briefly what a main objective of requirements development might be.

Exercise 17.4 *Stages of Requirements Engineering.* Without consulting the chapter text, try to recapitulate, in some six or so lines of informal text, the ordered stages of requirements engineering.

Exercise 17.5 *Requirements Acquisition.* Without consulting the chapter text, try to characterise, in a few lines, how this chapter defines requirements acquisition.

Exercise 17.6 *Requirements Validation.* Without consulting the chapter text, try to characterise, in a few lines, how this chapter defines requirements validation.

Exercise 17.7 *Requirements Analysis.* Without consulting the chapter text, try to characterise, in a few lines, how this chapter defines requirements analysis.

Exercise 17.8 *Requirements Documentation.* Without consulting the chapter text, try to list, as exhaustively and in as structured a fashion as possible, a possible, generic domain requirements table of contents listing.

18

Requirements Stakeholders

- The **prerequisite** for studying this chapter is that you have read Chaps. 1 and 17 of this volume.
- The **aims** are to introduce the concept of (requirements) stakeholders, and to distinguish between between different categories of stakeholders.
- The **objective** is to ensure that you carefully consider and include the concerns of all relevant stakeholders when developing requirements.
- The **treatment** is informal and systematic.

We refer to Chap. 9 for a treatment of domain stakeholders.

18.1 Introduction

In our treatment of domain engineering we first defined, analysed and made use of the stakeholder concept (cf. Chap. 9). We shall now review this stakeholder concept, "tuning" it, as it were, to the requirements engineering phase.

The emphasis in the current chapter will be on the possibly changed role, and hence perspectives, of stakeholders. We shall focus on, i.e., analyse, just two such sets of ("extreme") stakeholder groups: At one extreme, there are those stakeholders who represent "real-life", non-informatics, non-IT,[1] application domains. We say that the requirements are *customer driven*. And at the other "extreme" there are those stakeholders who represent such software houses which focus on developing, marketing and selling *COTS*, commercial off-the-shelf software. We say that the requirements are *market driven*. Based on this analysis the reader should be able to perform similar analyses on the

[1] By "noninformatics, non-IT" applications we mean those that are themselves not aimed at IT, but at such things as administration (government and private), at airport, at air traffic, at financial service, at freight logistics, at harbour, at healthcare, at hotels, at manufacturing, at markets (retail and wholesale), at restaurants, at tourism, at transportation, etc., applications of computing systems.

perspectives of stakeholder groups differing from those treated here. That is: Typically lying somewhere in the spectrum between the above-listed — and below-treated — two "extremes".

18.2 General Application Stakeholders

The perspectives of general application stakeholders — when confronted with the task of expressing requirements — are, classically, the following: They are primarily interested in just and only their own instantiation of the domain. The software developer cum vendor may try persuade them to fit some (i.e., a not insignificant number of) requirements to software packages (subsystems, systems) that the vendor (developer) has already had experience with. Maybe that would work, but the general domain application stakeholder, i.e., the client — the customer — must carefully analyse the risks that this might pose: Is the client really getting what is wanted?

Other factors that determine perspectives are: Will the requirements that "I", as a general application (requirements) stakeholder, express lead to too costly software, will deployment of the software really lead to the desired strategic gains foreseen (better economics, improved competitiveness, improved working conditions, etc.), or will deployment of the software require such substantial business process reengineering that it will seriously "upset" the company, one way or the other? We shall have opportunity to review these issues in a later chapter on feasibility.

The remainder of Part V will primarily deal with requirements development principles, techniques and tools relevant to this category of clients. That is, those who desire "turnkey" development of requirements for their "own", special, custom-made software.

18.3 COTS Software House Stakeholders

But before we delve into the details of requirements development principles, techniques and tools for general application software, let us examine the other "extreme" of software development: That where software houses develop software for "shrink-wrapped", commercial off-the-shelf (COTS) software. To which stakeholders do they turn when acquiring requirements?

18.3.1 General

COTS software from any one specific software house usually represents a line of related products. The product relationship represents a set of related segments of the domain being served by that software. The COTS software house stakeholders span from the owners, via executive, strategic, tactical and operational management to the software engineers and their supporting

technicians. We refer to Sect. 9.2.2 for a discussion of software development domain stakeholders.

18.3.2 "Corporate Knowledge"

The main perspectives of the COTS software house stakeholders builds on very strong in-house knowledge of the application domain. This is typically represented by having developed and researched a reasonably widely spanning and thorough, i.e., deep, domain model.

18.3.3 Classes of Domain-Specific Requirements

The main perspectives of the COTS software house stakeholders are, accordingly, while "themselves" developing COTS software requirements, those of securing sets of complementing, i.e., "neighbouring", requirements. Each set covers a well-defined client- and user-oriented segment of the application domain, with members of any given set of requirements being related to one another in some tree-structured manner, and such that the member sets cover related segments of the domain.

18.3.4 Generic COTS Software Stakeholder Perspective

We assume that a software house is marketing, developing, selling, instantiating and servicing a number of software products for a particular domain. The perspective that the marketing stakeholders of that software house may have is to secure a logically coherent set of products such that its customers, its clients, can start with some products and can "grow" into related, "additional feature" software products and onto related, "orthogonal or entirely 'different' feature" software products. The perspective that the software development stakeholders of that software house may have is to ensure that pairs of "additional feature" software products represent extensive reuse, and to ensure that pairs of "orthogonal or entirely new feature" software products interface smoothly to other such software products. Otherwise the perspectives of marketing, development, etc., stakeholders are additionally those of the application domain owner, client and user stakeholders, and that the COTS software can be instantiated in either of the following ways: either by the customer, or by other software houses, consulting firms, or the original COTS software house.

18.4 Discussion

18.4.1 General

In the remainder of this part we shall mostly assume *customer-driven* requirements acquisition. Not much is known about *market-driven* requirements acquisition.

18.4.2 Principles, Techniques and Tools

We summarise the principles, techniques and tools shared with domain stakeholder development (Sect. 9.4.2):

Principle. *Requirements Stakeholder:* At the very outset of a development project identify all possible and potential requirements stakeholders. It is better to include too many than to forget some who can later create a nuisance, or more, when rightfully intervening. Be prepared, throughout a project, to revise the list of requirements stakeholders. ∎

Principle. *Requirements Stakeholder Perspective:* At the very outset of a development project define, together with designated requirements stakeholders, their role, their "jurisdiction", their "rights and duties". Be prepared, throughout a project, to revise the roles of stakeholders. ∎

Techniques. *Requirements Stakeholder Liaison:* (i) Maintain openly inspectable lists of all contemplated, respectively of all actual, requirements stakeholders. (ii) Liaise regularly with all actual requirements stakeholders. (iii) Inform all other (contemplated) requirements stakeholders of "what's going on". (iv) Write down in clear (natural, yet legally binding) language the role of each actual stakeholder. (v) Maintain a dossier of all communications with all requirements stakeholders. Typically such communications deal, as we shall see later, with: Role assignments, acquisition, and validation. ∎

Tools. *Requirements Stakeholder Liaison:* The tools mentioned under information documents, see Sect. 2.4.10, apply equally well here. ∎

18.5 Exercises

18.5.1 Preamble

The first 4 exercises (18.1–18.4) of this chapter are *closed book* exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions. Exercises similar to the below were posed in Sect. 9.5.2.

18.5.2 The Exercises

Exercise 18.1 *Requirements Stakeholder.* This is a repeat question (see Exercise 8.8): Without consulting chapter texts in these volumes, try to characterise, in a few lines, how this chapter defines the concept of a requirements stakeholder.

Exercise 18.2 *Requirements Stakeholder Perspective.* This is a repeat question (see Exercise 8.9): Without consulting chapter texts in these volumes, try to characterise, in a few lines, how this chapter defines the concept of requirements stakeholder perspective. (Does it?)

Exercise 18.3 *General Application Versus Software Development Requirements Stakeholders.* Without consulting chapter texts in these volumes, try to enumerate, in three or so lines, how this chapter perceives of a spectrum from general application stakeholders to software development stakeholders. Do you see any contrast to Chap. 9's view on this?

Exercise 18.4 *General Application Versus Software Development Requirements Stakeholder Perspectives.* Given your answer to the previous exercise (Exercise 18.3 above), augment the answer by providing for each entry in your enumerated list with a brief, 3–5 line characterisation of corresponding perspectives.

Exercise 18.5 *Domain-Specific Requirements Stakeholders.* For the fixed topic, selected by you, present as exhaustive a list of stakeholders as you think is relevant for your requirements.

Exercise 18.6 *Domain-Specific Requirements Stakeholder Perspectives.* Given your answer to the previous exercise (Exercise 18.5 above), augment the answer by providing for each entry in your enumerated list a brief, 3–5 line characterisation of corresponding requirements-specific perspectives.

19

Requirements Facets

- The **prerequisite** for studying this chapter is that you, as a requirements engineer, need to know: *what are the constituents of a proper model of requirements?*
- The **aims** are to introduce the concept that a proper requirements prescription is made up from most of the following constituent prescriptions, i.e., facets: (i) domain, (ii) interface and (iii) machine requirements, and, within each of these three groups of facets, of (i) projections, determinations, instantiations, extensions and fittings, respectively of (ii) shared data intialisation and refreshment, computational data and control, man-machine dialogues, man-machine physiological, and machine-machine dialogues, and of (iii) performance, dependability, maintenance, platform, and documentation requirements respectively; and to present principles, techniques and tools for the prescription of these facets.
- The **objective** is to ensure that you will become a thoroughly professional requirements engineer.
- The **treatment** is from systematic to formal.

Throughout requirements engineering remember to adhere to:

The "Golden Rule" of Requirements Engineering

Prescribe only those requirements that can be objectively shown to hold for the designed software.

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

An "Ideal Rule" of Requirements Engineering

When prescribing (incl. formalising) requirements, also formulate tests (theorems, properties for model checking) whose actualisation should show adherence to the requirements.

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

19.1 Introduction

As is the case with Chap. 11, "Domain Facets", this chapter constitutes a second "high point" of the present volume. It is in this chapter that we present principles and techniques of requirements engineering which are not, today, otherwise available in any other textbook on software engineering. So take your time to become thoroughly familiar with the contents of the present chapter.

The chapter is structured as follows: First we rough-sketch, with little or no consideration of the carefully worked out domain descriptions, an initial set of (eureka) requirements — such as they may emerge from a more or less undigested requirements acquisitions process (Sect. 19.2). On the basis of the rough (eureka requirements) sketch we create a requirements terminology and install the first terms into that terminology. Then we decompose the further requirements development into the four major facets, which are then covered in the next sections: "Business Process Reengineering" (Sect. 19.3), "Domain Requirements" (Sect. 19.4), "Interface Requirements" (Sect. 19.5) and "Machine Requirements" (Sect. 19.6). As an ongoing effort, during the requirements facets development stages, we use and maintain, that is, revise and install, additional terms into the terminology.

19.2 Rough Sketching and Terminology

The aim of this section is to remind the reader that in order to come up with a proper model of requirements we must first have performed proper identification of and requirements acquisition from stakeholders. After such a requirements acquisition stage we can analyse the acquired requirements prescription units. And after such an analysis we are ready to rough sketch, i.e., to make a first attempt at constructing, some requirements document, while, at the same time, establishing a terminology document. In this section we shall overview these two aspects of "Requirements Engineering".

19.2.1 Initial Requirements Modelling

In Example 17.1 we illustrated examples of "one", or "two", or "three liner" requirements description units. Once, as a result of requirements acquisition

(Chap. 20), you have gathered what you may think of as a sufficient number of such analysed requirements description units, you are ready to rough sketch a requirements prescription.

19.2.2 Rough-Sketch Requirements

A rough-sketch requirements prescription is (thus) based on a number of partially "digested", i.e., partially analysed and conceptualised, requirements description units. The requirements engineer is encouraged to try to formulate a reasonably complete and consistent rough-sketch requirements prescription, in order to do a more thorough requirements analysis and concept formation.

The requirements description units, in a sense, only express the stakeholders' views on requirements. These units may reflect a somewhat incoherent "total view". After a reasonably proper requirements analysis and concept formation stage, the requirements engineer (i.e., the analyst), is able to formulate a more coherent total view. Rough-sketching these requirements thus affords a first opportunity for the requirements engineer to express the requirements.

Example 19.1 *A Rough-Sketch Container Terminal Domain:* To illustrate a rough-sketch requirements we need first be able to refer to a domain description. In this case we present a rough-sketch domain description.

Entities

We itemize list entities of container harbours, in no particular order, only as they come to mind:

- **Container terminal:** A container terminal is a composite entity. It consists of a harbour basin of water, of one or more quays, of one or more container pools, and of zero, one or more container freight stations. The harbour water basin connects on one side to the open sea, and on the other side to one or more quays. Attributes of a container terminal are: its name, its maritime location (latitude and longitude), its number of quays, number of pools, etc.
- **Quay:** A quay is a composite entity. A quay is like a straight road: The quays connect on one side to the harbour basin, and on the other side, possibly via a container terminal internal road net, to one or more container pools, and, possibly via these, to the possible container freight stations. The quay also consists of one or more cranes. Quays have attributes: length, width, number of cranes, position within the container terminal, possibly a name, etc.
- **Container:** A container is a composite entity. It consists of (i) the container box (which has length [say 20 or 40 feet], height, width, owner, etc., attributes), (ii) its contents (which may be empty, and which we choose to abstract from, i.e., to not consider (in other words: disregard)), and (iii)

its bill of lading. The latter has attributes such as: contents listing, which agent (i.e., merchant) is sending this container, which agent (merchant) is to receive the container, from where, via where, and to where, etc.

- **Bill of Lading (BoL):** A document which evidences a contract of carriage by sea. The document has the following functions:
  1. A receipt for goods, signed by a duly authorised person on behalf of the carriers.
  2. A document of title to the goods described therein.
  3. Evidence of the terms and conditions of carriage agreed upon between the two parties.

  At the moment three different models are used:
  1. A document for either combined transport or port-to-port shipments, depending on whether the relevant spaces for place of receipt and/or place of delivery are indicated on the face of the document.
  2. A classic marine BoL in which the carrier is also responsible for the part of the transport actually performed by himself.
  3. Sea waybill: A nonnegotiable document, which can only be made out to a named consignee. No surrender of the document by the consignee is required.
- **Container ship:** A container ship is a composite entity. It consists of one or more locations which can each hold, or which actually hold a container. So the container ship also consists of these containers. Container locations are called cells, and cells are laid out in bays, rows and tiers (like an x, y, z coordinate system). Thus containers are stacked. The container ship is further so arranged as to have these columns (i.e., stacks) of containers be accessible from the top, through what is called a hatchway, an opening, that can be covered by what is called a hatch cover. This hatch cover is removed when unloading and loading containers to the appropriate stacks that it covers. Ship attributes have to do with the exact arrangement of bays, rows and tiers, and thus as to how many containers the ship can, and at any moment, actually carry. Ships can berth at a quay. They then occupy a certain length of that quay.
- **Ship/quay crane:** A crane, either aboard the ship, or positioned at quayside, can lift (unload) containers out of hatchways and onto (a truck on) the quay, or, the other way around (load containers). Cranes have attributes: operating area (along a quay), possibly a unique name (identifier), carrying (lifting) weight, handling speed (capacity), etc. For any ship there is a maximum number of such cranes that can serve the ship at any one time.
- **Container truck:** A truck is a composite entity. It consists of a chassis and usually zero or one container. The chassis may be considered either composite or atomic. Whatever is chosen, the chassis enables the container truck to move. Container trucks have attributes: carrying (load) capacity, service speed, etc.

- **(Un)Loading plan:** A load plan for a container ship is a document which specifies the sequences of stacking and unstacking of containers, as that container ship calls on a succession of container harbours. Since containers can only be removed from or added onto the top of the cell position stacks on a container ship, the order in which these stacks are loaded and unloaded determines is crucial. No container is ever to be temporarily unloaded in order to get at containers "below" it — whereupon, once these containers have been unloaded, the temporarily unloaded containers are again loaded. No *Towers of Hanoi* puzzles here!
- **Pool:** A pool is a composite entity. It consists of one or more areas where a stack of containers may be, as well as the containers actually positioned there. Some pools can receive and (can thus) handle refrigerated containers (reefers). Stacks within a pool are usually ordered by row and column. Pools have attributes: location (name and position) within the container terminal, capacity (number and height of stacks), whether reefer or ordinary containers, etc.
- **Pool crane:** A pool crane, like a quay/ship crane, can move containers, one at a time, between container trucks/chassis and pool stacks.

Functions

- **Calling:** A container ship contacts, i.e., calls, a container terminal to advise it of its intended arrival, giving its call sign. The 'calling may, or may not imply a request for permission to go to a previously scheduled quay position.
- **Unloading movement:** This is a simple function and could be regarded as an atomic function. Often it is called a movement. The function concerns the unloading of a single container from a cell position aboard the container ship by a designated crane onto a container truck or a container chassis.
- **Loading movement:** See the above, since it is basically the reverse movement.

These two movements reflect the fact that container truck and container chassis can only move one container at a time.

- **Chassis/truck movement:** We also consider this a simple, atomic function: Moving, by motor driven vehicle, one container from a crane at a quay to a crane at a pool, or vice versa.
- **Hatch cover removal (opening):** An atomic function which opens up for the hatchway so that containers can be loaded or unloaded.
- **Hatch cover replacement (closure):** An atomic function which closes the hatchway.

Events

We rough-sketch some possible events:

- The arrival of a container vessel to a quay position
- The departure of a container vessel from a quay position
- The failure to remove (to open) a hatch cover
- The failure to replace (to close) a hatch cover
- The failure of a crane to grip a container
- The failure of a crane to release a container
- The failure of a container truck/chassis to move
- The failure of a container vessel to move
- The outbreak of an epidemic disease

Behaviours

- **A ship visit:** A normal, "uneventful" ship visit behaviour starts with the ship calling (action) and proceeds to the arrival of the container vessel at a quay position (event). Some hatch covers may be opened. It then continues with one or more concurrent sequences of container unloadings and loadings (actions). It ends (possibly) with the closing of hatch covers (actions) and the departure of the container vessel from the quay position.
- **A merchant freight truck visit:** A freight truck usually carries just one (say a 40-foot) container, or, in cases, two (20-foot) containers. A merchant freight truck is a freight truck which carries one merchant's container(s), overland, to or from a container terminal. Its visit is for three purposes: to deliver one or two containers, to fetch one or two containers, or both. Its behaviour wrt. the container terminal is: arrival (an event) at the container terminal, registration (a function) at the container terminal gate (statement of purpose, showing of papers (waybills, bill of loadings), etc.), unloading and/or loading of containers (either at a special area, called the container yard (or, in certain cases, at the container freight station), or directly to a pool area, or, even, directly on the quay, for immediate ship loading or unloading).
- **A 24-hour crane behaviour:** We encourage the reader to try complete this item as an exercise (Exercise 19.15).
- **A 24-hour container truck/chassis behaviour:** We encourage the reader to try complete this item as an exercise (Exercise 19.16).

Please note that Exercises 19.15–19.16 asks you to both consider describing actual domain behaviours and prescribing desirable requirements. ▪

Now, on the background of the above rough domain sketch, we are ready to express a rough requirements sketch.

Example 19.2 *A Rough-Sketch Requirements for Container Stowage:* After some discussion with stakeholders we arrive at the following base requirements for a *ship and pool areas container loading plan* computing system. (What we

here name ship and pool area container loading plans are, more colloquially, called stowage plans.)

1. **Container:** Every *container* c (that is to be involved in the planning of loading plans, and hence subject to actual loading and unloading) *shall possess* the following *attributes:* (i) length and (ii) BoL, b.
2. **Bill of Lading:** The BoL states the route which the container, c, is to take, or is taking or has taken. It is a requirement that the system *shall* establish and maintain BoLs for all relevant containers.
3. [**Ship sailing**] **route:** A route is here considered a sequence of two or more container terminal visits. A container terminal visit is a pair: the name of a container terminal (T) and the name of a container ship (s) or, for the last in such a sequence, say **nil.** The ship S takes the container C from container terminal t. Let $r : < (t_1, s_1), \ldots, (t_i, s_i), (t_{i+1}, s_{i+1}), \ldots, (t_n, \textbf{nil}) >$ designate a route for some container. It expresses that that container is transported from container terminal t_i to container terminal t_{i+1} by container ship s_i. It is a requirement that the system *shall* establish and maintain ship sailing routes, for all of a ship owner's relevant container ships.
4. **Ship container stack layout ('context'):** For every relevant container ship (say, in the ship owner's fleet of such), full **information shall be maintained** of how each ship is laid out wrt. container stacks (this is called contextual information).
5. **Ship container stack 'state':** For every container ship being considered, we further require that *a state shall be maintained.* The state is information about the location of all current containers: where, aboard, i.e., in which stack and cell position, they are stored. A well-formedness about this state expresses that each container has a BoL which states that it should indeed be aboard that ship at the moment the state is recorded.
6. **Pool area container stack layouts ('context'):** For every relevant container terminal, and for every container pool area (that is relevant to the ship owner for which these requirements are to be developed, and within these container terminals), *information* about the topological layout and pool area stacks, whether for ordinary containers, or for reefers, whether for 20-foot or for 40-foot (etc.) containers, *shall be kept and regularly updated* to reflect any changes in pool area layouts, etc.
7. **Pool area container stack 'states':** For every pool area container stack being (thus) considered, we further require that *a state shall be maintained.* The state is information about all current containers being stored in that pool area stack and their location, that is, BoLs and where (i.e., bay, row, cell position), etc.
8. **Shipping orders:** There *shall* at any moment *be a latest set of shipping orders.* By shipping orders we understand a current set of outstanding orders for the shipping of containers.

(a) **Pragmatics: Outstanding (container shipping) order.** By an outstanding (container shipping) order we mean an order for a container transport, i.e., an order whose transport is being requested, but for which no acknowledgement of its precise shipping has yet been given.
(b) **Syntax: Outstanding (container shipping) order.** The order document specifies (i.e., restates) the BoL of the container and a sequence of one or more container terminals.
(c) **Semantics: Outstanding (container shipping) order.** The meaning of an outstanding (container shipping) order is, if it is accepted, that it enters the allocation and scheduling process of the relevant shipowner(s), and thus, eventually, is confirmed.

9. **Confirmed (container) shipping order:** By a confirmed (container) shipping order we mean a shipping order which is no longer outstanding: Its syntax has been understood, and its semantics has been implemented. That is, it has been used in the construction of one or more ship container loading plans (and possibly also in one or more container pool area loading plans). Whether the container in question is actually en route is here left open.
10. **Ship container loading plans:** Based on the above forms of information, i.e., items 1–9, the required computing system *shall generate two kinds* of reasonably optimal ship container loading plans (i.e., *documents*): static and dynamic.
11. **Static ship container loading plan:** A static ship container loading plan is a plan that prescribes which containers are loading and unloading at which container terminals, for a given ship, i.e., for a given route that this ship is to follow, and for a given set of outstanding shipping orders. The plan also states where each container is to be located aboard the ship.
12. **Dynamic ship container loading plan:** Given a static ship container loading plan, and given a container terminal (i.e., the name of a terminal at which the ship for that loading plan is berthed), the dynamic ship container loading plan specifies the sequences in which containers are to be unloaded and loaded.
    - As an example of the issues involved in loading and unloading, let us consider the following:
      - ⋆ Let container c_i be loaded on stack s in terminal t_i.
      - ⋆ Let container c_{i+1} be loaded on stack s in terminal t_i or t_{i+1} (i.e., immediately "on top of" c_i).
      - ⋆ Now container c_{i+1} can be unloaded from stack s in terminal t_{i+2}.
      - ⋆ Container c_i can be unloaded from stack s in terminal t_{i+2}, or some suitable later terminal.
    - That is, a stack push and pop discipline must be adhered to.
13. **[Reasonably] optimal static ship container loading plan:** A static loading plan is said to be [reasonably] optimal if no other such plan can

be found which "fills" all stacks of a ship to their (almost) fullest capacity while adhering to the stacking discipline.

14. [**Reasonably**] **optimal dynamic ship container loading plan:** A dynamic loading plan is said to be [reasonably] optimal if no other such plan can be found which generates the longest sequences of ship crane container movements with respect to the same ship stack.
15. **The generation of plans:** The intent of any dynamic ship container loading plan is that actual unloadings and loadings *shall* be commensurate with, i.e., "follow", that plan.
16. **Container pool area loading plan:** And so on; this plan will not be prescribed.
17. **Container ship loadings and unloadings:** By container ship loadings and unloadings we understand the sequences of ship crane positions, along the quay, next to, i.e., servicing, a given ship, as well as the movement, for each ship crane position, of containers to and from the ship (i.e., from and to the quay). Since translocating a ship crane (from one quay/ship position to another) takes time we wish to minimise the number of ship crane translocations.

Lest you have lost sight of what the rough-sketch requirements really were, we here summarise these:

2. Initialisation and refreshment of container BoLs
3. Initialisation and refreshment of ship sailing routes
4. Initialisation and refreshment of ship container stack layouts
5. Initialisation and refreshment of ship container stack states
6. Initialisation and refreshment of pool area container stack layouts
7. Initialisation and refreshment of pool area container stack states
8. Storage and reference to shipping orders, includes securing item 9
11. Generation of static ship container loading plan, securing item 13
12. Generation of dynamic ship container loading plan, securing item 14
16. Generation of container pool area loading plan (prescription omitted)
17. Minimise ship crane translations, securing item 15

We remind the reader that the above constitutes a set of rough-sketched requirements and that we likewise presented only rough-sketched descriptions of some aspects of the domain of container terminals in Example 19.1. ▪

So the above gave you some kind of rough-sketch example of what requirements may entail. The example was not that small. It had to be "semi-large". You have to see, with your "own eyes", that rough sketches are not small. In fact, they are much larger than the above example.

Before we proceed to the main material of this chapter on requirements facets, let us take a brief look at the interaction between rough-sketching and terminologisation.

19.2.3 Requirements Terminology

We briefly covered, in Chap. 2, the topic of terminology. We do this to put that topic in a more proper context, that is, to hint at the size and complexity, of a realistic terminology.

Example 19.3 *An Incomplete Container Terminal Terminology:*
This terminology section is (i) far from complete, (ii) and much too long. And it only covers the domain, not the requirements. We bring in a rather extensive extract so that the reader can see what it takes to construct a terminology. Namely that it takes quite a lot. The sheer size of the example, albeit just a minor part of the real list, should indicate to the reader the seriousnees with which we press the issue of constructing realistic terminologies. The terms are culled from the Internet [276] (a list of terms from the P&O Nedlloyd shipping company). We stress that we have copied freely from [276] and that we encourage the reader, as Exercise 19.1, to rephrase and formalise part of this terminology.

1. *Actual voyage number:* A code for identification purposes of the voyage and vessel which actually transports the container/cargo.
2. *Agency fee:* Fee payable by a ship owner or ship operator to a port agent.
3. *Agent:*
   (a) A person or organization authorised to act for or on behalf of another person or organization.
   (b) In P&O Nedlloyd, an Agent is a corporate body with which there is an agreement to perform particular functions on behalf of them for an agreed payment. An Agent is either a part of the P&O Nedlloyd organization or an independent body. The following functions and responsibilities may apply to the activities of an agent.
      i. *Sales:* Marketing, acquisition of cargo, issuing quotations, concluding contracts in coordination with P&O Nedlloyd. Basically the agent is the first point of entry into the P&O Nedlloyd organization for a shipper.
      ii. *Bookings:* Booking of cargo in accordance with allotments assigned to the agent for a certain voyage by P&O Nedlloyd.
      iii. *Customs:* Dealing with the national customs administration for cargo declarations, manifest alterations and cargo clearance on behalf of P&O Nedlloyd.
      iv. *Documentation:* Responsible for timeliness and correctness of all documentation required, regarding the carriage of cargo.
      v. *Handling:* Taking care of all procedures connected with physical handling of cargo.
      vi. *Equipment control:* Managing of all equipment stock in a particular area.

vii. *Issuing:* Authorised to sign and issue Bills of Lading and other transport documents.
viii. *Collecting:* Authorised to collect freight and charges on behalf of P&O Nedlloyd.
ix. *Delivery:* The agent who releases the cargo and is responsible for its delivery to the consignee.
x. *Handling of cargo claims:* Handling of cargo claims as per agency contract.
xi. *Husbanding:* Handling non-cargo-related operations of a vessel as instructed by the master, owner or charterer.

4. *Area code:* A code for the area where a container is situated.
5. *Area off hire lease:* Geographical area where a leased container becomes off hire.
6. *Area off hire sublease:* Geographical area where a subleased container becomes off hire.
7. *Area on hire lease:* Geographical area where a leased container becomes on hire.
8. *Area on hire sublease:* Geographical area where a subleased container becomes on hire.
9. *Arrival date:* The date on which goods or a means of transport is due to arrive at the delivery site of the transport.
10. *Arrival notice:* A notice sent by a carrier to a nominated notify party advising of the arrival of a certain shipment or consignment.
11. *Auto container:* Container equipped for the transportation of vehicles.
12. *Automated guided vehicle system:* Unmanned vehicles equipped with automatic guidance equipment which follow a prescribed path, stopping at each necessary station for automatic or manual loading or unloading.
13. *Automatic identification:* A means of identifying an item, e.g., a product, parcel or transport unit, by a machine (device) entering the data automatically into a computer. The most widely used technology at present is barcode; others include radio frequency, magnetic strips and optical character recognition.
14. *BoL:* See Bill of Lading.
15. *Barcoding:* A method of encoding data for fast and accurate electronic readability. Barcodes are a series of alternating bars and spaces printed or stamped on products, labels, or other media, representing encoded information which can be read by electronic readers, used to facilitate timely and accurate input of data to a computer system. Barcodes represent letters and/or numbers and special characters like +, /, −, etc.
16. *Barge:* Flat-bottomed inland cargo vessel for canals and rivers with or without own propulsion for the purpose of transporting goods.
17. *Bay:* A vertical division of a vessel from stem to stern, used as a part of the indication of a stowage place for containers. The numbers run from stem

to stern; odd numbers indicate a 20-foot position, even numbers indicate a 40-foot position.

18. *Bay plan:* A stowage plan which shows the locations of all the containers on the vessel.
19. *Berth:* A location in a port where a vessel can be moored, often indicated by a code or name.
20. *Bill of Lading:* Abbreviation: BoL. A document which evidences a contract of carriage by sea. The document has the following functions:
    (a) A receipt for goods, signed by a duly authorised person on behalf of the carriers.
    (b) A document of title to the goods described therein.
    (c) Evidence of the terms and conditions of carriage agreed upon between the two parties.

    At the moment 3 different models are used:
    (d) A document for either Combined Transport or Port-to-Port shipments depending on whether the relevant spaces for place of receipt and/or place of delivery are indicated on the face of the document.
    (e) A classic marine Bill of Lading in which the carrier is also responsible for the part of the transport actually performed by himself.
    (f) Sea Waybill: A non-negotiable document, which can only be made out to a named consignee. No surrender of the document by the consignee is required.
21. *Bill of Lading clause:* A particular article, stipulation or single proviso in a Bill of Lading. A clause can be standard and can be preprinted on the BoL.
22. *Bill of Material:* A list of all parts, subassemblies and raw materials that constitute a particular assembly, showing the quantity of each required item.
23. *Boat:* A small open-decked craft carried aboard ships for a specific purpose, e.g., lifeboat, workboat.
24. *Bonded:* The storage of certain goods under charge of customs viz. customs seal until the import duties are paid or until the goods are taken out of the country.
    (a) Bonded warehouse (place where goods can be placed under bond).
    (b) Bonded store (place on a vessel where goods are placed behind seal until the time that the vessel leaves the port or country again).
    (c) Bonded goods (dutiable goods upon which duties have not been paid, i.e., goods in transit or warehoused pending customs clearance).
25. *Box:* Colloquial name for container (e.g., Box-club).
26. *Bulk container:* A container designed for the carriage of free-flowing dry cargo, loaded through hatchways in the roof of the container and discharged through hatchways at one end of the container.
27. *Business process:* A business process is the action taken to respond to particular events, convert inputs into outputs, and produce particular re-

sults. Business processes are what the enterprise must do to conduct its business successfully.

28. *Business process model:* The business process model provides a breakdown (process decomposition) of all levels of business processes within the scope of a business area. It also shows process dynamics, lower-level process interrelationships. In summary it includes all diagrams related to a process definition, allowing for understanding what the business process is doing (and not how).
29. *Business process redesign (BPR):* The process of redesigning business practice models including the exchange of data and services amongst the stakeholders (i.e., finance, merchandising, production, distribution) involved in the life cycle of a client's product.
30. *Call:* The visit of a vessel to a port.
31. *Call sign:* A code published by the International Telecommunication Union in its annual List of Ships' Stations to be used for the information interchange between vessels, port authorities and other relevant participants in international trade. *Note:* The code structure is based on a three-digit designation series assigned by the ITU and one digit assigned by the country of registration. (PDHP = P&O Nedlloyd Rotterdam)
32. *Cargo:*
    (a) Goods transported or to be transported, all goods carried on a ship covered by a BoL.
    (b) Any goods, wares, merchandise, and articles of every kind whatsoever carried on a ship, other than mail, ship's stores, ship's spare parts, ship's equipment, stowage material, crew's effects and passengers' accompanied baggage.
    (c) Any property carried on an aircraft, other than mail, stores and accompanied or mishandled baggage. Also referred to as 'goods'.
33. *Carrier:* The party undertaking transport of goods from one point to another.
34. *Cell:* Location aboard a container vessel where one container can be stowed.
35. *Cell position:* The location of a cell aboard of a container vessel identified by a code for, successively, the bay, the row and the tier, indicating the position of a container on that vessel.
36. *Cellular vessel:* A vessel, specially designed and equipped for the carriage of containers.
37. *Consignee:* The party such as mentioned in the transport document by whom the goods, cargo or containers are to be received.
38. *Consignment:* A separate identifiable number of goods (available to be) transported from one consignor to one consignee via one or more than one modes of transport and specified in one single transport document.
39. *Consignment instructions:* Instructions from either the seller/consignor or the buyer/consignee to a freight forwarder, carrier or his agent, or other

provider of a service, enabling the movement of goods and associated activities. The following functions can be covered:

- Movement and handling of goods (shipping, forwarding and stowage).
- Customs formalities.
- Distribution of documents.
- Allocation of documents (freight and charges for the connected operations).
- Special instructions (insurance, dangerous goods, goods release, additional documents required).

40. *Container:* An item of equipment as defined by the International Organization for Standardization (ISO) for transport purposes. It must be:
    (a) a permanent character and accordingly strong enough to be suitable for repeated use;
    (b) specially designed to facilitate the carriage of goods, by one or more modes of transport, without intermediate reloading;
    (c) fitted with devices permitting its ready handling, particularly from one mode of transport to another;
    (d) so designed as to be easy to fill and empty;
    (e) having an internal volume of one cubic meter or more.
41. *Container chassis:* A vehicle specially built for the purpose of transporting a container so that, when container and chassis are assembled, the produced unit serves as a road trailer.
42. *CFS: Container freight station:* A facility at which (export) LCL (less than container load) cargo is received from merchants for loading (stuffing) into containers or at which (import) LCL cargo is unloaded (stripped) from containers and delivered to merchants.
43. *CLP: Container load plan:* A list of items loaded in a specific container and where appropriate, their sequence of loading.
44. *Container logistics:* The controlling and positioning of containers and other equipment.
45. *Container manifest:* The document specifying the contents of particular freight containers or other transport units, prepared by the party responsible for their loading into the container or unit.
46. *Container moves:* The number of actions performed by one container crane during a certain period.
47. *Container pool:* A certain stock of containers which is jointly used by several container carriers and/or leasing companies.
48. *Container ship:* A vessel, i.e., a floating structure designed for the transport of containers.
49. *Container stack:* Two or more containers, one placed above the other, forming a vertical column.
50. *Container terminal:* Place where loaded and/or empty containers are loaded or discharged into or from a means of transport.

51. *Container yard:* Abbreviation: CY. A facility at which FCL traffic and empty containers are received from or delivered to the Merchant by or on behalf of the Carrier.
52. *Fully cellular container ship:* Abbreviation: FCC. A vessel specially designed to carry containers, with cell-guides under deck and necessary fittings and equipment on deck.
53. *Full container load:* Abbreviation: FCL.
    (a) A container stuffed or stripped under risk and for account of the shipper and/or the consignee.
    (b) A general reference for identifying container loads of cargo loaded and/or discharged at merchants' premises.
54. *Grid number:* An indication of the position of a container in a bay plan by means of a combination of page number, column and line. The page number often represents the bay number.
55. *Hatch cover:* Watertight means of closing the hatchway of a vessel.
56. *Hatch way:* Opening in the deck of a vessel through which cargo is loaded into, or discharged from the hold and which is closed by means of a hatch cover.
57. *LCL:* Less than container load.
58. *Merchant:* For cargo carried under the terms and conditions of the Carrier's Bill of Lading and of a tariff, it means any trader or persons (e.g., Shipper, Consignee) and including anyone acting on their behalf, owning or entitled to possession of the goods.
59. *Reefer container:* A thermal container with refrigerating appliances (mechanical compressor unit, absorption unit, etc.) to control the temperature of cargo.
60. Etcetera!

We again refer to [276] for full details. ▪

The "moral" of the above three examples is the composite of: a real domain description is long; a real requirements prescription is long; and a real terminology is long. In a textbook we can only hint at, but not illustrate, the real size of our descriptions, prescriptions and specifications.

19.2.4 Systematic Narration

From the rough sketches of requirements to a properly expressed, consistent, relatively complete and well-structured requirements document, there is still a long way to go in order to cover all relevant aspects, here called facets, of the requirements. It is the purpose of the next sections to overview proper structures, proper principles and proper prescription techniques, for attaining such well-designed requirements documents.

19.3 Business Process Reengineering Requirements

We remind the reader of Section 11.2.1.

Characterisation. By *business process reengineering* we understand the reformulation of previously adopted business process descriptions, together with additional business process engineering work. ▪

Business process reengineering (BPR) is about *change*, and hence BPR is also about *change management*. The concept of workflow is one of these "hyped" as well as "hijacked" terms: They sound good, and they make you "feel" good. But they are often applied to widely different subjects, albeit having some phenomena in common. By workflow we shall, very loosely, understand the physical movement of people, materials, information and "centre ('locus') of control" in some organisation (be it a factory, a hospital or other). We have, in Vol. 1, Chap. 12 (Petri Nets), in Sect. 12.5.1 covered the notion of *work flow systems*.

19.3.1 Michael Hammer's Ideas on BPR

Michael Hammer, a guru of the business process reengineering "movement", states [140]:

1. *Understand a method of reengineering before you do it for serious.*

So this is what this chapter is all about!

2. *One can only reengineer processes.*
3. *Understanding the process is an essential first step in reengineering.*

And then he goes on to say: *"but an analysis of those processes is a waste of time. You must place strict limits, both on time you take to develop this understanding and on the length of the description you make."* Needless to say we question this latter part of the third item.

4. *If you proceed to reengineer without the proper leadership, you are making a fatal mistake. If your leadership is nominal rather than serious, and isn't prepared to make the required commitment, your efforts are doomed to failure.*

By leadership is basically meant: "upper, executive management".

5. *Reengineering requires radical, breakthrough ideas about process design. Reengineering leaders must encourage people to pursue stretch goals*[1] *and to think out of the box; to this end, leadership must reward creative thinking and be willing to consider any new idea.*

[1] A 'stretch goal' is a goal, an objective, for which, if one wishes to achieve that goal, one has to stretch oneself.

This is clearly an example of the US guru, "new management"-type 'speak'!

6. *Before implementing a process in the real world create a laboratory version in order to test whether your ideas work. . . . Proceeding directly from idea to real-world implementation is (usually) a recipe for disaster.*

Our careful both informal and formal description of the existing domain processes, as covered in Chap. 11, as well as the similarly careful prescription of the reengineered business processes shall, in a sense, make up for this otherwise vague term "laboratory version".

7. *You must reengineer quickly. If you can't show some tangible results within a year, you will lose the support and momentum necessary to make the effort successful. To this end "scope creep" must be avoided at all cost. Stay focused and narrow the scope if necessary in order to get results fast.*

We obviously do not agree, in principle and in general, with this statement.

8. *You cannot reengineer a process in isolation. Everything must be on the table. Any attempts to set limits, to preserve a piece of the old system, will doom your efforts to failure.*

We can only agree. But the wording is like mantras. As a software engineer, founded in science, such statements as the above are not technical, are not scientific. They are "management speak".

9. *Reengineering needs its own style of implementation: fast, improvisational, and iterative.*

We are not so sure about this statement either! Professional engineering work is something one neither does fast nor improvisational.

10. *Any successful reengineering effort must take into account the personal needs of the individuals it will affect. The new process must offer some benefit to the people who are, after all, being asked to embrace enormous change, and the transition from the old process to the new one must be made with great sensitivity as to their feelings.*

This is nothing but a politically correct, pat statement! It would not pass the negation test: Nobody would claim the opposite. Real benefits of reengineering often come from not requiring as many people, i.e., workers and management, in the corporation as before reengineering. Hence: What about the "feelings" of those laid off?

19.3.2 What Are *BPR Requirements?*

Two "paths" lead to business process reengineering:

- A client wishes to improve enterprise operations by deploying new computing systems (i.e., new software). In the course of formulating requirements for this new computing system a need arises to also reengineer the human operations within and without the enterprise.

- An enterprise wishes to improve operations by redesigning the way staff operates within the enterprise and the way in which customers and staff operate across the enterprise-to-environment interface. In the course of formulating reengineering directives a need arises to also deploy new software, for which requirements therefore have to be enunciated.

One way or the other, business process reengineering is an integral component in deploying new computing systems.

19.3.3 Overview of BPR Operations

We suggest six domain-to-business process reengineering operations:

1. introduction of some new and removal of some old *intrinsics;*
2. introduction of some new and removal of some old *support technologies;*
3. introduction of some new and removal of some old *management and organisation substructures;*
4. introduction of some new and removal of some old *rules and regulations;*
5. introduction of some new and removal of some old work practices (relating to *human behaviours*); and
6. related *scripting.*

19.3.4 BPR and the Requirements Document

Requirements for New Business Processes

The reader must be duly "warned": The BPR requirements are not for a computing system, but for the people who "surround" that (future) system. The BPR requirements state, unequivocally, how those people are to act, i.e., to use that system properly. Any implications, by the BPR requirements, as to concepts and facilities of the new computing system must be prescribed (also) in the domain and interface requirements.

Place in Narrative Document

We shall thus, in Sects. 19.3.5–19.3.10, treat a number of BPR facets. Each of whatever you decide to focus on, in any one requirements development, must be prescribed. And the prescription must be put into the overall requirements prescription document.

As the BPR requirements "rebuilds" the business process description part of the domain description[2], and as the BPR requirements are not directly requirements for the machine, we find that they (the BPR requirements texts) can be simply put in a separate section.

[2] — Even if that business process description part of the domain description is "empty" or nearly so!

There are basically two ways of "rebuilding" the domain description's business process's description part (D_{BP}) into the requirements prescription part's BPR requirements (R_{BPR}). Either you keep all of D as a base part in R_{BPR}, and then you follow that part (i.e., R_{BPR}) with statements, R'_{BPR}, that express the new business process's "differences" with respect to the "old" (D_{BP}). Call the result R_{BPR}. Or you simply rewrite (in a sense, the whole of) D_{BP} directly into R_{BPR}, copying all of D_{BP}, and editing wherever necessary.

Place in Formalisation Document

The above statements as how to express the "merging" of BPR requirements into the overall requirements document apply to the narrative as well as to the formalised prescriptions.

Formal Presentation: Documentation

We may assume that there is a formal domain description, $\mathcal{D}_{BP}$, (of business processes) from which we develop the formal prescription of the BPR requirements. We may then decide to either develop entirely new descriptions of the new business processes, i.e., actually prescriptions for the business reengineered processes, $\mathcal{R}_{BPR}$; or develop, from $\mathcal{D}_{BP}$, using a suitable schema calculus, such as the one in RSL, the requirements prescription $\mathcal{R}_{BPR}$, by suitable parameterisation, extension, hiding, etc., of the domain description $\mathcal{D}_{BP}$.

19.3.5 Intrinsics Review and Replacement

Characterisation. By *intrinsics review and replacement* we understand an evaluation as to whether current intrinsics stays or goes, and as to whether newer intrinsics need to be introduced. ▪

Example 19.4 *Intrinsics Replacement:* A railway net owner changes its business from owning, operating and maintaining railway nets (lines, stations and signals) to operating trains. Hence the more detailed state changing notions of rail units need no longer be part of that new company's intrinsics while the notions of trains and passengers need be introduced as relevant intrinsics. ▪

Replacement of intrinsics usually point to dramatic changes of the business and are usually not done in connection with subsequent and related software requirements development.

19.3.6 Support Technology Review and Replacement

Characterisation. By *support technology review and replacement* we understand an evaluation as to whether current support technology as used in

the enterprise is adequate, and as to whether other (newer) support technology can better perform the desired services. ∎

Example 19.5 *Support Technology Review and Replacement:* Currently the main information flow of an enterprise is taken care of by printed paper, copying machines and physical distribution. All such documents, whether originals (masters), copies, or annotated versions of originals or copies, are subject to confidentiality. As part of a computerised system for handling the future information flow, it is specified, by some domain requirements, that document confidentiality is to be taken care of by encryption, public and private keys, and digital signatures. However, it is realised that there can be a need for taking physical, not just electronic, copies of documents. The following business process reengineering proposal is therefore considered: Specially made printing paper and printing and copying machines are to be procured, and so are printers and copiers whose use requires the insertion of special signature cards which, when used, check that the person printing or copying is the person identified on the card, and that that person may print the desired document. All copiers will refuse to copy such copied documents — hence the special paper. Such paper copies can thus be read at, but not carried outside the premises (of the printers and copiers). And such printers and copiers can register who printed, respectively who tried to copy, which documents. Thus people are now responsible for the security (whereabouts) of possible paper copies (not the required computing system). The above, somewhat construed example, shows the "division of labour" between the contemplated (required, desired) computing system (the "machine") and the "business reengineered" persons authorised to print and possess confidential documents.

It is implied in the above that the reengineered handling of documents would not be feasible without proper computing support. Thus there is a "spill-off" from the business reengineered world to the world of computing systems requirements. ∎

19.3.7 Management and Organisation Reengineering

Characterisation. By *management and organisation reengineering* we understand an evaluation as to whether current management principles and organisation structures as used in the enterprise are adequate, and as to whether other management principles and organisation structures can better monitor and control the enterprise. ∎

Example 19.6 *Management and Organisation Reengineering:* A rather complete computerisation of the procurement practices of a company is being contemplated. Previously procurement was manifested in the following physically separate as well as designwise differently formatted paper documents: *requisition form, order form, purchase order, delivery inspection form, rejection*

and return form, and *payment form*. The supplier had corresponding forms: *order acceptance and quotation form, delivery form, return acceptance form, invoice form, return verification form*, and *payment acceptance form*. The current concern is only the procurement forms, not the supplier forms. The proposed domain requirements are mandating that all procurer forms disappear in their paper version, that basically only one, the procurement document, represents all phases of procurement, and that order, rejection and return notification slips, and payment authorisation notes, be effected by electronically communicated and duly digitally signed messages that represent appropriate subparts of the one, now electronic procurement document. The business process reengineering part may now "short-circuit" previous staff's review and acceptance/rejection of former forms, in favour of fewer staff interventions.

The new business procedures, in this case, subsequently find their way into proper domain requirements: those that support, that is monitor and control all stages of the reengineered procurement process. ▪

19.3.8 Rules and Regulations Reengineering

Characterisation. By *rules and regulations reengineering* we understand an evaluation as to whether current rules and regulations as used in the enterprise are adequate, and as to whether other rules and regulations can better guide and regulate the enterprise. ▪

Here it should be remembered that rules and regulations principally stipulate business engineering processes. That is, they are — i.e., were — usually not computerised.

Example 19.7 *Rules and Regulations Reengineering:* Our example continues that of Example 11.19. We kindly remind the reader to restudy that example. Assume now, due to reengineered support technologies, that interlock signalling can be made magnitudes safer than before, without interlocking. Thence it makes sense to reengineer the rule of Example 11.19 from: *In any three-minute interval at most one train may either arrive to or depart from a railway station* into: *In any 20-second interval at most two trains may either arrive to or depart from a railway station.*

This reengineered rule is subsequently made into a domain requirements, namely that the software system for interlocking is bound by that rule. ▪

19.3.9 Human Behaviour Reengineering

Characterisation. *Human Behaviour Reengineering:* By *human behaviour reengineering* we understand an evaluation as to whether current human behaviour as experienced in the enterprise is acceptable, and as to whether partially changed human behaviours are more suitable for the enterprise. ▪

Example 19.8 *Human Behaviour Reengineering:* A company has experienced certain lax attitudes among members of a certain category of staff. The progress of certain work procedures therefore is reengineered, implying that members of another category of staff are henceforth expected to follow up on the progress of "that" work.

In a subsequent domain requirements stage the above reengineering leads to a number of requirements for computerised monitoring of the two groups of staff. ∎

19.3.10 Script Reengineering

On one hand, there is the engineering of the contents of rules and regulations, and, on another hand, there are the people (management, staff) who script these rules and regulations, and the way in which these rules and regulations are communicated to managers and staff concerned.

Characterisation. By *script reengineering* we understand evaluation as to whether the way in which rules and regulations are scripted and made known (i.e., posted) to stakeholders in and of the enterprise is adequate, and as to whether other ways of scripting and posting are more suitable for the enterprise. ∎

Example 19.9 *Script Reengineering:* We refer to Examples 11.22–11.25. They illustrated the description of a perceived bank script language. One that was used, for example, to explain to bank clients how demand/deposit and mortgage accounts, and hence loans, "worked".

With the given set of "schematised" and "user-friendly" script commands, such as they were identified in the referenced examples, only some banking transactions can be described. Some obvious ones cannot, for example, *merge two mortgage accounts, transfer money between accounts in two different banks, pay monthly and quarterly credit card bills, send and receive funds from stockbrokers,* etc.

A reengineering is therefore called for, one that is really first to be done in the basic business processes of a bank offering these services to its customers. We leave the rest as an exercise, cf. Exercise 19.13. ∎

19.3.11 Discussion: Business Process Reengineering

Who Should Do the Business Process Reengineering?

It is not in our power, as software engineers, to make the kind of business process reengineering decisions implied above. Rather it is, perhaps, more the prerogative of appropriately educated, trained and skilled (i.e., gifted) other kinds of engineers or business people to make the kinds of decisions implied

above. Once the BP reengineering has been made, it then behooves the client stakeholders to further decide whether the BP reengineering shall imply some requirements, or not.

Once that last decision has been made in the affirmative, we, as software engineers, can then apply our abstraction and modelling skills, and, while collaborating with the former kinds of professionals, make the appropriate prescriptions for the BPR requirements. These will typically be in the form of domain requirements, which are covered extensively in Sect. 19.4.

General

Business process reengineering is based on the premise that corporations must change their way of operating, and, hence, must "reinvent" themselves. Some corporations (enterprises, businesses, etc.) are "vertically" structured along functions, products or geographical regions. This often means that business processes "cut across" vertical units. Others are "horizontally" structured along coherent business processes. This often means that business processes "cut across" functions, products or geographical regions. In either case adjustments may need to be made as the business (i.e., products, sales, markets, etc.) changes. We otherwise refer to currently leading books on business process reengineering: [139, 140, 176, 186].

19.4 Domain Requirements

Characterisation. By *domain requirements* we understand requirements which are expressed solely in terms of domain phenomena and concepts. ∎

So in setting out, initially, acquiring (that is, eliciting or "extracting") requirements, the requirements engineer naturally starts "in" or "with" the domain. That is, the requirements engineer asks questions, of the stakeholders, that eventually should lead to the formulation of domain requirements. The structuring of these questions — it is strongly suggested — should follow the structuring and contents of the domain facets description of the domain model, Sects. 11.3–11.8, and the five kinds of domain-to-requirements operations outlined next and treated in some depth in the following.

19.4.1 Domain-to-Requirements Operations

Characterisation. By a *domain-to-requirements operation* we shall understand a transformation of domain description documents into requirements description documents. ∎

These document transformation operations are carried out by the requirements engineer. They follow as the result of the requirements engineer working closely with possibly alternating groups of stakeholders.

We suggest the following five domain-to-requirements operations covered in depth in five subsections below (Sects. 19.4.4–19.4.8).

1. domain projection
2. domain determination
3. domain instantiation
4. domain extension
5. domain fitting

19.4.2 Domain Requirements and the Requirements Document

Some remarks need to be made before we go into details of domain requirements modelling techniques.

Requirements for Functionalities

Domain requirements are about "operating" part of the domain "inside" the machine. Domain requirements engineering is about which parts to leave out, i.e., which parts to "emulate", and then in what "shape, forms and contents".

Place in Narrative Document

In Sects. 19.4.4–19.4.8 we shall treat a number of domain requirements facets. Each of whichever you decide to focus on, in any one requirements development, must be prescribed.

The domain requirements all take their "departure point", that is, are based upon, the entire domain description. That is, the domain requirements represent a kind of "rewrite" of the domain description. Whether this "rewrite" is done one way, or another way, for that we cannot really state any hard principles. It all depends, so much, on the subject domain and the subject requirements. There are basically two ways of doing the "rebuilding" of the domain description's non-business process description part (D[3]) into the requirements prescription part's domain requirements (R_{DR}), and that is as follows:

Either you keep all of D as a base part (R'_{DR}) in R_{DR}, and then you follow that part (i.e., R'_{DR}) with statements, R''_{DR}, that express the new business process's "differences" with respect to the "old" (D). Call the result R_{DR}. Or you simply rewrite (in a sense, the whole of) D directly into R_{DR}, copying all of D, and editing wherever necessary.

[3] Here D stands for the (i) intrinsics, the (ii) support technology, the (iii) management and organisation, the (iv) rules and regulations, the (v) script, and the (vi) human behaviour parts

Place in Formalisation Document

The above statements as how to express the "rewrite" of requirements into the overall requirements document applies, in particular, to narrative prescriptions. But as we shall see, it also applies to formal prescriptions.

Formal Presentation: Documentation

We may assume that there is a formal domain description, $\mathcal{D}$, from which we develop the formal prescription of the domain requirements. We may then decide to either develop entirely new descriptions of the new "domain", i.e., actually prescriptions for the domain requirements, $\mathcal{R}_{DR}$; or develop, from $\mathcal{D}$, using a suitable schema calculus, such as the one in RSL, the requirements prescription, $\mathcal{R}_{DR}$, by suitable parameterisation, extension, hiding, etc., of the domain description $\mathcal{D}$.

19.4.3 A Domain Example

The bulk of this, the domain requirements section, is "carried" by a number of examples, one each, basically, for each of the domain-to-requirements transformation schemes. To place these transformations in a proper context we first present a rather simple-minded domain description.

Example 19.10 *A Simple Domain Example: A Timetable System:* We choose a very simple domain: that of a traffic timetable, say flight timetable. In the domain you could, in "ye olde days", hold such a timetable in your hand, you could browse it, you could look up a special flight, you could tear pages out of it, etc. There was no end as to what you could do to such a timetable. So we will just postulate a sort, TT, of timetables.

Airline customers, clients, in general, only wish to inquire a timetable (so we will here omit treatment of more or less "malicious" or destructive acts). But you could still count the number of digits "7" in the timetable, and other such ridiculous things. So we postulate a broadest variety of inquiry functions, qu:QU, that apply to timetables, tt:TT, and yield values, val:VAL.

Specifically designated airline staff may, however, in addition to what a client can do, update the timetable. But, recalling human behaviours, all we can ascertain for sure is that update functions, up:UP, apply to timetables and yield two things: another, replacement timetable, tt:TT, and a result, res:RES, such as: *"your update succeeded"*, or *"your update did not succeed"*, etc. In essence this is all we can say for sure about the domain of timetable creations and uses.

We can view the domain of the timetable, clients and staff as a behaviour which nondeterministically alternates (⌈⌉) between the client querying the timetable client_0(tt), and the staff updating the same staff_0(tt).

Formal Presentation: A Timetable Domain

```
scheme TI_TBL_0 =
  class
    type
      TT, VAL, RES
      QU = TT → VAL
      UP = TT → TT × RES
    value
      client_0: TT → VAL, client_0(tt) ≡ let q:QU in q(tt) end
      staff_0: TT → TT × RES, staff_0(tt) ≡ let u:UP in u(tt) end

      tim_tbl_0: TT → Unit
      tim_tbl_0(tt) ≡
          (let v = client_0(tt) in tim_tbl_0(tt) end)
        ⌈⌉ (let (tt′,r) = staff_0(tt) in tim_tbl_0(tt′) end)
  end
```

The timetable function, tim_tbl, is here seen as a never ending process, hence the type **Unit.** It nondeterministically[4] alternates between "serving" the clients and the staff. Either of these two nondeterministically[4] chooses from a possibly very large set of queries, respectively updates. ∎

19.4.4 Domain Projection

Usually the *span* of the requirements is far "narrower" than the *scope* of the domain. That is, the conceived or actually described domain covers phenomena and concepts that will not be of concern when constructing requirements for some particular application. We shall therefore have to explicitly express a "projection".

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

[4] The nondeterminism referred to is internal in the sense that no outside behaviour influences the choice.

In a sense, of course, the document resulting from a domain projection is still a domain description, but — for pragmatic reasons — we shall refer to it as a domain requirements prescription.

A Specific Example

Example 19.11 *Projection of Airline Timetable and Air Space:*

We start out by formulating a *rough-sketch domain description* for the subdomain of airline timetables: There are airports, and one can fly between certain airports. There are airlines, and an airline offers flight services between such airports and at certain times. These services are recorded in an airline timetable. It lists for every flight offered its flight number and flight days, and a list of two or more airport visits: names of airports, and arrival and departure times.

There is the air space. It consists of airports, of air corridors (zero, one or more between pairs of airports), and of controlled areas around airports where the flight of aircraft is specially monitored (and partly controlled) by air traffic control centres.

Formal Presentation: Projection of Airline Timetable and Air Space, I

```
scheme AIR_TT_SPACE =
  extend TI_TBL_0 with
  class
    type
      AS, Airport, Air_Corridor, Controlled_Area, ATC
    ...
  end
```

Now to a *rough-sketch domain projection prescription:* From the above we leave out any description of the air space. That is, we project "away" air corridors, controlled areas and air traffic control centres. We leave the details to the reader.

Formal Presentation: Projection of Airline Timetable and Air Space, II

```
scheme TI_TBL_1 = TI_TBL_0
```

We have taken the liberty, above, in AIR_TT_SPACE, not to model the details of timetables and the air space. ▪

You may rightfully claim that the above example was construed so as to fit the idea of projection. That may be so. But the idea has been demonstrated, has it not?

A General Example

It is typical to have sorts in a domain description. Once these are projected onto the requirements they change from being abstractions of phenomena to being concepts of these. The former are descriptions, informal or formal, of "things out there", in the domain. The latter are prescriptions, informal or formal, of "things in there", in the software to be built! Whereas observer (and functions defined on the basis of observer) functions are just postulated, the projected observer (etc.) functions prescribe functions that must be implemented. To make that distinction clear we may choose to rename these functions.

Example 19.12 *From Domain Sorts to Requirements Sorts, I:* A transport net consists of segments and junctions such that every segment is connected to exactly two distinct junctions and such that to every junction there is connected one or more segments. Thus from a transport net one may observe its segments (e.g., street segments) and junctions (e.g., street intersections). To achieve a proper, consistent and complete net description we will, most likely, have introduced the concepts of segment and junction identifications — and related, via axioms, segments, junction and their identifiers.

Formal Presentation: A Transport Net Domain Description

```
type
  N, S, J, Si, Ji
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
axiom
  ∀ s:S • card obs_Jis(s)=2 ∧
  ∀ n:N, s,s′:S •
    {s,s′} ⊆ obs_Ss(n) ∧ s≠s′ ⇒ obs_Si(s)≠obs_Si(s′) ∧
    s ∈ obs_Ss(n) ⇒
      let {ji,ji′} = obs_Jis(s) in
      ∃ j,j′:J • {j,j′} ⊆ obs_Js(n) ∧ ji=obs_Ji(j) ∧ ji′=obs_Ji(j′) end ∧
  ∀ j:J • card obs_Sis(j)≥1 ∧
  ∀ n:N, j,j′:J •
    {j,j′} ⊆ obs_Js(n) ∧ j≠j′ ⇒ obs_Ji(j)≠obs_Ji(j′) ∧
    j ∈ obs_Js(n) ⇒
      let sis = obs_Sis(j) in
      ∀ si:Si • si ∈ sis ⇒ ∃ s:S • s ∈ obs_Ss(n) ∧ si=obs_Si(s) end
```

We can annotate the above axioms, line by line: (1) Each segment is connected to exactly two distinct junctions. (3) Two segments of a net, if distinct, have distinct segment identifications. (4–6) For every segment of a net one can observe the identifications of two junctions — and these identifications must be those of junctions of the net. (7) Each junction is connected to one or more distinct segments. (9) Two junctions of a net, if distinct, have distinct junction identifications. (10–12) For every junction of a net one can observe the identifications of one or more segments — and these identifications must be those of segments of the net.

The annotation of the formalisation is really part also of the informal narrative description. ▪

Domain projection now considers which entities: sorts and values, axioms relating these, functions: observer functions, etc., events and behaviours are to be represented, somehow, in the required software.

Example 19.13 *From Domain Sorts to Requirements Sorts, II:* We continue Example 19.12. In this example we may decide to project all that is described in Example 19.12. This means that nets, their segments and junctions shall be represented in the required software. This also means that segment and junction identifiers shall be represented in the required software. Whereas the nets, segments and junctions (i.e., their descriptions) were (models of) real phenomena in the domain, the net, segment and junction prescriptions are models of the required software. Observer functions become functions that must now be implemented. As such we may choose to rename them. Axioms are no longer axioms. They become invariants that must hold of any data structure representation of nets, segments and junctions.

Formal Presentation: A Transport Net Domain Requirements Prescription

```
type
  N, S, J, Si, Ji
value
  xtr_Ss: N → S-set
  xtr_Js: N → J-set
  xtr_Si: S → Si
  xtr_Ji: J → Ji
  xtr_Jis: S → Ji-set
  xtr_Sis: J → Si-set

  wf_N: N → Bool
  wf_N(n) ≡
    ∀ s:S•s ∈ xtr_Ss(n)⇒card xtr_Jis(s)=2 ∧
    ∀ s,s′:S •
```

```
    {s,s′}⊆xtr_Ss(n)∧s≠s′ ⇒ xtr_Si(s)≠xtr_Si(s′) ∧
    s ∈ xtr_Ss(n) ⇒
      let {ji,ji′}=xtr_Jis(s) in
      ∃ j,j′:J•{j,j′}⊆xtr_Js(n)∧ji=xtr_Ji(j)∧ji′=xtr_Ji(j′) end ∧
∀ j:J•j ∈ xtr_Js(n)⇒card xtr_Sis(j)≥1 ∧
∀ j,j′:J •
    {j,j′}⊆xtr_Js(n)∧j≠j′ ⇒ xtr_Ji(j)≠xtr_Ji(j′) ∧
    j ∈ xtr_Js(n) ⇒
    let sis=xtr_Sis(j) in
    ∀ si:Si•si ∈ sis⇒∃ s:S•s ∈ xtr_Ss(n)∧si=xtr_Si(s) end
```

At most a mere renaming, you may say. Yes, but the restatement of the projected domain onto the domain requirements means that from *the domain is "such and such"* we have now required *the software shall implement "such and such"*. ∎

The projection of domain observer functions to requirements extraction functions usually are implemented in terms of queries of a relational database. The various attributes of sorts (as above: segment and junction identifiers (and whatever other attributes one might associate with segments and junctions (length, average cost of traversal, state-of-repair, etc.))) then become attributes of relation tuples. We refer to Sect. 28.3.3 for the story on relational databases.

From Concepts to Phenomena

The projection of the domain description of Example 19.12 onto the domain requirements prescription of Example 19.13 reflects a subtlety: We may claim that the segment and junction identifications of Example 19.12 were mere concepts. There may not have been any physically recognisable phenomena amounting to these identifications *other than the — almost "law of nature" — fact that the mere manifestations of two distinct segments and two distinct junctions amount to the unique identifications of all such segments and junctions*. There may thus not be any physically discoverable junction identifiers associated with segments (and segment identifiers associated with junctions). But it is clear that from junctions one can identify connected segments, and from segments one can identify the "end" junctions.

Conceptual segment and junction identifiers of Example 19.13 now become eventually physically discoverable phenomena of the required software. As such the segment and junction identifications of Example 19.13 are models of phenomena.

19.4.5 Domain Determination

Often a domain exhibits *nondeterminism*, that is: A function result or a behaviour can either be such and such or it can be such and such (different from the first such and such), or it can be such and such (different from the first two such and suches!). Or a function result or a behaviour can be *loose* (i.e., loosely described): not all possible outcomes of a function application, or not all possible behaviours of a phenomenon may have been described, or even knowable. Sometimes, for a requirements, the stakeholders may wish to remove such seeming uncertainty — nondeterminism, or looseness — as to some function results or some behaviours.

Characterisation. By *domain determination* we understand an operation that applies to a (projected) domain description, i.e., a requirements prescription, and yields a domain requirements prescription, where the latter has made deterministic, or specific, some function results or some behaviours of the former. ▪

Certainly the result of domain determination represents, not a domain description (any longer), but a requirements prescription. The point of requiring some software is to exactly make certain behaviours, certain function outcomes, determinate — predictable.

Example 19.14 *Determination of Airline Timetable Queries:* To exemplify this rough-sketch domain (to) requirements operation we first present a rough domain description, then the "more deterministic" domain requirements prescription. (i) A rough-sketch timetable-querying domain description is: There is given a further undefined notion of timetables. There is also given a concept of querying a timetable. A timetable query, abstractly speaking, denotes (i.e., stands for) a function from timetables to results. Results are not further defined. (i) A rough-sketch timetable querying domain requirements description is: There are given notions of departure and arrival times, and of airports, and of airline flight numbers.

Formal Presentation: Determination of Airline Timetable Queries, I

```
scheme TI_TBL_2 =
  extend TI_TBL_1 with
    class
      type
        T, An, Fn
    end
```

A timetable consists of a number of air flight journey entries. Each entry has a flight number, and a list of two or more airport visits. an airport visit consists

of three parts: An airport name, and a pair of (gate) arrival and departure times.

Formal Presentation: Determination of Airline Timetable Queries, II

```
scheme TI_TBL_3 =
  extend TI_TBL_2 with
    class
      type
        JR′ = (T × An × T)*
        JR = {| jr:JR′ • len jr ≥ 2 ∧ ... |}
        TT = Fn ~m~> JR
    end
```

We illustrate just one, simple form of airline timetable queries. A simple airline timetable query either just browses all of an airline timetable, or inquires of the journey of a specific flight. The simple browse query thus need not provide specific argument data, whereas the flight journey query needs to provide a flight number. A simple update query inserts a new pairing of a flight number and a journey to the timetable, whereas a delete query need just provide the number of the flight to be deleted.

The result of a query is a value: the specific journey inquired, or the entire timetable browsed. The result of an update is a possible timetable change and either an "OK" response if the update could be made, or a "Not OK" response if the update could not be made: Either the flight number of the journey to be inserted was already present in the timetable, or the flight number of the journey to be deleted was not present in the timetable.

That is, we assume above that simple airline timetable queries only designate simple flights, with one aircraft. For more complex air flights, with stopovers and changes of flights, see Example 19.16.

You may skip the rest of the example, its formalisation, if your reading of these volumes does not include the various formalisations. First, we formalise the syntactic and the semantic types:

Formal Presentation: Determination of Airline Timetable Queries, III

```
scheme TI_TBL_3Q =
  extend TI_TBL_3 with
    class
      type
        Query == mk_brow() | mk_jour(fn:Fn)
        Update == mk_inst(fn:Fn,jr:JR) | mk_delt(fn:Fn)
        VAL = TT
        RES == ok | not_ok
    end
```

Then we define the semantics of the query commands:

Formal Presentation: Determination of Airline Timetable Queries, IV

```
scheme TI_TBL_3U =
  extend TI_TBL_3 with
    class
      value
        M_q: Query → QU
        M_q(qu) ≡
          case qu of
            mk_brow() → λtt:TT•tt,
            mk_jour(fn)
              → λtt:TT • if fn ∈ dom tt
                           then [fn↦tt(fn)] else [] end
    end  end
```

And, finally, we define the semantics of the update commands:

Formal Presentation: Determination of Airline Timetable Queries, V

```
scheme TI_TBL_3U =
  extend TI_TBL_3 with
    class
        M_u: Update → UP
        M_u(up) ≡
          case qu of
            mk_inst(fn,jr) → λtt:TT •
              if fn ∈ dom tt
                then (tt,not_ok) else (tt ∪ [fn↦jr],ok) end,
            mk_delt(fn) → λtt:TT •
              if fn ∈ dom tt
                then (tt \ {fn},ok) else (tt,not_ok) end
    end  end
```

We can "assemble" the above into the timetable function — calling the new function the timetable system, or just the system function. Before we had:

Formal Presentation: Determination of Airline Timetable Queries, VI

```
value
  tim_tbl_0: TT → Unit
  tim_tbl_0(tt) ≡
```

```
        (let v = client_0(tt) in tim_tbl_0(tt) end)
    ‖ (let (tt′,r) = staff_0(tt) in tim_tbl_0(tt′) end)

Now we get:

value
  system: TT → Unit
  system() ≡
        (let q:Query in let v = M_q(q)(tt) in system(tt) end end)
    ‖ (let u:Update in let (r,tt′) = M_u(q)(tt) in system(tt′) end end)

Or, for use in Example 19.32:

  system(tt) ≡ client(tt) ‖ staff(tt)

  client: TT → Unit
  client(tt) ≡
    let q:Query in let v = M_q(q)(tt) in system(tt) end end

  staff: TT → Unit
  staff(tt) ≡
    let u:Update in let (r,tt′) = M_u(q)(tt) in system(tt′) end end
```

■

We remind the reader that the above example can be fully understood by just reading the rough-sketch texts, that is, without reading their formalisations.

19.4.6 Domain Instantiation

Domain descriptions are usually "lifted" to cover several instances of domains: A railway system domain description may cover railways in several — or be claimed to cover them in "all" — countries! The similar situation holds true for a domain description of "the" financial service industry, "the" healthcare sector, etc. Usually software is being requested for specific instances of such application domains: the railway software of a specific region, the banking software for a specific bank, the hospital software for a specific region's healthcare system, and so on.

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

Example 19.15 *Instantiation: Local Region Railway Nets:* The domain description to be (rough-sketch) requirements instantiated is provided by the rough sketch of Example 11.8- We also refer to Fig. 19.1. The constraints are: There are exactly n stations (where n is given). The n stations have the following names: $s_1, s_2, \ldots, s_n$. These stations can be linearly ordered ($< s_1, s_2, \ldots, s_n >$) such that if two stations are connected by a line, as are s_i, s_{i+1} for $i \in \{1..n-1\}$, then they are connected by exactly two lines, $l_{f_{i,i+1}}, l_{f_{i+1,i}}$, one permitting traffic in one direction ($l_{f_{i,i+1}}$ from s_i to s_{i+1}), the other in the other direction ($l_{f_{i+1,i}}$ from s_{i+1} to s_i). Each station has exactly one platform, with tracks on either side. Both tracks can be reached from any line incident upon the station. Any line emanating from the station can be reached from both station tracks. We refer to Exercise 19.2 which asks for a formalisation of the above. ∎

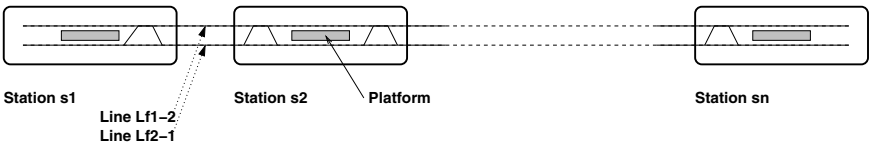


Fig. 19.1. A schematic local region railway net

We leave as an exercise, Exercise 19.2, to formalise Example 19.15.

19.4.7 Domain Extension

We make a distinction between genuine domain extensions and "domain extensions" due to "forgotten" domain facets. The distinction, as are the two kinds of extensions, are pragmatic notions.

Genuine Extensions

Certain phenomena in a domain are conceivable "in theory", but occur rarely in reality — like someone counting to a trillion! But with computing, computers can do your counting! So, although these phenomena, in a sense, "belong" to the domain, they are really only believably feasible when spoken of in connection with computing, hence requirements.

Characterisation. By *domain extension* we understand an operation that applies to a (projected and possibly determined and instantiated) domain description, i.e., a (domain) requirements prescription, and yields a (domain) requirements prescription. The latter prescribes that a software system is to support, partially or fully, an operation that is not only feasible but also computable in reasonable time. ∎

Example 19.16 *Extension: n-Transfer Travel Inquiry:* We assume a projected and instantiated timetable (see Example 19.14).

A query of a timetable may, syntactically, specify an airport of origin, a_o, an airport of destination, a_d, and a maximum number, n, of intermediate stops. The query semantically designates the set of all those trips of one up to n direct air journeys between a_o and a_d, i.e., trips where the passenger may change flights (up to $n-1$ times) at intermediate airports.

Formal Presentation: Extension: n-Transfer Travel Inquiry

```
scheme TI_TBL_3C =
  extend TI_TBL_3 with
    class
      type
        Query′ == Query | mk_conn(fa:An,ta:An,n:Nat)
        VAL′ = VAL | CNS
        CNS = (JR*)-set
      value
        Mq(mk_conn(fa,ta,n)) ≡ ...
    end
```

Here we leave it to the reader to define the "connections" function! At present you need not be concerned with the fact that TI_TBL_3C does not include the timetable initialisation command. To secure that we need to "juggle" some of the previously defined TI_TBL_x schemes. We omit showing this. ∎

The point about this example is that for n being just 4 or above, a hand calculation is infeasible. But a `Prolog` program of less than a dozen lines, when the basis for executions, will start producing results after very few seconds on most PCs, for example for n=5.

"Forgotten" Domain Descriptions

Sometimes one forgets to describe some domain facet. The discovery that one (might) have forgotten such a facet is usually made during domain requirements prescription. A stakeholder requirements is such that the domain requirements engineer lacks a "socket", some text and possibly formulas in the domain description which can serve as a basis for projection, instantiation, determination and extension. An example may serve to focus the idea.

Example 19.17 *A "Forgotten" Transport Net Domain Description:* We continue Examples 19.12–19.13.

We have not equipped segments with attributes (such as lengths, geodetic (cadastral) coordinates, segment state of fitness (i.e., "need of repair"),

or other). And we have therefore not described any functions that observe attributes, attribute values for given attributes, and, for example, those segments of a net which possess attributes (A) of specified values (VAL).

We "discover" this general omission during the requirements gathering stage when stakeholders, for one set of requirements, express the requirement to offer travellers shortest routes in nets, or, for another set of requirements, express the requirement to maintain a high level of fitness of segments.

So we extend the domain description of Example 19.12. With every segment we associate a finite, usually small number of attributes (that is, attribute names, $a : A$). And with every attribute we associate a set of attribute values ($v_1, v_2, \ldots : V$). Thus we are able to observe which attributes are associated with a given segment, and, for that segment and an attribute of that segment, we are able to observe the associated attribute value.

Now we can express the further extensions: Assume ordering relations, $\preceq_{a_i}$, one per attribute $a_i : A$, on attribute values. Now we shall require a function which, from a net, extracts all those segments which for a given attribute have attribute values within a given range.

Formal Presentation: "Extended" Domain Description

```
type
  /* N, S, J, Si, and Ji as in Example 19.12 */
  A, VAL
value
  obs_As: S → A-set
  obs_A_VAL: S × A ~→ VAL
    pre obs_A_VAL(s,a): a ∈ obs_As(s)

  ⪯a: VAL × VAL → Bool

  is_in_range: S × (A × (VAL × VAL)) → Bool
  is_in_range(s,(a,(v,v′))) ≡
    v⪯a obs_A_VAL(s,a)⪯a v′

  extract_Ss: N × (A × (VAL × VAL)) → S-set
  extract_Ss(n,(a,(v,v′))) ≡
    {s|s:S•s ∈ obs_Ss(n)∧a ∈ obs_As(s)∧v⪯a obs_A_VAL(s,a)⪯a v′}
```

The reader can extend the above to also cover junctions. ∎

Once identified, "repairing" the description of a "forgotten" domain facet can either be thought of as a domain extension — and that is why we have placed the issue of "forgetfulness" in this section on domain extension — or it may

prompt the requirements engineer to have the "original" domain description updated.

To keep in line with our treatment of the omission, we decide to handle the "repair" in the extension part of our domain requirements engineering.

Thus we have obviously decided to project the repaired domain facet onto the domain requirements prescription. This first part of the domain extension is then to be followed by possibly further domain to requirements operations.

19.4.8 Domain Requirements Fitting

Often a domain being described "fits" onto, is "adjacent" to, "interacts" in some areas with, another domain: transportation with logistics, healthcare with insurance, banking with securities trading and/or insurance, and so on.

Characterisation. By *domain requirements fitting* we understand an operation that applies to two or more, say m, projected and possibly determined, instantiated and extended domain descriptions, i.e., to two or more, say m, original domain requirements prescriptions, and yields $m + n$ (resulting, revised original plus new, shared) domain requirements prescriptions. The m revised original domain requirements prescriptions resulting from the fitting prescribe most of the original (m) domain requirements. The n (new, shared) domain requirements prescriptions resulting from the fitting prescribe requirements that are shared between two or more of the m revised original domain requirements. ▪

Example 19.18 *Shared Domain Requirements:* Let the domain be that of multi-modal transportation nets: A multi-modal transportation net has segments (roads, rail lines, air lanes and shipping lanes) and junctions (street intersections, train stations, airports and harbours). Segments and junctions are uniquely identified. Segments possess attributes: to which two junctions they are connected, length, standard traversal time, standard traversal cost, wear-and-tear (relevant for rail lines and roads), modality (whither road, rail, air lane or shipping lane), and possibly other attributes. Junctions also possess attributes: to which one or more segments they are connected, standard traversal time, standard traversal cost (which is a function of the entry and exit segments: if of the same segment modality then maybe the cost is zero whereas if of different segment modalities then it reflects the cost of transfer (unloading and loading), and the set of one or more modalities of the connected segments. One can speak of paths, from junction via a segment to a connected junction, and routes — as sequences of connected paths. Hence one can speak of the longest route(s) and the shortest standard traversal time between two junctions. One can also speak of best wear-and-tear quality route(s) also between two junctions.

We outline two rough text original domain requirements.

A transportation net maintenance support system: The software package for this support system shall help rail line maintenance planners to identify segments (i.e., lines) in need of immediate repair (that is, corrective maintenance) or scheduled preventive maintenance (that is inspection), and, when such has been effected to record the (new) wear-and-tear status of maintained segments. These requirements imply further determination of segment attributes. Etcetera.

A transportation net logistics support system: The software package for this support system shall help combined road-rail travel planners to identify combinations of one or more of shortest length route(s), shortest traversal time route(s), least costly route traversal(s), and/or route(s) with fewest transfers between transport modalities. Etcetera.

The shared domain requirements are the following: Nets consisting of segments and junctions, thus also identification of segments and junctions; provision for segment attributes; and ability to select segments of a given modality.

We leave it to the reader to formulate what is specific to the two revised original domain requirements.

Exercise 19.3 asks that you provide formal models of the domain, the two original requirements, and the 2+1 revised original + shared domain requirements outlined above. ▪

Another example:

Example 19.19 *Fitting of Passenger Transfers Between Busses and Trains:* We assume that there are two domain requirements prescriptions, one for metropolitan bus systems of bus lines, bus stops, etc., and one for railway systems of rail lines and stations. We further assume that one of the prescriptions has been in existence for some time — maybe even that an existing product is based on those requirements — and that the other prescription is currently being developed.

Rough sketches are as follows:

The bus system consists of a set of bus lines, each being numbered and otherwise designated in a bus timetable, where this bus timetable, modulo "every" hour, for every bus line, specifies at which minutes ("past the hour") the bus stops at each stop of the line. After this there follow a number of other entity, function and possibly behaviour descriptions.

Formal Presentation: Fitting of Passenger Transfers, I

```
scheme BUS =
  class
    type
      BSn, BLn, Min
      BTT′ = BLn →m (BSn × Min)*
      BTT = {| btt:BTT′ • wf_BTT(btt) |}
```

```
    value
      wf_BTT: BTT′ → Bool
      ...
  end
```

The railway system consists of a set of train lines, each being numbered and otherwise designated in a train timetable, where this timetable, modulo "every" hour, for every train line specifies at which minutes ("past the hour") the train stops at stations of the line. After this there follow a number of other entity, function and possibly behaviour descriptions.

Formal Presentation: Fitting of Passenger Transfers, II

```
scheme RAIL =
  class
    type
      Sn, RLn, Min
      RTT′ = RLn m→ (Sn × Min)*
      RTT = {| rtt:RTT′ • wf_RTT(rtt) |}
    value
      wf_RTT: RTT′ → Bool
      ...
  end
```

Now the "fitting": Certain stations (bus stops) are to be designated as bus (train) transfer stations (bus stops). Passenger travel routes may include transfers at such stations (bus stops) between buses and trains. After this there follows a number of other entity, function and possibly behaviour prescriptions.

Formal Presentation: Fitting of Passenger Transfers, III

```
scheme BUS_RAIL =
  extend BUS with extend RAIL with
  class
    type
      Transfer′ = Bsn m→ Sn
      Transfer = {| tr:Transfer′ • card dom tr = card rng tr |}
    value
      ...
  end
```

End of Example 19.19 ▪

19.4.9 Discussion: Domain Requirements

We have outlined five reasonably distinguishable operations that the requirements engineer may need perform in order to construct a domain requirements prescription. There may be other such operations. The above five have been found useful in several development projects. Knowing about them, their underlying principles, and their techniques and tools should help the requirements engineer to more efficiently acquire domain requirements prescriptions, and to document them, i.e., to structure their documentation logically.

19.5 Interface Requirements

Characterisation. By *interface requirements* we understand those requirements that are expressed solely in terms of such phenomena and concepts that are *shared* between the domain and the machine. The machine is the hardware to be prescribed and the software to be developed. ▪

The term 'shared' is crucial. For "something" to be *shared* between the domain and the machine, that "something" must be present in the domain. It must be en entity, a function, an event or a behaviour which has been projected, instantiated, possibly made more deterministic, possibly extended and possibly fitted. And that "something" must be present in the machine: Its attributes, including value, if an entity, must "somehow" be more or less regularly monitored by (read in from the domain, or set by, output from the) machine. Its functionality, if a function, must somehow replace that "present" in, or "co-opted", taken over from the domain, and its behaviour, if a behaviour, must somehow "simulate" the behaviour of the domain, or its occurrence, if an event, must somehow be replicated: If in the domain, then recorded by the machine, and if in the machine, then signaled to the domain.

The "something" is said to be a *shared phenomenon cum concept*. We use the "somehow" hedge to indicate to the reader that the interface requirements shall stipulate, shall prescribe that 'somehow'! Shared phenomena cum concepts is what this section (Sect. 19.5) is all about! The shared "things" are usually phenomena in the domain, but always concepts in the machine. Domain concepts can also be shared.

Example 19.20 *Shared Phenomena:* We may think of a train traffic monitoring and control system being interface requirements developed. The following phenomena are identified as among those being shared: *rail units, signals, road level crossing gates, train sensors* (optical sensor sensing passing trains) and *trains*. ▪

Example 19.21 *Shared Concepts:* We continue Example 19.20. The following train traffic concepts are among those being identified as being shared: *state of units,* including whether a unit is *open, closed, reserved, occupied,* etc., *routes* (a route is, in general, not humanly visible (being often geographically widespread)), and hence *open routes.* ▪

19.5.1 Shared Phenomena and Concept Identification

A crucial step of requirements development is therefore that of identifying, from among the many phenomena and concepts of the projected (etc.) domain which of these are shared. Examples 19.20 and 19.21 gave informal, rough-sketch examples. Whether and how to categorise these shared phenomena and concepts is what the rest of this section on interface requirements is about.

Suffice it to state that we here expect that the requirements engineers — in close collaboration with requirements stakeholders — list these shared "things", and, along the road, while individually pursuing any one of the interface requirements facets, annotate this list with classifiers (whither one of the six interface requirements facets treated next, "where used", etc.).

19.5.2 Interface Requirements Facets

We shall consider six kinds of interface requirements:

- *shared data initialisation requirements,*
- *shared data refreshment requirements,*
- *computational data and control requirements,*
- *man-machine dialogue requirements,*
- *man-machine physiological interface requirements*, and
- *machine-machine dialogue requirements.*

We foresee further identification of (i.e., other) interface requirements facets than the six so far listed. And we foresee an analysis, in the future, of some of the six listed facets into a more finely granulated set of (more or less) orthogonal interface requirements facets. Suffice it now, for the purposes of this part of this volume, namely that of presenting basic principles and techniques of requirements engineering, to bring in just these six facets.

The first three interface requirements facets motivate the need for the last three interface requirements facets. Shared data generally reside in the domain and in the machine. Computational data and control typically (but do not exclusively) reside in the human users who may interface with the machine during its computations, i.e., may interact with the machine. These first three interface requirements facets prescribe what information shall (need to) be shared, as well as some abstract principles according to which the external domain information shall be communicated into internal machine data and vice versa. The dialogue requirements facets prescribe how that information

concretely shall be communicated between humans and/or other machines (and equipment in general) and the machine being requirements prescribed. We now explain these six facets of interface requirements. But first we bring in a brief aside.

19.5.3 Interface Requirements and the Requirements Document

Some remarks need to be made before we go into details of domain requirements modelling techniques.

Requirements for "Input/Output"

Interface requirements are about: "putting" part of the domain "inside" the machine. Interface requirements engineering is about how to get parts of the domain into a machine (to become part of its state), from the domain, or from other machines; and how to reflect [new, computed] states back into the domain, or onto other machines. Thus interface requirements are about shared (usually entity) phenomena and concepts.

Place in Narrative Document

In Sects. 19.5.4–19.5.9 we shall treat a number of interface requirements facets. Each of whichever you decide to focus on, in any one requirements development, must be prescribed. The interface requirements all take their "departure point", that is are based upon, the entire domain description, as well as potentially available machine input/output technology.

That is, the interface requirements represent a kind of "merging" of some form of the domain description, with descriptions of relevant, i.e., chosen, input/output technology. The two "merged" descriptions become a prescription, the interface requirements prescription. Since that "merge" was not present in the domain, the interface requirements prescription becomes an entirely new document part.

Place in Formalisation Document

The above statements on how to express the interface requirements also apply to formal interface requirements prescriptions.

Formal Presentation: Documentation

We may assume that there is a formal domain description, $\mathcal{D}$ (from which we develop parts of the formal prescription of the interface requirements), and narrative descriptions of the input/output technologies. We further assume that there are formal descriptions, $\mathcal{D}_{IO}$, of these input/output technolo-

gies. We then develop an entirely new document, the interface requirements, $\mathcal{R}_{I/F}$. It somehow "merges" parts of $\mathcal{D}$ with parts of $\mathcal{D}_{IO}$ into the resulting $\mathcal{R}_{I/F}$.

This section on interface requirements is about the "merge" principles and techniques.

19.5.4 Shared Data Initialisation

Information that is shared between the domain and the machine is often nontrivial in its structure and extent. Special care must be taken to introduce such information to the machine.

Characterisation. By *shared data initialisation* we understand an operation that creates a *shared data structure* in the machine. ▪

Thus a shared data initialisation requirements is an operation on requirements documents. It applies to a (projected and possibly determined, instantiated, extended and fitted) domain description, i.e., a domain requirements prescription, and yields an interface requirements prescription, where the latter prescribes that certain information of the domain is to be represented as a *shared data structure* in the machine, and generally how such data is initially to be set up by the machine.

Example 19.22 *Shared Data Initialisation of Railway Net:* We rough-sketch illustrate a case of shared data initialisation based on the rough sketch of Example 11.8 (Page 265). The software system shall start in an initial state which — rough-sketching — represents an empty rail net, and "ends" in a state which includes a representation of an "entire" rail net, i.e., a representation of all static and dynamic properties of each and every rail unit. In addition — as will be seen from other parts of these domain requirements[5] — it shall be possible to simply relate rail units to their physical surroundings: whether the rail runs along a platform, in a tunnel, up/down hill, is curved, etc.; the pertinent electric train power line segment; etc. A special software subsystem shall handle the initial establishment of this start state as follows: ..., etc.

We refer to Exercise 19.11 which asks that you complete the "..., etc." and provide a formalisation of the above. ▪

The ellipses, ..., indicate that a longer narrative follows. The whole thing can furthermore be formalised on the basis of formalisation of the projected, determinate, instantiated, and possibly extended and fitted domain requirements. We leave that as an exercise (cf. Exercise 19.2).

[5] This is not illustrated in these examples.

19.5.5 Shared Data Refreshment

Shared data, once initialised, usually need be kept updated. The domain — usually — changes, irrespective of any computing system inserted into it.

Characterisation. 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*. ▪

Thus a shared data refreshment requirements is an operation on requirements documents. It applies to an interface requirements prescription, where the latter prescribes that certain information of the domain is to be represented as a *shared data structure* in the machine. The shared data refreshment requirements then prescribe how often, and by which means, that shared data structure is to be refreshed (i.e., updated).

Example 19.23 *Shared Data Refreshment of Railway Net:* We continue Example 19.22 by providing a rough sketch of a shared data refreshment requirements. Regular inspections of the wear and tear of the rail net units, signals, optical gates (and other sensors), road level crossings, etc., shall lead to similarly updating of that equipment's shared data structure, and such regular inspections shall be prompted by the machine and as prescribed by the required software. Inspections, with resulting updates, may take place before the usual expiry of inspection interval. And so on.

We refer to Exercise 19.12 which asks that you complete the "And so on" and to provide a formalisation of the above. ▪

The ellipses, ..., indicate that a longer narrative follows. The whole thing can furthermore be formalised on the basis of formalisation of the interface requirements resulting from shared data initialisation. We leave that as an exercise (cf. Exercise 19.11).

19.5.6 Computational Data and Control Interface Requirements

For many applications it is the case that the flow of computations that may be desired by the users, i.e., the stakeholders, shall be influenced by interaction between the machine and these users. That is: It is often to be prescribed how such interaction shall take place, whether by users interrupting the machine, or the machine polling the users, and what it shall entail, i.e., which computational consequences the user interference shall have. It is this, perhaps "grey-zone" facet that we call the computational data and control interface.

Characterisation. By *computational data and control interface requirements* we understand requirements which prescribe that certain forms of input be provided over the user-machine interface, in order to help control the flow of computation: when to start or stop certain subcomputations, and/or with which argument data such subcomputations should be carried out, etc. ▪

The argument data may characterise certain "boundary" conditions, or initial program points, or other, for such subcomputations.

Example 19.24 *Computational Data and Control Interface:* We continue Example 19.22. In that example reference (...) was made to a software subsystem. It is this software subsystem which, such as we (now) requirements specify it, needs frequent computational data and control directives from the person or persons who monitor the input of the mass data. The railway net is represented, in the machine (database), by geographical area (i.e., area by area). Input of rail unit data is, in batches, by such areas. Hence a computational data input specifies that "until further notice" the next many future unit inputs are intended to "belong" to that area. Another computational data input (i.e., the "further notice") specifies "the end" of such a series of area-specific unit data. Occasionally, during unit data input, that and past input may need be checked ("vetted"). Hence a computational data input may specify that such vetting is to be performed,[6] and other, immediately subsequent computational data input may be prompted as to the specific nature of the desired checks. Finally, prompts may inquire as to whether further checks need to be done, or the check series terminated. (We do not here specify the vetting procedures.) ∎

The computational data and control interface is typically specified, semiformally, by means of message or live sequence charts (MSCs [182–184], respectively LSCs [73, 149, 203]), or by formal RSL/CSP specifications. RSL/CSP was covered in Vol. 1, Chap. 21. MSC and LSC were covered in Vol. 2, Chap. 13.

19.5.7 Man-Machine Dialogue

Characterisation. By *man-machine dialogue requirements* we understand the prescription of the syntax (including sequential structure) and semantics of the communications (i.e., messages) transferred, in either direction, over the interface between man and machine, whether communicated textually through a keyboard (by the human) or on the screen (by machine), by a mouse or other tactile means (by human), or by voice (by human) or sound (by machine). ∎

It must be stressed that the man-machine dialogue referred to above subsumes the physiological interfaces mentioned next, but that it emphasises the sequencing of possibly alternative events and messages. Thus man-machine dialogue is "overall" wrt. the individual man-machine physiological events and messages.

[6] We envisage that certain kinds of checks cannot be performed concurrently with the unit input.

Example 19.25 *Man-Machine Dialogue Requirements:* We continue Example 19.23.

When, for any rail unit, its wear and tear information becomes older than six months, a message is to be displayed on the console (screen) of the railway net maintenance group responsible for that rail unit (this is an interface requirement). This group must respond within 72 hours with the requested update information (this is a business process reengineering requirement). ▪

Man-machine dialogues are typically specified, semiformally, by means of message or live sequence charts (MSCs [182–184], respectively LSCs [73, 149, 203]), or by formal RSL/CSP specifications.

19.5.8 Man-Machine Physiological Interface

Humans can, "thanks" to a variety of technological "gadgets", communicate with computers in various ways. (i) Besides the conventional keyboard, they can also communicate by other tactile means: (ii) the "mouse"; (iii) "pointing with fingers at the screen"; "pressing, with, for example, fingers", fields of the screen; etc.; (v) possibly by voice, etc. These technological "gadgets" imply the man-machine physiological interface. Computers can likewise communicate with humans by means of graphics and sound.

Characterisation. By *man-machine physiological interface* we understand the possibly combined use of three forms of man-machine interfaces: (A) *graphical (visual) user interface*, (B) *audio (voice, sound) interface* and (C) *tactile (keyboard, touch, "point", button, etc.) interface*. ▪

Example 19.26 *Man-Machine Physiological Interface Requirements of Railway Net Status Input:* We continue Example 19.25. If no update of a rail unit's wear and tear status has occurred within 72 hours of its visual display request, then a series of alarm bells shall sound (...) with one-hour intervals in designated offices of the railway groups responsible for recording this status, and, synchronised with this, bright red alarm lamps shall blink in line and station management offices. ▪

By a *graphical user interface* (GUI) we understand a visual display unit (VDU, e.g., a colour screen). Typically the VDU screen can be programmed to display various "windows", icons, scroll-down "curtains", etc., with these being possibly labelled, and/or providing fields for text (keyboard) input.

Example 19.27 *Man-Machine Physiological Interface Requirementsof GUIs and Databases:* Assume that a database records the data which reflects the topology of some railway net, or that records the contents of a timetable. Also assume that some graphical user interface (GUI) windows represent the

interface between man and machine such that items (fields) of the GUI are indeed "windows" into the underlying database. We prescribe and model, as an interface requirements, such GUIs and databases, the latter in terms of a relational, say a SQL, database:

Formal Presentation: GUIs and Databases, I

type
  Nm, Pos, Rn, An, Txt
  GUI = Nm $\overrightarrow{m}$ (Item × Pos)
  Item = Txt × Imag
  Imag = Icon | Curt | Tabl | Wind
  Icon == mk_Icon(val:Val)
  Curt == mk_Curt(vall:Val*)
  Tabl == mk_Tabl(rn:Rn,tbl:TPL-**set**)
  Wind == mk_Wind(gui:GUI)

Annotations:

- A gui:GUI item, irrespective of the position, pos:Pos, of that item on the screen,
- maps distinct item names, Nm, into items, item:Item.
- An item has some "labeling" text, txt:Txt, and an image, imag:Imag.
- An image, imag:Imag, is either an icon, icon:Icon, a curtain, curt:Curt, a table, tabl:Tabl, or a window, wind:Wind.
- An icon has a value, mk_Icon(val:Val).
- A curtain consists of a list of values, mk_Curt(vall:Val*).
- A table, mk_Tabl(rn:Rn,tbl:TPL-**set**), names the relation, rn:Rn, from which the set tuples, tbl:TPL-**set**, of the table are queried.
- A window, mk_Wind(gui:GUI), is, hence recursively, a graphical user interface.

Formal Presentation: GUIs and Databases, II

  Val = VAL | REF | GUI
  VAL = mk_Intg(i:Intg) | mk_Bool(b:**Bool**)
    | mk_Text(txt:**Text**) | mk_Char(c:**Char**)

Annotations:

- A value (val:Val) is
  - ⋆ either a proper value (in VAL),
  - ⋆ or a reference (to a database entry),
  - ⋆ or a graphical user interface (gui:GUI).

- A proper value (val:VAL) is
  - ⋆ either an integer (mk_Intg(i:Intg)),
  - ⋆ or a Boolean truth (mk_Bool(b:**Bool**)) value,
  - ⋆ or a text string mk_Text(txt:**Text**) value,
  - ⋆ or a character mk_Char(c:**Char**) value.

Formal Presentation: GUIs and Databases, III

```
RDB = Rn ~m~> TPL-set
TPL = An ~m~> VAL
REF == mk_Ref(rn:Rn,an:An,sel:SEL)
SEL = An ~m~> OptVal
OptVal == null | mk_Val(val:VAL)
```

Annotations:

- A relational database (rdb:RDB) maps unique relation names (rn:Rn) into relations, and these are sets of tuples (tpls:TPL-**set**).
- A tuple (tpl:TPL) maps unique attribute names into proper values (val:VAL).
- A reference (is a proper value and) consists of a relation name, (rn:Rn), an attribute name (an:An) and a selection criterion (sel:SEL).
- A selection criterion (An $\overrightarrow{m}$ OptVal) is a possibly empty map from attribute names into possibly optional, proper values.
- An optional value is either `nil`, or is a proper value (mk_Val(val:VAL)).

Further on database references: Wherever, in a GUI, there is a reference, it is the value designated by that reference which is displayed. The reference relation name designates a relation in the database. The reference attribute name, an, designates an attribute of any tuple in the designated relation. If there is a tuple in the relation whose values equal those expressed in the selector, attribute by attribute, then that tuple's value at an is the value displayed; otherwise the optional (i.e., the so-called surrogate) value `null` is displayed. That is, the reference is a hidden quantity.

Formal Presentation: GUIs and Databases, IV

```
value
  de_ref: REF × RDB → OptVAL
  de_ref(mk_Ref(rn,an,sel))(rdb) ≡
    if ∃ tpl:TPL • tpl ∈ rdb(rn)∧tpl/dom sel = sel
      then
        let tpl:TPL • tpl ∈ rdb(rn)∧tpl/dom sel = sel in
        tpl(an) end
      else null
```

```
    end
    pre rn ∈ dom rdb ∧
        ∃ tpl:TPL•tpl ∈ rdb(rn) ∧ dom sel ∪{an}⊆dom tpl
```

Annotations:
Further on database references:

- To de_reference a database reference
- consisting of a relation name, rn,
- an attribute name, an, and
- a selection criterion, sel,
- is to inquire whether there exists a tuple, tpl,
- in the name relation, rdb(rn),
- for which the selection criterion applies: tpl/**dom** sel = sel.
- If such a tuple is found, then it is the result of the dereferencing;
- if not, then the `null` value is yielded.

Icons effectively designate a system operator or a user-definable constant or variable value, or a value that "mirrors" that found in a relation column satisfying an optional value (OptVal), and similarly for curtains and tables. Tables more directly reflect relation tuples (TPL). GUIs (windows) are defined recursively.

If, for example, the names space values of Nm, Rn, and An, and the chosen constant texts, Txt, suitably mirror names and phenomena of the domain, then we may be on our way to satisfying a "classical" user interface requirement, namely that *"the system should be user friendly"*.

Thus a definition, much like the one of GUI above, is, in a sense, pulled out of the "thin" air and presented, without much further ado, as part of an interface requirements. Where was its domain "counterpart"? Or one might just be content with the reuse of the above definition.

For a specific interface requirements there now remains the task of relating all shared phenomena and data to one another via the GUI. In a sense this amounts to mapping concrete types onto primarily relations, and entities of these (phenomena and data) onto the icons, curtains, and tables. ▪

Example 19.28 *Man-Machine Physiological Interface Requirements: A Specific GUI for Timetables:* We exemplify a very simple GUI. We omit naming the only three items: (i) the scroll-down curtain which displays (i.e., lists) the client and staff commands — as well as the no command (`nil`); (ii) a prompt field which initially is blank, i.e., nil, but which — depending on the clicked command name of the scroll-down curtain — lists the command field

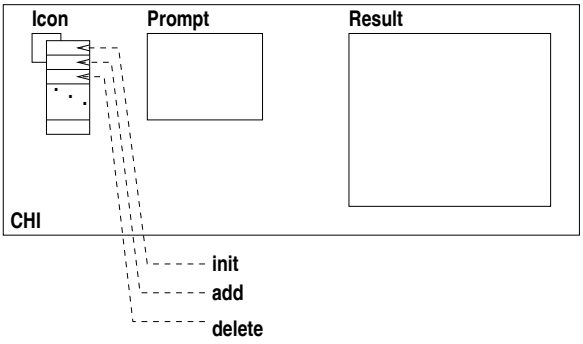


Fig. 19.2. An example CHI: staff clicking icon

names for desired values, and for which the user (client or staff) is to provide appropriate text values; (iii) finally, a result field.

Formal Presentation: Specific GUI of Timetable, I

type
GUI = Curt × Prompt × Result
Curt == browse | display | connection | init | add | delete | nil
Prompt = Query | Update | Conn | nil
Result = RES

Annotations:

- The graphical user interface, gui:GUI, consists of three items:
  - ⋆ a scroll-down curtain, curt:Curt,
  - ⋆ a prompt field, prompt:Prompt,
  - ⋆ and a result field, result:Result.
- A scroll-down curtain in the concrete lists exactly the available query and update commands possible on a timetable.
- These are designated by the keywords: browse, display, connection, init, add and delete.
- At most one of these keywords can be selected, i.e., is therefore highlighted. Thus the above model defines a curtain to be just one of these, or, when none is selected, the nil option.
- The prompt field, prompt:Prompt, is to contain an appropriate query/update command, as "selected" by the curtain highlight, or nil.
- The result field, res:Result, will contain a result value.

In Example 19.14 we defined the semantics of query and update commands. We now use these definitions to define the requirements, namely that these commands obtain their arguments, and, when subject to execution, deliver

(deposit) their result into the user interface, that is, as part of the GUI. We exemplify, perhaps rather too extensively, the resulting query and update function semantics. First the query commands:

Formal Presentation: Specific GUI of Timetable, II

```
value
  client: GUI → TT → GUI
  client(,,)(tt) ≡
    let icon = browse ⌈⌉ display ⌈⌉ connection in
    case icon of:
      browse → (browse,mk_Brws(),M_q(mk_Brws())(tt)),
      display
        → let fn:Fn • fn ∈ dom tt ∨ ... in
            (display,mk_Disp(fn),M_q(mk_Disp(fn))(tt)) end,
      connection
        → let ℓ:Nat,da,ta:An•{da,ta}⊆Ans(tt) ∧ ... in
            (connection,
                mk_Conn(ℓ,da,ta),
                    M_q(mk_Conn(ℓ,da,ta))(tt)) end
    end end
```

Annotations:

A client, by his own decision, either issues a browse, or a display, or a connection query.

- If browse then
  - ⋆ it means that the curtain alternative browse has been "clicked", and is hence highlighted,
  - ⋆ that the prompt field shows an obvious mk_Brws() command, requiring no arguments,
  - ⋆ and the result field shows the result, $\mathcal{M}_q$(mk_Brws())(tt), of interpreting that command on the timetable.
- If display then
  - ⋆ it means that the curtain alternative display has been "clicked", and is hence highlighted,
  - ⋆ that a flight number is provided by the client, here shown as nondeterministically selected,
  - ⋆ that the prompt field shows the corresponding display command, mk_disp(fn),
  - ⋆ and the result field shows the result, $\mathcal{M}_q$(mk_Disp(fn))(tt), of interpreting that command on the timetable.
- If connection then

- ⋆ it means that the curtain alternative display has been "clicked", and is hence highlighted,
- ⋆ that the maximum number of flight changes, ℓ, and departure da and destination ta airports are provided by the client, here shown as non-deterministically selected,
- ⋆ that the prompt field shows the corresponding connection command mk_Conn(ℓ,da,ta),
- ⋆ and the result field shows the result of interpreting that command on the timetable, $\mathcal{M}_q$(mk_Conn(ℓ,da,ta))(tt).

Formal Presentation: Specific GUI of Timetable, III

Then the semantics of update commands wrt. the graphical user interface:

```
value
  staff: GUI → TT → GUI × TT
  staff(,,)(tt) ≡
    let icon = init ⌈⌉ add ⌈⌉ delete ⌈⌉ ... in
    case icon of:
      init → let (r,tt′) = M_u(mk_init())(tt) in ((init,tt′,r),tt′) end,
      add → let fn:Fn,j:Journey • fn ∉ dom tt ∨ ... in
              let (r,tt′) = M_u(mk_add(fn,j))(tt) in
              ((add,mk_add(fn,j),r),tt′) end end,
      delete → let fn:Fn • fn ∈ dom tt ∨ ... in
              let (r,tt′) = M_u(mk_del(fn))(tt) in
              ((delete,mk_del(fn),r),tt′) end end
    end end
```

Annotations: We leave annotations as an exercise to the reader.

The semantics functions illustrate the internal nondeterministic choices that the client, respectively the staff, makes — as seen from the point of view of the semantics — of the parameters that go into the specific query, respectively update commands. For the display query it is the choice of the flight number. For the connection query it is the choice of the maximum number of changes of flights, as well as the choice of the from (departure, or airport of origin) and to (destination) airports. For the add journey update it is the choice of the flight number and the journey (of that flight). For the delete flight update it is the choice of the flight number.

We "reassemble" the above formula into the previously defined system function, cf. Example 19.14. Before we had:

Formal Presentation: Specific GUI of Timetable, IV

```
value
  system: TT → Unit
```

(**let** q:Query **in let** v = $\mathcal{M}_q$(q)(tt) **in** system(tt) **end end**)
⌈⌉ (**let** u:Update **in let** (r,tt′) = $\mathcal{M}_u$(q)(tt) **in** system(tt′) **end end**)

Annotations:

- The system nondeterministically (internally, ⌈⌉) chooses
- whether to engage in a q:Query behaviour,
- or in an u:Update behaviour.
- In either case a command is arbitrarily selected and interpreted on the global timetable tt.
- The system then continues with a possibly updated timetable tt′.

Now we get:

Formal Presentation: Specific GUI of Timetable, V

value
system: GUI → TT → **Unit**
(**let** gui′ = client(gui)(tt) **in** system(gui′)(tt) **end**)
⌈⌉ (**let** (gui′,tt′) = staff(gui)(tt) **in** system(gui′)(tt′) **end**)

Annotations:

- The system, still nondeterministically (internally, ⌈⌉) chooses, but now between
- either the client behaviour
- or the staff behaviour.
- In both cases, the system "temporarily" hands either of these behaviours, the timetable tt.

■

19.5.9 Machine-Machine Dialogue

The desired machine is usually serving in a context in which it has been fitted to other machines or to supporting technologies. These may provide sensory data or accept actuation (i.e., control) data. Some fitted machines may provide for, or accept mass data transfers. Usually supporting technologies provide for, or accept rather "small", i.e., single (simple) data transfers.

Characterisation. By *machine-machine dialogue requirements* we understand syntax (incl. sequential structure) and semantics (i.e., meaning) of the communications (i.e., messages) transferred in either direction over the automated interface between machines (including supporting technologies). ■

Example 19.29 *Machine-Machine Dialogue Requirements: A Simple Cabin Tower Rail Switch Monitoring and Control:*

This example is from a rather outdated railway station. Today's railway stations provide for what is known as interlocking: The simultaneous setting and resetting of several, i.e., groups of switches and signals.

Rail switches are assumed, upon request, to provide sensory signals, which report on their state: "straight" or "turn-off". And these rail switches will respond to control signals which, within an assumed response time of their being issued, set the switch to a desired state ("straight" or "turn-off"). The cabin tower maintains a display which shows the states of all switches in its associated station. Associated with this cabin tower display are two buttons: Pressing either of these shall correspond to sending "straight" or "turn-off" control signals. Only one of these buttons can be pressed in any one-minute interval. At half-minute intervals each switch reports its status, and that status shall be reflected in the cabin tower display. When a "straight" or "turn-off" control button is depressed, then a signal shall be sent to the designated switch, and that switch shall react accordingly within a 15-second time lapse. The cabin tower switch display shall sound and flash appropriate alarms if the switch status, within half a minute, is not the desired (control signalled) one. ▪

The above example admittedly provides only a very rough sketch indication. It also "links" up to (that is, strongly depends on related) machine (including support technology) requirements, as covered next.

Example 19.30 *Machine-Machine Dialogue Requirements: Bulk Data Communication:* Suppose that an application calls for the massive transfer of data over noisy distances. That is, the probability that transferred data may be corrupted, i.e., change value during communication, is considerable. What is known as a suitable data communication protocol therefore has to be prescribed, one that helps ensure detection of corrupted data so as to enable retransmission until it has been decided that a correct, i.e., uncorrupted, transfer has been completed.

These data communication protocols are of the kind that we would call machine-machine dialogues. Other than treating this as a metaexample we shall not go into detail in this book, but refer to, for example, [332] for a more authoritative treatment. ▪

19.5.10 Discussion: Interface Requirements

Dialogue Prescription Techniques and Tools

We have not, in this section on interface requirements, shown any examples of, or formalised the dialogue aspects of interface requirements. The term

interface implies at least two interacting behaviours. Therefore techniques and tools (i.e., notations) for process modelling are used in such formalisations. We refer to Vol. 1, Chap. 21 (*Concurrent Specification Programming*) and Vol. 2, Chap. 13 (*Message and Live Sequence Charts*), where we cover formal tools and techniques for modelling such interaction.

General

We have outlined six reasonably distinguishable facets that the requirements engineer may need perform in order to construct an interface requirements prescription. There may be other such facets. The above six have been found useful in several development projects. Knowing about them, their underlying principles, and their techniques and tools should help the requirements engineer to more efficiently acquire interface requirements prescriptions, and to document them, i.e., to structure their documentation logically.

Special Principles and Techniques

Interface requirements, in most people's minds and expression, are concerned with so-called "user-friendliness". That is, interface requirements focus, very much, on the form of the dialogues and the layout of GUIs. Much can be said about this. We shall venture our definition of "user-friendliness".

Characterisation. By a *user-friendly man-machine interface* we understand one which somehow satisfies the following criteria:

- **Faithful:** The interface reflects only the shared phenomena and concepts, and reflects "absolutely" no machine (i.e., hardware + software) concepts (i.e., jargon). That is, the terminology used "across" the interface is that of the domain.
- **Didactic:** The sequence of presentation of shared phenomena and concepts reflects some clarified view on how these phenomena and concepts relate, which are the more important ones, and which reflect current or changing business processes, support technologies, managements and organisations, rules and regulations, etc.
- **Pedagogic:** The number of phenomena and concepts presented in any one step of interaction is small, say from one to at most five. The order of presentation is initially from core phenomena and concepts to increasingly derived phenomena and concepts. That order may initially be pedantic, but is accepted by novice users. For more experienced users means for clear, logical "shortcuts" should be made available.
- **Physiologic:** The number of current and alternative physiologic "gadgets"[7] needed to maintain interaction should be modest and be balanced against simplicity or complexity of interaction.

[7] Screen, keyboard, mouse, other tactile instruments ("pointing to", pressure-sensitive screens), audio (i.e., loudspeakers), microphone, etc.

- **Psychologic:** Interaction response, incl. prompt times and texts should not irritate[8] or shame the users, or make these users feel inadequate, or guilty (say, of "not knowing").
- **Artistic:** And then it is certainly user-friendly, this author believes, if the interface reflects some artistic ideas.

The above characterisation is only approximate. We also refer to Sect. 6.2 for a discourse on "What Is Art?". ▪

If referring to special textbooks [233,312] on the subject, we advise the reader to pay strict attention to the issues we have raised: Make sure that interface requirements, when referring to phenomena and concepts, refer "strictly" to those that are well understood in the domain.

19.6 Machine Requirements

Characterisation. By *machine requirements* we understand those requirements that can be expressed solely in terms of (or with prime reference to) machine concepts. ▪

19.6.1 Machine Requirements Facets

We shall, in particular, consider the following five kinds of machine requirements: *performance requirements, dependability requirements, maintenance requirements, platform requirements* and *documentation requirements*. There may be other kinds of machine requirements, but these suffice to sharpen our quest for comprehensive requirements. And there may be machine requirements which are "not quite" one or the other of the kinds listed above, or which also contain (albeit minor) uses of terms of the domain without being "typical" interface requirements.[9] We now cover each of the main kinds of machine requirements identified above.

19.6.2 Machine Requirements and the Requirements Document

Some remarks need to be made before we go into details of domain requirements modelling techniques.

[8] The response to a user query, which took the user maybe a minute to prepare, should not follow the submission of that query in the order of microseconds, rather 1.5–3 seconds is more pleasing, psychologically. For short, "click"-type "queries", response times of 100 milliseconds seem OK.

[9] The use of domain terms, for us still to claim that the requirements are proper machine requirements, must be of generic nature, that is, they can be substituted by terms from other domains without changing the real nature of the machine requirements.

Requirements for "the Machine Only"

Machine requirements are about the machine only! They, the machine requirements, "in the extreme" contain no references to any specific aspect of the domain.

But there may be general references, and they could be of the same nature for whichever domain was the base, such as, such and such function invocations shall terminate in less than m microseconds, whereas such and such function invocations shall terminate in less than n seconds. Or, such and such data shall be replicated for back-up reasons, or auxiliary storage for performing such and such functions shall be less than 500 KB.

The machine requirements all take their "departure point", that is, are based upon, potentially available machine technology, whether central, or distributed, or input/output, or peripheral.

Place in Narrative and Formalisation Document

In Sects. 19.6.3–19.6.8 we shall treat a number of machine requirements facets. Each of whichever you decide to focus on, in any one requirements development, must be prescribed.

The machine requirements are really void of any (material) reference to domain phenomena and concepts. Hence the machine requirements prescriptions form a separate, "freestanding" document. That document must describe both the machine component (i.e., hardware, and software) interfaces and functionalities (the latter, say, in pre/postcondition form).

19.6.3 Performance Requirements

Characterisation. By *performance requirements* we mean machine requirements that prescribe storage consumption, (execution, access, etc.) time consumption, as well as consumption of any other machine resource: number of CPU units (incl. their quantitative characteristics such as cost, etc.), number of printers, displays, etc., terminals (incl. their quantitative characteristics), number of "other", ancillary software packages (incl. their quantitative characteristics), of data communication bandwidth, etcetera. ∎

Pragmatically speaking, performance requirements translate into financial resources spent, or to be spent.

Example 19.31 *Performance Requirements: Timetable System Users and Staff — Narrative Prescription Unit:* We continue Example 19.16. The machine shall serve 1000 users and 1 staff member. Average response time shall be at most 1.5 seconds, when the system is fully utilised. ∎

Till now we may have expressed certain (functions and) behaviours as generic (functions and) behaviours. From now on we may have to "split" a specified behaviour into an indexed family of behaviours, all "near identical" save for the unique index. And we may have to separate out, as a special behaviour, (those of) shared entities.

Example 19.32 *Performance Requirements: Timetable System Users and Staff:* We continue Example 19.14 and Example 19.31. In Example 19.14 the sharing of the timetable between users and staff was expressed parametrically.

Formal Presentation: Timetable System Users and Staff, I

```
system(tt) ≡ client(tt) ‖ staff(tt)

client: TT → Unit
client(tt) ≡ let q:Query in let v = M_q(q)(tt) in system(tt) end end

staff: TT → Unit
staff(tt) ≡
   let u:Update in let (r,tt′) = M_u(u)(tt) in system(tt′) end end
```

We now factor the timetable entity out as a separate behaviour, accessible, via indexed communications, i.e., channels, by a family of client behaviours and the staff behaviour.

Formal Presentation: Timetable System Users and Staff, II

```
type
   CIdx /∗ Index set of, say 1000 terminals ∗/
channel
   { ct[i]:QU,tc[i]:VAL | i:CIdx }
   st:UP,ts:RES
value
   system: TT → Unit
   system(tt) ≡ time_table(tt) ‖ (‖ {client(i)|i:CIdx}) ‖ staff()

   client: i:CIdx → out ct[i] in tc[i]  Unit
   client(i) ≡ let qc:Query in ct[i]!M_q(qc) end tc[i]?;client(i)

   staff: Unit → out st in ts  Unit
   staff() ≡ let uc:Update in st!M_u(uc) end let res = ts? in staff() end

   time_table: TT → in {ct[i]|i:CIdx},st  out {tc[i]|i:CIdx},ts  Unit
   time_table(tt) ≡
```

```
  □ {let qf = ct[i]? in tc[i]!qf(tt) end | i:CIdx}
  □ let uf = st? in let (tt′,r)=uf(tt) in ts!r; time_table(tt′) end end
```

Please observe the "shift" from using ⌈⌉ in system earlier in this example to □ just above. The former expresses nondeterministic internal choice. The latter expresses nondeterministic external choice. The change can be justified as follows: The former, the nondeterministic internal choice, was "between" two expressions which express no external possibility of influencing the choice. The latter, the nondeterministic external choice, is "between" two expressions where both express the possibility of an external input, i.e., a choice. The latter is thus acceptable as an implementation of the former. ▪

The next example, Example 19.33, continues the performance requirements expressed just above. Those two requirements could have been put in one phrase, i.e., as one prescription unit. But we prefer to separate them, as they pertain to different kinds (types, categories) of resources: terminal + data communication equipment facilities versus time and space.

Example 19.33 *Performance Requirements of Storage and Speed for n-Transfer Travel Inquiries:* We continue Example 19.16. When performing the *n-Transfer Travel Inquiry* (rough sketch) prescribed above, the first — of an expected many — result shall be communicated back to the inquirer in less than 5 seconds after the inquiry has been submitted, and, at no time during the calculation of the "next" results must the storage buffer needed to calculate these exceed around 100,000 bytes. ▪

19.6.4 Dependability Requirements

To properly define the concept of *dependability* we need first introduce and define the concepts of *failure, error,* and *fault.*

Characterisation. A machine *failure* occurs when the delivered service deviates from fulfilling the machine function, the latter being what the machine is aimed at [287]. ▪

Characterisation. An *error* is that part of a machine state which is liable to lead to subsequent failure. An error affecting the service is an indication that a failure occurs or has occurred [287]. ▪

Characterisation. The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a *fault* [287]. ▪

The term hazard is here taken to mean the same as the term fault.

One should read the phrase: "adjudged or hypothesised cause" carefully: In order to avoid an unending trace backward as to the cause,[10] we stop at *the cause which is intended to be prevented or tolerated.*

Characterisation. The service delivered by a machine is its *behaviour* as it is perceptible by its user(s), where a user is a human, another machine or a(nother) system which *interacts* with it [287]. ▪

Characterisation. *Dependability* is defined as the property of a machine such that reliance can justifiably be placed on the service it delivers [287]. ▪

We continue, less formally, by characterising the above defined concepts [287]. "A given machine, operating in some particular environment (a wider system), may fail in the sense that some other machine (or system) makes, or could in principle have made, a *judgement* that the activity or inactivity of the given machine constitutes a *failure*".

The concept of *dependability* can be simply defined as "the quality or the characteristic of being dependable", where the adjective 'dependable' is attributed to a machine whose failures are judged sufficiently rare or insignificant.

Impairments to dependability are the unavoidably expectable circumstances causing or resulting from "undependability": faults, errors and failures. *Means* for dependability are the techniques enabling one to provide the ability to deliver a service on which reliance can be placed, and to reach confidence in this ability. *Attributes* of dependability enable the properties which are expected from the system to be expressed, and allow the machine quality resulting from the impairments and the means opposing them to be assessed.

Having already discussed the "threats" aspect, we shall therefore discuss the "means" aspect of the *dependability tree.*

- Attributes:
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Integrity
  - ⋆ Reliability
  - ⋆ Safety
  - ⋆ Security
- Means:
  - ⋆ Procurement
    - · Fault prevention

[10] An example: "The reason the computer went down was the current supply did not deliver sufficient voltage, and the reason for the drop in voltage was that a transformer station was overheated, and the reason for the overheating was a short circuit in a plant nearby, and the reason for the short circuit in the plant was that ..., etc."

      - Fault tolerance
    - Validation
      - Fault removal
      - Fault forecasting
- Threats:
    - Faults
    - Errors
    - Failures

Despite all the principles, techniques and tools aimed at *fault prevention*, *faults* are created. Hence the need for *fault removal*. *Fault removal* is itself imperfect. Hence the need for *fault forecasting*. Our increasing dependence on computing systems in the end brings in the need for *fault tolerance*. We refer to special texts [212, 213, 226] on the above four topics.

Characterisation. By a *dependability attribute* we shall mean either one of the following: *accessibility, availability, integrity, reliability, robustness, safety* and *security*. That is, a machine is dependable if it satisfies some degree of "mixture" of being accessible, available, having integrity, and being reliable, safe and secure. ▪

The crucial term above is "satisfies". The issue is: To what "degree"? As we shall see — in a later section — to cope properly with dependability requirements and their resolution requires that we deploy mathematical formulation techniques, including analysis and simulation, from statistics (stochastics, etc.).

In the next seven subsections we shall characterise the dependability attributes further. In doing so we have found it useful to consult [212].

Accessibility

Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Their being granted access to computing time is usually specified, at an abstract level, as being determined by some internal nondeterministic choice, that is: essentially by *"tossing a coin"!* If such internal nondeterminism was carried over, into an implementation, some *"coin tossers"* might never get access to the machine.

Characterisation. A system being *accessible* — in the context of a machine being dependable — means that some form of *"fairness"* is achieved in guaranteeing users "equal" access to machine resources, notably computing time (and what derives from that). ▪

Example 19.34 *Accessibility Requirements: Timetable Access:* Based on Examples 19.14 and 19.16, we can express: The timetable (system) shall be inquirable by any number of users, and shall be updateable by a few, so authorised, airline staff. At any time it is expected that up towards a thousand users are directing queries at the timetable (system). And at regular times, say at midnights between Saturdays and Sundays, airline staff are making updates to the timetable (system). No matter how many users are "on line" with the timetable (system), each user shall be given the appearance that that user has exclusive access to the timetable (system). ▪

Availability

Usually a desired, i.e., the required, computing system, i.e., the machine, will be used by many users — over "near-identical" time intervals. Once a user has been granted access to machine resources, usually computing time, that user's computation may effectively make the machine unavailable to other users — by "going on and on and on"!

Characterisation. By *availability* — in the context of a machine being dependable — we mean its readiness for usage. That is, that some form of *"guaranteed percentage of computing time"* per time interval (or percentage of some other computing resource consumption) is achieved — hence some form of *"time slicing"* is to be effected. ▪

Example 19.35 *Availability Requirements: Timetable Availability:* We continue Examples 19.14, 19.16, and 19.34: No matter which query composition any number of (up to a thousand) users are directing at the timetable (system), each such user shall be given a reasonable amount of compute time per maximum of three seconds, so as to give the psychological appearance that each user — in principle — "possesses" the timetable (system). If the timetable system can predict that this will not be possible, then the system shall so advise all (relevant) users. ▪

Integrity

Characterisation. A system has *integrity* — in the context of a machine being dependable — if it is and remains unimpaired, i.e., has no faults, errors and failures, and remains so, without these, even in the situations where the environment of the machine has faults, errors and failures. ▪

Integrity seems to be a highest form of dependability, i.e., a machine having integrity is 100% dependable! The machine is sound and is incorruptible.

Reliability

Characterisation. A system being *reliable* — in the context of a machine being dependable — means some measure of continuous correct service, that is, measure of time to failure. ∎

Example 19.36 *Timetable Reliability:* Mean time between failures shall be at least 30 days, and downtime due to failure (i.e., an availability requirements) shall, for 90% of such cases, be less than 2 hours. ∎

Safety

Characterisation. By *safety* — in the context of a machine being dependable — we mean some measure of continuous delivery of service of either correct service, or incorrect service after benign failure, that is: Measure of time to catastrophic failure. ∎

Example 19.37 *Timetable Safety:* Mean time between failures whose resulting downtime is more than 4 hours shall be at least 120 days. ∎

Security

We shall take a rather limited view of security. We are not including any consideration of security against brute-force terrorist attacks. We consider that an issue properly outside the realm of software engineering.

Security, then, in our limited view, requires a notion of *authorised user*, with authorised users being fine-grained authorised to access only a well-defined subset of system resources (data, functions, etc.). An *unauthorised user* (for a resource) is anyone who is not authorised access to that resource.

A terrorist, posing as a user, should normally fail the authorisation criterion. A terrorist, posing as a brute-force user, is here assumed to be able to capture, somehow, some authorisation status. We refrain from elaborating on how a terrorist might gain such status (keys, passwords, etc.)!

Characterisation. A system being *secure* — in the context of a machine being dependable — means that an *unauthorised user*, after believing that he or she has had access to a requested system resource: (i) cannot find out what the system resource is doing, (ii) cannot find out how the system resource is working and (iii) does not know that he/she does not know! That is, prevention of unauthorised access to computing and/or handling of information (i.e., data). ∎

The characterisation of security is rather abstract. As such it is really no good as an a priori design guide. That is, the characterisation gives no hints as how to implement a secure system. But, once a system is implemented, and claimed secure, the characterisation is useful as a guide on how to test for security!

Example 19.38 *Security Requirements: Timetable Security:* We continue Examples 19.14, 19.16, 19.34, and 19.35. Timetable users can be any airline client logging in as a user, and such (logged-in) users may inquire the timetable. The timetable machine shall be secure against timetable updates from any user. Airline staff shall be authorised to both update and inquire, in a same session. ▪

Example 19.39 *Security Requirements: A Hospital Information System:* General access to (including copying rights of) specially designated parts of a(ny) hospital patient's medical journals is granted, in principle, only to correspondingly specially designated hospital staff. In certain forms of (otherwise well-defined) emergency situations any hospital paramedic, nurse or medical doctor may "hit a panic button", getting access to a hospital patient's medical journal, but with only viewing, not copying rights. Such incidents shall be duly and properly recorded and reported, such that proper postprocessing (i.e., evaluation) of such "panic button" accesses can take take place. ▪

Robustness

Characterisation. A system is *robust* — in the context of dependability — if it retains its attributes after failure, and after maintenance. ▪

Thus a robust system is "stable" across failures and "across" possibly intervening "repairs" and "across" other forms of maintenance.

• • •

Fault Analysis: In pursuing the formulation of requirements for dependable systems it is often required that the requirements engineer perform what is called *fault analysis*. A particular approach is called *fault tree analysis*. Dependable systems development is worth a whole study in itself. So we cut short our mentioning of this very important subject by emphasising its importance and otherwise referring the reader to the relevant literature. A good introduction to the issues of safety analysis in the context of formal techniques is [292]. We strongly recommend this source — also for references to "the relevant literature".

19.6.5 Fault Tree Analysis

Source: Kirsten Mark Hansen

This example was kindly provided by Kirsten Mark Hansen. It is edited from Chap. 4 of her splendid PhD Thesis [141].

Fault tree analysis is one of the most widely used safety analysis techniques. It presumes a hazard analysis, which has revealed the catastrophic system failures [31]. For each system failure, it deduces the possible combinations of component failures which may cause this failure.

Fault tree analysis is a graphical technique, in which fault trees are drawn using a predefined set of symbols. The graphic representation may be appealing, but it also causes the fault trees to be big and unmanageable.

A fault tree analysis is closely related to a system model, as the different levels of system abstraction are reflected in the tree. The root corresponds to a system failure, and the immediate causes of this failure are deduced as logical combinations (conjunction and disjunction) of failures of the system components.

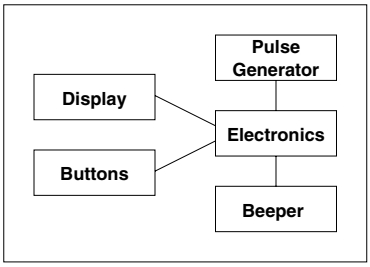


Fig. 19.3. Alarm clock

Figure 19.3 shows an alarm clock which is built from the components: A display, some buttons, a pulse generator, some electronics, and a beeper. A fault tree analysis of the failure of the alarm clock failing to activate the alarm is presented in Fig. 19.4. The causes of this failure may either be the beeper failing; the pulse generator not generating the right pulses; the electronics failing, either by not activating the beeper or by not registering the buttons pushed; or the buttons failing. We assume that the display has no impact on this failure. Each of the components may again be considered as a system consisting of components. The analysis stops when a component is considered to be atomic.

A minimal cut set of a fault tree is the smallest combination of component failures which, if they all occur, will cause the top event to occur. Smallest means that if just one component failure is missing from the cut set, then the

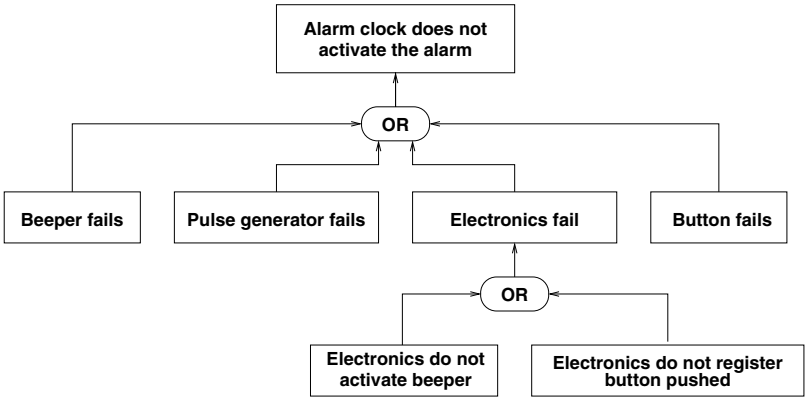


Fig. 19.4. Fault tree for an alarm clock

top event does not occur. The fault tree in Fig. 19.4 has five minimal cut sets, each containing a leaf as its only element. Two fault trees are defined to be equivalent if they have the same minimal cut sets.

A concept related to the minimal cut set is the minimal path set. A minimal path set is the smallest combination of primary events whose non-occurrence assures the non-occurrence of the top event. The fault tree in Fig. 19.4 has one minimal path set containing all the leaves of the tree.

As fault trees are used to analyse safety-critical systems for safety, it is important that they have an unambiguous semantics. We will later illustrate that often this is not the case. The aim of this chapter is therefore to assign a formal semantics to fault trees, and to illustrate how such a semantics may be used in the formulation of system safety requirements. The main reference in this chapter is the fault tree handbook [358], which has been used intensively in defining the syntax and the semantics of fault trees.

Some of the nodes of a fault tree are called events by safety analysts. In order to avoid confusion, we stress that we use the safety analysis meaning of the term event, namely the occurrence of a system state, rather than the computer science meaning of an event, namely a transition between two states.

Fault Tree Syntax

A fault tree analysis consists of building fault trees by connecting nodes from a predefined set of node symbols by directed edges. Edges are directed in the sense that for a given node the child nodes are called input nodes, and the father node is called the output node. The node symbols are divided into three groups: event symbols, gate symbols, and transfer symbols. We describe each of the groups separately.

Event Symbols

The event symbols are divided into primary event symbols and intermediate event symbols, where the primary event symbols are the leaves of the tree.

Primary events: The primary event symbols are shown in Fig. 19.5.

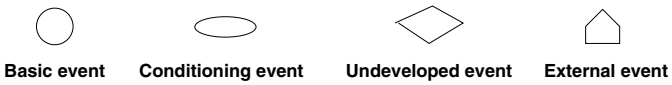


Fig. 19.5. Primary event symbols

- **Basic event:** A basic event contains an atomic component failure.
- **Conditioning event:** Conditioning events are most often used as input to PRIORITY AND and to INHIBIT gates. When used as input to a PRIORITY AND gate, the condition event is used to specify the order in which the input events must occur.
- **Undeveloped event:** An undeveloped event contains a non-atomic component failure. The fault tree is not developed further from this event due to lack of time, money, interest, etc. The component is not atomic, so it is possible later to develop the event further.
- **External event:** The content of an external event is not a failure, but something that is expected to occur in the system environment.

Intermediate events: The intermediate events consist only of one symbol, namely the intermediate event symbol, a rectangular box. Intermediate events cannot be found in the leaves of a fault tree.

Gate Symbols

Gate symbols designate Boolean combinators. They are shown in Fig. 19.6.

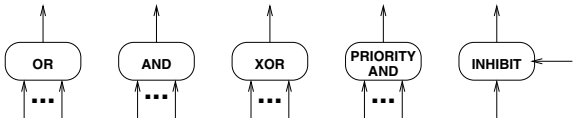


Fig. 19.6. Gate symbols

OR gate: The informal description of an OR gate is that the output event occurs when at least one of the input events occur. An OR gate may have any number of input events. Fig. 19.4 is an example of a fault tree with two OR gates.

AND gate: The informal description of an AND gate is that the output event occurs only when all the input events occur. An AND gate may have any number of input events. Fig. 19.7 is an example of a fault tree with an AND gate. This fault tree states that all brakes on a bike have failed, when both the foot brake "and" the hand brake have failed.

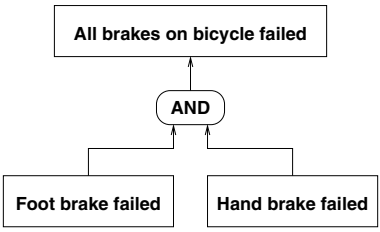


Fig. 19.7. Fault tree with AND gate

INHIBIT gate: An INHIBIT gate is a special case of an AND gate. An INHIBIT gate has one input event and one condition. The output event occurs when both the input event occurs and the condition is satisfied. In the fault tree in Fig. 19.8, the chemical reaction goes to completion when all reagents and the catalyst are present.

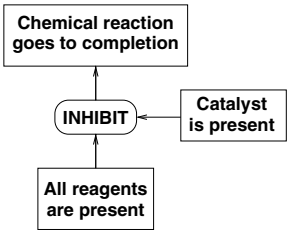


Fig. 19.8. Fault tree with INHIBIT gate

XOR (exclusive or) gate: The output event occurs only if exactly one of the input events occurs. If more than one of the input events occur, the output event does not occur. An XOR gate may have any number of input events. Fig. 19.9 shows a fault tree with an XOR gate. This fault tree states that a train is not at the platform, either if the train is ahead of the platform, or if it is behind the platform. Since the (specific) train cannot be at both places it is exactly at one or the other.

PRIORITY AND gate: The output event occurs only if all the input events occur, and if they occur in a left to right order. A PRIORITY AND gate may

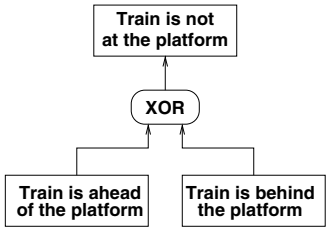


Fig. 19.9. Fault tree with XOR gate

have any number of input events. The fault tree in Fig. 19.10 states that the door is locked if the door is (first) closed and the key is (then) turned.

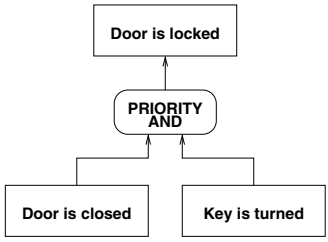


Fig. 19.10. Fault tree with PRIORITY AND gate

Fault Tree Semantics

In our attempt to give fault trees a formal semantics, we discovered that the accepted informal descriptions of fault tree gates are ambiguous, allowing several very different interpretations. For instance, the semantics of an AND gate is defined as [358]: "The output fault occurs only if all the input faults occur"; but what does this mean? Does it mean that all input faults have to occur at the same time, or does it mean that all input faults have to occur, but that they need not overlap in time? Does the output fault necessarily occur when the input faults occur? Clearly such uncertainty is not desirable when dealing with safety-critical systems. In this section we therefore give fault trees a formal semantics.

Primary Events

The first step in assigning a formal semantics to fault trees is to define a model of the system on which the fault tree analysis is performed. Assume that we have defined such a model and that it takes the form of system states evolving over time. (This "system states evolving over time" model is the basis for the

duration calculus [381,382]. We refer to Chap. 15, Vol. 2, for an introduction to the duration calculus.) Using this model, we interpret the leaves of a fault tree, i.e., the basic events, the undeveloped events, the conditioning events, and the external events as duration calculus formulas. Such a formula may *for instance* be:

- the constants *true*, *false*
- occurrence of a state P, i.e., $\lceil P \rceil$
- occurrence of a transition to state P, i.e., $\lceil \neg P \rceil ; \lceil P \rceil$
- lapse of a certain time, i.e., $\ell \geq (30 + \epsilon)$, or
- a limit of some duration, i.e., $\int P \leq 4 \times \epsilon$.

We consider the distinction between the different types of leaves to be pragmatic, describing why the fault tree has not been developed further from the that leaf, and therefore we make no distinction between the types of the leaves in the semantics.

Intermediate Events

The semantics of intermediate events is defined by the semantics of the leaves, edges, and gates in the subtrees in which the intermediate events are the roots. Intermediate events are merely names for the corresponding subtrees.

Edges

We now consider the meaning of the intermediate event, A, connected to an event, B, by an edge, see Fig. 19.11.

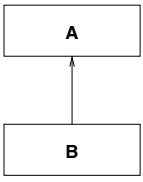


Fig. 19.11. Fault tree with no gates

Assume that the semantics of B is B. We then define the semantics of A to be

$$A = B,$$

i.e., as logical identity, meaning that the system failure A occurs when the failure B occurs. This semantics is pessimistic in the sense that it assumes that if something has a possibility of going wrong, then it does go wrong. Informal readings of fault trees often state that it is not mandatory that A holds when

B holds [353, 358], which is formalised as $A \Rightarrow B$. This semantics allows an optimistic interpretation of fault trees in the sense that a system failure may be avoided if the operator intervenes fast enough, has enough luck, etc. In our opinion, speed, luck, and the like should not be parameters in safety-critical systems, and we have therefore rejected this semantics. Another issue is whether A and B occur at the same time or if there is some delay from the occurrence of B to the occurrence of A. Often there will be such a delay, but we have refrained from modelling it, as this again would give the impression that once B has occurred there is a chance that A can be prevented.

Gates

We now consider the semantics of intermediate events connected to other events through gates.

OR: For the fault tree in Fig. 19.12 assume that the semantics of $B_1, \ldots, B_n$ is $B_1, \ldots, B_n$. We define the semantics of A to be

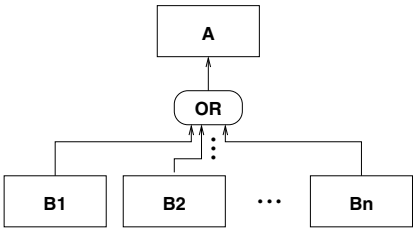


Fig. 19.12. Fault tree with OR gate

$$A = B_1 \vee \ldots \vee B_n,$$

i.e., A holds iff either B_1 or $\ldots$ or B_n holds. This interpretation shows that an OR gate introduces single point failure. The failure occurs if just one of the formulas holds.

AND: In the fault tree in Fig. 19.13 assume that the semantics of $B_1, \ldots, B_n$ is $B_1, \ldots, B_n$.

We then define the semantics of A to be

$$A = B_1 \wedge \ldots \wedge B_n,$$

i.e., A holds iff $B_1, \ldots, B_n$ hold simultaneously. We have considered a more liberal interpretation of AND gates in which B_1 to B_n need not hold simultaneously, namely $A = \Diamond B_1 \wedge \ldots \wedge \Diamond B_n$. This has been rejected since this formula "remembers any occurrence of a B_i", such that if B_2 becomes true 1 year after B_1, and B_3 becomes true 3 years after B_2, and $\ldots$, then A holds. This is clearly not the intended meaning of an AND gate.

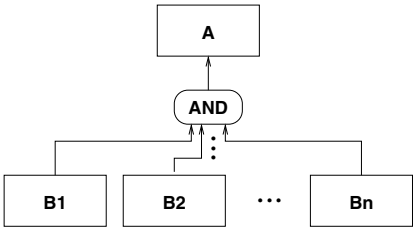


Fig. 19.13. Fault tree with AND gate

INHIBIT: We only consider INHIBIT gates in which the condition is *not* a probability statement. According to the fault tree handbook, [358], the fault

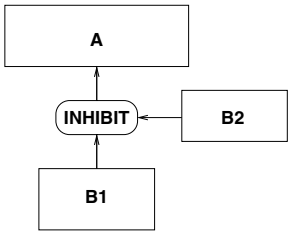


Fig. 19.14. Fault tree with INHIBIT gate

tree in Fig. 19.14 reads: "If the output A occurs then the input B_1 has occurred in the past while condition B_2 was true". We interpret this to be if A holds, then both B_1 and B_2 hold, i.e., as an AND gate with B_1 and B_2 as inputs. Thus the semantics of an INHIBIT gate is

$$A = B_1 \wedge B_2.$$

XOR: A fault tree with an XOR gate is given in Fig. 19.15 (left). According

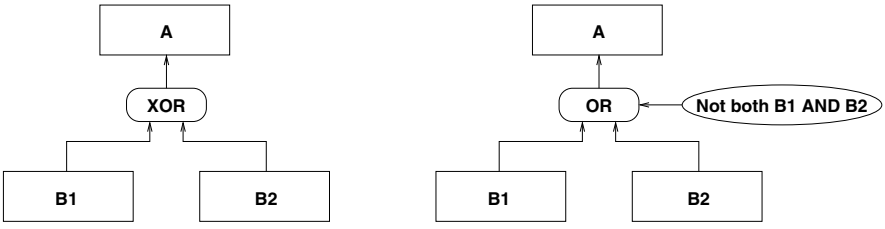


Fig. 19.15. Fault trees. Left with XOR gate. Right with OR gate and Condition

to the fault tree handbook, [358], this tree may be drawn as in the same figure to the right, in which "Not both B_1 AND B_2" is a necessary condition for the root formula to hold. As for the INHIBIT gate we interpret the condition "Not both B_1 AND B_2" as a leaf which should also hold. By interpreting "Not both B_1 AND B_2" as $\neg(B_1 \wedge B_2)$, we obtain the semantics

$$A = (B_1 \vee B_2) \wedge \neg(B_1 \wedge B_2)$$

which may be rewritten to

$$A = (B_1 \wedge \neg B_2) \vee (\neg B_1 \wedge B_2).$$

This generalises to

$$\begin{aligned} A = \; & (B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1})). \end{aligned}$$

PRIORITY AND: A fault tree with a PRIORITY AND gate is given in Fig. 19.16. The informal semantics states that the output event occurs if

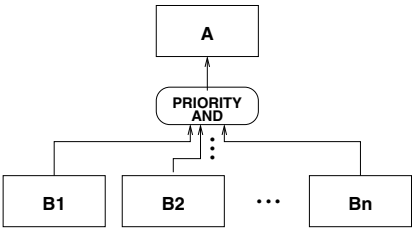


Fig. 19.16. Fault tree with PRIORITY AND gate

all the input events occur in a left to right order. Assuming that $B_1, \ldots, B_n$ have the semantics $B_1, \ldots, B_n$, we define the semantics of A to be

$$A = B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n) \ldots).$$

Refinement

As we saw in the beginning of this section, fault trees are often used to model system failures at different abstraction levels, Figs. 19.3 and 19.4.

If there is a shift in abstraction levels in a fault tree, we require that it is indicated by a dashed line connecting a root in one tree (concrete model)

to a leaf in another tree (abstract model) as in Fig. 19.17. (In that figure we have "abstracted" the Boolean combinators: BCx, BCy, BCz are either of OR, AND, PRIORITY AND, INHIBIT or XOR.)

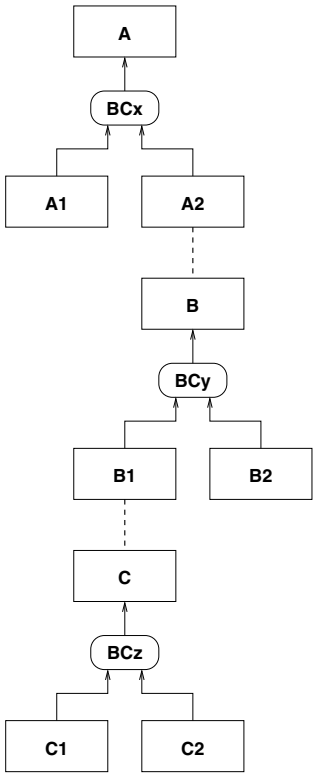


Fig. 19.17. Fault tree with three abstraction levels

We consider such a dashed line to connect two fault trees, where each of the fault trees is defined in one system model. For each of the fault trees, the semantics of the tree is defined as described previously. The dashed line indicates a refinement relation between the systems for which the fault tree analysis is performed. Consider the simple fault tree in Fig. 19.18 in which A has the semantics A and is defined by the state functions Var_a, and B has the semantics B and is defined by the state functions Var_b.

Assume that Var_a is a subset of Var_b. As a fault tree describes the undesired system behaviours, i.e., $\neg A$ for the abstract system, and $\neg B$ for the concrete system, the refinement relation between the two systems is given by

$$\neg B \Rightarrow \neg A$$

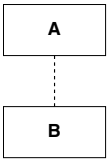


Fig. 19.18. Simple fault tree with refinement

where $\neg A$ is interpreted over the domain Var_b. It is equivalent to

$$A \Rightarrow B.$$

If the state functions of the concrete system, B, relate to the state functions of the abstract system, A, through a transformation ϕ, then the refinement relation under transformation is interpreted over $Var_a \cup Var_b$ and is given by

$$\phi \wedge \neg B \Rightarrow \neg A$$

which is equivalent to

$$\phi \wedge A \Rightarrow B.$$

In Fig. 19.17, assume that A_1 has the semantics A_1, A_2 has the semantics A_2, B_1 has the semantics B_1, etc., then it may be deduced from the semantics of fault trees that A has the semantics $A_1 \vee A_2$, B has the semantics $B_1 \wedge B_2$ and C has the semantics $C_1 \vee C_2$. Further assume that the fault tree containing the A's is defined in model 1, which has the state functions Var_a; the fault tree containing the B's is defined in model 2, which has the state functions Var_b; and the fault tree containing the C's is defined in model 3, which has the state functions Var_c. Further assume that Var_b relates to Var_a through the transformation ϕ, and that Var_b is a subset of Var_c. The proof obligations that arise from the fault tree are therefore

$$\phi \wedge A_2 \Rightarrow B_1 \wedge B_2$$

which is interpreted over $Var_a \cup Var_b$, and

$$B_1 \Rightarrow C_1 \vee C_2$$

in which B_1 is interpreted over Var_c.

In program development the chain of refinements is from *true* towards $false$. For fault trees the refinements from the top towards the bottom are from $false$ towards true. The reason for this is that fault trees specify the undesired system states, whereas program development specifies the desired system states.

Deriving Safety Requirements

Traditionally, fault trees are used to analyse existing system designs with regard to safety. Instead of first developing a design, and then performing a safety analysis, we propose that the design and the safety analysis should be developed concurrently, thereby making it possible to let the fault tree analysis influence the design. In order to do this, the fault tree analysis and the system design must at each abstraction level use the same system model. Given a common model, the system safety requirements may be deduced from the fault tree analysis. Safety requirements derived in this way can be used during system development in order to validate the design, but they can also be used in a constructive way by influencing the design. We illustrate this below.

For each fault tree in which the root is interpreted as S, the system should be designed such that S never occurs, i.e., the safety commitment which the system should implement is

$$\Box\neg S.$$

If we have n fault trees in which the roots are interpreted as $S_1, \ldots, S_n$, the safety commitment which may be deduced from these fault trees is

$$\Box\neg S_1 \wedge \ldots \wedge \Box\neg S_n,$$

i.e., the system should ensure that no top event in any fault tree ever holds. This corresponds to combining the trees by an OR gate.

Deriving Component Requirements

Assume that we have a fault tree like the one in Fig. 19.11, and that the safety commitment is $\Box\neg A$. As the fault tree has the semantics $A = B$, $\Box\neg A$ must be implemented by implementing $\Box\neg B$. If the fault tree contains gates, the derived specifications depend on the types of the gates.

OR gates: The fault tree in Fig. 19.12 has the semantics $A = B_1 \vee \ldots \vee B_n$. In order to make the system satisfy the safety commitment $\Box\neg A$, we must implement

$$\Box\neg(B_1 \vee \ldots \vee B_n)$$

or equivalently

$$\Box\neg B_1 \wedge \ldots \wedge \Box\neg B_n.$$

This formula expresses that the system only satisfies its safety commitments if all its components satisfy their local safety commitments. Now suppose that the designer cannot control the first component, i.e., it is outside the scope of the design of that component whether it satisfies B_1 or not. Making the

safe choice of B_1 being *true* causes $\Box\neg B_1$ to be *false*, which trivially implies that the safety commitment is violated. Making the tacit assumption that B_1 is *false* is a very poor judgment, which essentially ignores the results of the safety analysis. The only reasonable option is to weaken the specification. We *assume* that the behaviour of the first component never satisfies B_1, i.e., that $\Box\neg B_1$ is *true*. To make the design team aware of this assumption, we add it to the environment assumptions. So, if the design involved the assumptions Asm before this design step, we have assumptions $Asm \wedge \Box\neg B_1$ afterwards. The specification of the requirements $Asm \Rightarrow Com$ has thus been weakened, to $Asm \wedge \Box\neg B_1 \Rightarrow Com$, and the designer should alert the appropriate persons as to this change in assumptions. Many design errors are located on interfaces. The interface is made clearer and the likelihood of errors is reduced if one has an explicit list of assumptions and adds to this list as the system development progresses.

AND gates: Bear in mind that the fault tree in Fig. 19.13 has the semantics $A = B_1 \wedge B_2 \wedge \ldots \wedge B_n$ and assume that the safety commitment is $\Box\neg A$. This safety commitment corresponds to specifying that the components never satisfy their duration formulas at the same time, i.e.,

$$\Box\neg(B_1 \wedge B_2 \wedge \ldots \wedge B_n).$$

One way to implement this is to implement the stronger formula

$$\Box\neg B_1 \vee \Box\neg B_2 \vee \ldots \vee \Box\neg B_n,$$

i.e., to design at least one of the components such that it always satisfies its local safety commitment. Often, the designer does not control all the input components of an AND gate. For such components a safe approach is to assume the worst case, namely that the component is in a critical state and thereby contributes to violation of the safety commitment. Let us for instance assume in the case of the fault tree in Fig. 19.13 that the first component is uncontrollable. The worst case is that the component satisfies B_1, i.e., that

$$\Box\neg(true \wedge B_2 \wedge \ldots \wedge B_n)$$

meaning that the designer has to implement

$$\Box\neg(B_2 \wedge \ldots \wedge B_n).$$

If it is not possible to make such an implementation, a final solution is to assume that B_1 always is false, and then see to it that this is implemented in another component by adding it to the list of assumptions, i.e., if we had the assumptions Asm before this design step, we have the assumptions $Asm \wedge \Box\neg B_1$ afterwards. One should, at some point, arrive at a conjunction of B_i's which can be used in the design. Otherwise we must conclude that the system is inherently unsafe. If the design relies on the absence of only one B_i, it is a design which is vulnerable to single point failures.

INHIBIT gates: As the semantics of INHIBIT gates are the same as for AND gates, the derivations of safety requirements for INHIBIT gates are the same as for AND gates.

XOR gates: An event A which is output from an XOR gate which has $B_1, \ldots, B_n$ as input events has the semantics

$$\begin{aligned} A = \; & (B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1})). \end{aligned}$$

A safety commitment $\Box\neg A$ must be implemented by

$$\begin{aligned} \Box\neg \; & ((B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1}))) \end{aligned}$$

which is equivalent to

$$\begin{aligned} \Box \; & ((\neg B_1 \vee B_2 \vee \ldots \vee B_n) \\ & \wedge \\ & \vdots \\ & \wedge \\ & (\neg B_n \vee B_1 \vee \ldots \vee B_{n-1})). \end{aligned}$$

This means that the designer has to make the design such that for every observation interval either all the input events are false, or at least two of the input events are true at the same time, i.e.,

$$\Box(\text{All-false} \vee \text{Two-true})$$

where

$$\text{All-false} \equiv \; \neg(B_1 \vee \ldots \vee B_n),$$

$$\begin{aligned} \text{Two-true} \equiv \; & ((B_1 \wedge B_2) \vee \ldots \vee (B_1 \wedge B_n) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge B_1) \vee \ldots \vee (B_n \wedge B_{n-1})). \end{aligned}$$

Now assume that one of the components is uncontrollable, i.e., the designer cannot control whether, e.g., B_1 is true or not. If the Exclusive Or (XOR)

gate has more than two input events, then the design may be made such that two of the other input events are always true. If this is not possible (perhaps because the XOR gate only has two input events), the designer either has to assume that B_1 is false and then make the design such that the rest of the B's are always false, or assume that B_1 is true and then make the design such that one of the other input events is always true. In either case, he has to make the rest of the design team aware of the assumption by adding it to the list of assumptions about the environment. So, if the design involved the assumptions, Asm, before this design step, and if the designer assumes that B_1 is always true, then the assumptions are $Asm \wedge \Box B_1$ after this design step, and if he assumes that B_1 is always false, then the assumptions are $Asm \wedge \Box\neg B_1$. In principle the designer may also assume that whenever one of the B's which he can control is true then B_1 is also true, and whenever all the B's he can control are false, then B_1 is also false. As B_1 is implemented in another component than the rest of the B's, and as A occurs if the components are out of synchronization just once, we do not recommend this solution.

PRIORITY AND gates: The fault tree in Fig. 19.10 has the semantics $A = B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n)\ldots)$. If the safety commitment is $\Box\neg A$, the designer must implement

$$\Box\neg(B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n)\ldots)).$$

This may either be done by making the design such that the B_i's do not occur in the specified order or such that one of the B_i's does not occur at all, i.e.,

$$\Box\neg B_1 \vee \Box\neg B_2 \vee \ldots \vee \Box\neg B_n.$$

If one of the B_i's, e.g., B_1 is uncontrollable, the worst case is that it does not satisfy its local safety commitment, i.e., that B_1 is true. The designer therefore assume that B_1 is true and attempts to make the design such that

$$\Box\neg(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n)\ldots)$$

holds. If it is not possible to make such a design, the last opportunity is to assume that B_1 always is false, and then to assure that this is implemented in another component by adding it to the list of assumptions about the environment, i.e., the assumptions become $Asm \wedge \Box\neg B_1$.

Refinement

Assume that we have a fault tree in which an event A, with the semantics A, is refined by an event B, with the semantics B, see Fig. 19.18. Further, assume that the refinement relation has been verified, and that the safety commitment is $\Box\neg A$. As part of the refinement relation is $A \Rightarrow B$, then $\Box\neg A$ must be implemented by implementing $\Box\neg B$.

Conclusion

In this section we have given fault trees a duration calculus semantics, and we have defined how a fault tree analysis may be used to derive safety requirements, both for systems and for system components. The semantics is compositional such that the semantics of the root is expressed in terms of the leaves. The derivation of safety requirements follows the structure of the fault tree and results in safety requirements for the system's components. This derivation of safety requirements for components should stop when the deduced requirements may be implemented using well-established methods, e.g., formal program development techniques for software components.

As for all other techniques, this technique for deriving safety requirements is no better than the people who use it. An error in the fault tree analysis is reflected in the safety requirements, and the system failures for which a safety analysis has not been performed are not extracted as requirements. If, however, we compare this method to the existing ways of deriving safety requirements, namely by more or less structured brainstorming, we think that this method is an improvement.

In terms of safety requirements, a minimal cut set corresponds to the smallest set of components which, if they do not fulfill their safety requirements, will cause the system not to fulfill its safety requirements. If the minimal cut set only contains one component, then the system is vulnerable to single point failure.

A minimal path set corresponds to the smallest set of components which must fulfill their safety requirements in order that the system fulfill its safety requirements. If all components have to fulfill their safety requirements, i.e., the cardinality of the minimal path set equals the number of components, then the system is unsafe, as it may fail if just one of the components fails.

We have defined the semantics in duration calculus, but other temporal logics, like e.g., TLA$^+$ [209,210,239] and linear temporal logic [228–230], could also have been applied. The important thing is that the logic is capable of expressing both the semantics of the intermediate events, based on the structure of the fault tree, and the semantics of the leaves.

Fault trees are sometimes used in a probabilistic analysis of safety. We have not given semantics to fault trees with probabilistic figures, as this requires a deeper knowledge of stochastic processes than we have. The foundation for assigning a formal semantics to such trees has been established in [223], in which a probabilistic duration calculus based on discrete Markov chains [354] is defined and in [90] which defines a conversion algorithm from fault trees to Markov chains. The idea, in probabilistic duration calculus, is that, given an initial probability distribution, i.e., the probability that the system is initially in a state v, and a transition probability matrix, i.e., the probability that the system enters state u, given that the system is in state v, then it is possible to calculate the probability that the system is in a certain state at a discrete time t.

19.6.6 Maintenance Requirements

Characterisation. By *maintenance requirements* we understand a combination of requirements with respect to: (i) *adaptive maintenance*, (iii) *corrective maintenance*, (ii) *perfective maintenance*, (iv) *preventive maintenance* and (v) *extensional maintenance*. ∎

Maintenance of building, mechanical, electrotechnical and electronic artifacts — i.e., of artifacts based on the natural sciences — is based both on documents and on the presence of the physical artifacts. Maintenance of software is based just on software, that is, on all the documents (including tests) entailed by software. We refer to the very beginning of Sect. 1.2.4 for a proper definition of what we mean by software.

Adaptive Maintenance

Characterisation. By *adaptive maintenance* we understand such maintenance that changes a part of that software so as to also, or instead, fit to some other software, or some other hardware equipment (i.e., other software or hardware which provides new, respectively replacement, functions). ∎

Example 19.40 *Adaptive Maintenance Requirements: Timetable System:* The timetable system is expected to be implemented in terms of a number of components that implement respective domain and interface requirements, as well as some (other) machine requirements. The overall timetable system shall have these components connected, i.e., interfaced with one another — where they need to be interfaced — in such a way that any component can later be replaced by another component ostensibly delivering the same service, i.e., functionalities and behaviour. ∎

Corrective Maintenance

Characterisation. By *corrective maintenance* we understand such maintenance which corrects a software error. ∎

Example 19.41 *Corrective Maintenance Requirements: Timetable System:* Corrective maintenance shall be done remotely: from a developer site, via secure Internet connections. ∎

Perfective Maintenance

Characterisation. By *perfective maintenance* we understand such maintenance which helps improve (i.e., lower) the need for hardware (storage, time, equipment), as well as software. ∎

Example 19.42 *Perfective Maintenance Requirements: Timetable System:* The system shall be designed in such a way as to clearly be able to monitor the use of "scratch" (i.e., buffer) storage and compute time for any instance of any query command. ▪

Preventive Maintenance

Characterisation. By *preventive maintenance* we understand such maintenance which helps detect, i.e., forestall, future occurrence of software or hardware errors. ▪

Preventive maintenance — in connection with software — is usually mandated to take place at the conclusion of any of the other three forms of (software) maintenance.

Extensional Maintenance

Characterisation. By *extensional maintenance* we understand such maintenance which adds new functionalities to the software, i.e., which implements additional requirements. ▪

Example 19.43 *Extensional Maintenance Requirements: Timetable System:* Assume a release of a timetable software system to implement a requirements that, for example, expresses that shortest routes but not that fastest routes be found in response to a travel query. If a subsequent release of that software is now expected to also calculate fastest routes in response to a travel query, then we say that the implementation of that last requirements constitutes extensional maintenance. ▪

• • •

Whenever a maintenance job has been concluded, the software system is to undergo an extensive acceptance test: a predetermined, large set of (typically thousands of) test programs has to be successfully executed.

19.6.7 Platform Requirements

Characterisation. By a [computing] *platform* is here understood a combination of hardware and systems software — so equipped as to be able to execute the software being requirements prescribed — and 'more'. ▪

What the 'more' is should transpire from the next characterisations.

Characterisation. By *platform requirements* we mean a combination of the following: (i) *development platform requirements,* (ii) *execution platform requirements,* (iii) *maintenance platform requirements* and (iv) *demonstration platform requirements.* ∎

Example 19.44 *Platform Requirements: Space Satellite Software:* Elsewhere prescribed software for some space satellite function is to satisfy the following platform requirements: shall be developed on a `Sun` workstation under `Sun UNIX`, shall execute on the military `MI1750` hardware computer running its proprietary `MI1750 Operating System`, shall be maintained at the NASA Houston, TX installation of `MI1750 Emulating Sun Sparc Stations`, and shall be demonstrated on ordinary `Sun` workstations under `Sun UNIX`. ∎

Development Platform

Characterisation. *Development Platform Requirements:* By *development platform requirements* we shall understand such machine requirements which detail the specific software and hardware for the platform on which the software is to be developed. ∎

Execution Platform

Characterisation. *Execution Platform Requirements:* By *execution platform requirements* we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be executed. ∎

Maintenance Platform

Characterisation. *Maintenance Platform Requirements:* By *maintenance platform requirements* we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be maintained. ∎

Demonstration Platform

Characterisation. *Demonstration Platform Requirements:* By *demonstration platform requirements* we shall understand such machine requirements which detail the specific (other) software and hardware for the platform on which the software is to be demonstrated to the customer — say for acceptance tests, or for management demos, or for user training. ∎

Discussion

Example 19.44 is rather superficial. And we do not give examples for each of the specific four platforms. More realistic examples would go into rather extensive details, listing hardware and software product names, versions, releases, etc.

19.6.8 Documentation Requirements

We refer to Chap. 2 for a thorough treatment of the kind of documents that normally should result from a proper software development project. And we refer to overviews of these documents as they pertain to domain engineering (Sects. 8.9 and 16.3), requirements engineering (Sects. 17.6 and 24.3), and software design (Sect. 30.3).

Characterisation. By *documentation requirements* we mean requirements of any of the software documents that together make up software (cf. the very first part of Section 1.2.4): (i) not only *code* that may be the basis for executions by a computer, (ii) but also its full *development documentation*: (ii.1) the stages and steps of *application domain description*, (ii.2) the stages and steps of *requirements prescription*, and (ii.3) the stages and steps of *software design* prior to code, with all of the above including all *validation* and *verification* (incl., *test*) *documents*. In addition, as part of our wider concept of software, we also include (iii) a comprehensive collection of *supporting documents:* (iii.1) *training manuals*, (iii.2) *installation manuals*, (iii.3) *user manuals*, (iii.4) *maintenance manuals*, and (iii.5–6) *development and maintenance logbooks*. ∎

We do not attempt, in our characterisation, to detail what such documentation requirements could be. Such requirements could cover a spectrum from the simple presence, as a delivery, of specific ones, to detailed directions as to their contents, informal or formal.

19.6.9 Discussion: Machine Requirements

We have — at long last — ended an extensive enumeration, explication and, in many, but not all cases, exemplification, of machine requirements. When examples were left out it was because the reader should, by now, be able to easily conjure up such examples.

The enumeration is not claimed exhaustive. But, we think, it is rather representative. It is good enough to serve as a basis for professional software engineering. And it is better, by far, than what we have seen in "standard" software engineering textbooks.

19.7 Composition of Requirements Models

19.7.1 General

In Sects. 19.3.4 ($\mathcal{X}$ = BPR), 19.4.2 ($\mathcal{X}$ = Domain Requirements), 19.5.3 ($\mathcal{X}$ = Interface Requirements), and 19.6.2 ($\mathcal{X}$ = Machine Requirements) we have briefly mentioned the topic of *"$\mathcal{X}$ and the Requirements Document"*.

We shall remind the reader to review these four subsections. They tell you a lot about how to document the requirements, as basically a set of four more or less separate subdocuments, whether informally, as a narrative, or formally, as an annotated formal definition.

19.7.2 Collating Requirements Facet Prescriptions

Sections 11.10 and 11.10.1 have titles similar to this overall section and the present section. We have done so in order to remind the reader that to analyse requirements and to prescribe these is a bit also of an art. You are kindly asked to review Sect. 11.10.1 and to carry forward its message to requirements modelling.

19.8 Discussion: Requirements Facets

19.8.1 General

We have covered the three main facets of requirements models: domain requirements, interface requirements and machine requirements. The reader who studies this volume on the basis of emphasising the formal techniques will have noted that there were rather few, if any, formalised examples. This was especially true for the machine requirements.

This does not mean that one could not furnish such examples. We have chosen not to show such examples for three reasons: First, the examples would be somewhat long. Second, such examples have already been shown e.g., in Vol. 2, Chap. 15. But, more important, we still, as of 2006, lack appropriate formal techniques and tools. But we observe, today, steady and impressive progress in formal techniques and tools for expressing machine requirements.

19.8.2 Principles, Techniques and Tools

Principle. *Requirements Facets:* "Divide and Conquer": Adopt a "separation of concerns" principle; hence model domain, interface and machine requirements separately, as near so as possible. ▪

Techniques. *Requirements Facets:* The techniques fall, as usual, into two classes: the informal techniques, which cover all the so-far-covered informal techniques of rough-sketching, terminologisation and narration; and the formal techniques, which. likewise, cover all the so-far-covered formal techniques of formal abstraction and modelling. ▪

Tools. *Requirements Facets:* The tools, like the techniques, fall, as usual, into two classes: the informal tools, which include ordinary text-processing tools with extensive cross-referencing and database storage facilities; and the formal tools, which include all the ones ordinarily used in connection with formal specification: syntax editors, type checkers, verification, model checking and test tools, and so on. ▪

19.9 Bibliographical Notes

Section 19.3.1 relied almost exclusively on [139, 140, 176, 186]. Section 19.6.4 similarly relied almost exclusively on the delightful [287] and [212]. Section 19.6.5 is a mere editing of Chap. 4 of the splendid [141].

19.10 Exercises

19.10.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples. We refer also to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

19.10.2 The Exercises

The use of the term 'describe' means to rough sketch and/or terminologise, and to narrate. If you are studying this volume in its formal version, then the term describe additionally means formalise.

Exercises 19.4–19.6 relate to the special topic that you are expected to have chosen, and can be solved either informally or formally. Exercises 19.12–19.13 are expected to be solved formally.

Exercise 19.1 *An Incomplete Container Terminal Terminology.* We refer to Example 19.3. There are two versions of this exercise: an informal version and a formal version.

- *Informal version:* Please define all sorts, that is, the abstract types, and please state signatures of all functions mentioned in Example 19.3. Then rephrase a selection of some 10 terms.

- *Formal version:* First, solve the above *informal version* exercise. Then, formalise the chosen selection of terms.

Exercise 19.2 *Domain Instantiation: Local Regional Railway Nets.* We refer to Example 19.15. Please read that example carefully. The problem is to formalise that example's description of a simple railway net. We ask for a solution which simply takes the railway net formalisations shown in Vol. 2, Chaps. 2 and 10, and imposes further, constraining axioms.

Exercise 19.3 *Domain Requirements: Fitting.* We refer to Example 19.18. Please provide formal models of the domain, the two original requirements, and the 2+1 revised original + shared domain requirements outlined in Example 19.18.

Exercise 19.4 *Domain Requirements.* For the fixed topic, selected by you, you are to suggest some two to three distinct domain requirements. Outline (informally, and/or formally) for each of the distinct domain requirements how they are projected, and/or made more deterministic, and/or instantiated, and/or extended, and/or fitted (the latter with some other requirements that you have to postulate) with respect to your narrative domain description given earlier (as answers to Exercises 11.1–11.7.)

Exercise 19.5 *Interface Requirements.* For the fixed topic, selected by you, and for the domain requirements that you have established in Exercise 19.4, identify shared phenomena and shared concepts, and suggest at least four distinct interface requirements, one each from the possible set of six possibilities covered in Sects. 19.5.4–19.5.9.

Exercise 19.6 *Machine Requirements.* For the fixed topic, selected by you,, and for the domain requirements that you have established in Exercise 19.4, suggest at least one machine requirement from each of the five kinds outlined in Sects. 19.6.3–19.6.4 and 19.6.6–19.6.8 (performance, dependability, maintenance, platform and documentation, respectively).

Exercise 19.7 *Container Terminals: A Preliminary (Flat) Formal Domain Model.* We refer to Example 19.1. Based on what is described in the referenced example, please propose a formal model of container terminals. You may wish to formulate the solution in flat `RSL`, i.e., without the use of the **scheme, class** and **object** constructs of `RSL`. See Exercise 19.9.

Exercise 19.8 *Container Terminals: A Preliminary (Flat) Formal Requirements Model.* We refer to Example 19.2. Based on what is described in the referenced example, please propose a formal model of indicated requirements for software for ship container loading plans. If you chose to formulate the solution to Exercise 19.7 in flat `RSL`, i.e., without the use of the **scheme, class** and **object** constructs of `RSL`, then you may choose to do likewise for the present exercise. (See Exercise 19.10.)

Exercise 19.9 *Container Terminals: Modular Formal Domain Model.* We refer to Exercise 19.7. If you already expressed the solution to that exercise using the **scheme, class** and **object** constructs of RSL, then that could be a solution to the present exercise. Otherwise, please rephrase your solution to Exercise 19.7 using these modular constructs of RSL.

Exercise 19.10 *Container Terminals: Modular Formal Requirements Model.* We refer to Exercise 19.8. If you already expressed the solution to that exercise using the **scheme, class** and **object** constructs of RSL, then that could be a solution to the present exercise. Otherwise, please rephrase your solution to Exercise 19.8, based on your solution to Exercise 19.9, by, preferably, using the schema calculus constructs of extension (**with**), hiding (**hide**), etc., of RSL.

Exercise 19.11 *Rail Net and Unit Data Structure Initialisation.* We refer to Example 19.22. Please read that example carefully. Suggest a context in which the initialisation takes place: Awareness of the geography, through some cartographic and/or geodetic map representation. Then complete the narrative and formalise what is indicated in Example 19.22.

Exercise 19.12 *Rail Net and Unit Data Structure Refreshment.* We refer to Example 19.23. Please read that example carefully. Suggest a context in which the refreshment takes place: awareness of the geography, through some cartographic and/or geodetic map representation — as well as some already existing state. Then complete the narrative and formalise what is indicated in Example 19.23.

Exercise 19.13 *Banking Script Language.* We refer to Example 19.9 — and all of the examples referenced initially in Example 19.9. Redefine, as suggested there, the banking script language to allow such transactions as: (i) *merge two mortgage accounts,* (ii) *transfer money between accounts in two different banks,* (iii) *pay monthly and quarterly credit card bills,* (iv) *send and receive funds from stockbrokers,* etc.

Exercise 19.14 *Computational Data and Control Interface.* We refer to Example 19.24. You are to sketch, using RSL/CSP, a formalisation of that example in terms of two processes: the user and the referenced software package. By sketching we mean that basically only (i) the type of messages sent between these processes, and (ii) the RSL/CSP input/output clauses that outline the interaction, are defined. What leads the computation (based on the software package) to decide when and where to interact with the user is not to be specified, only that the interaction occurs.

Exercise 19.15 *A 24-hour Crane Behaviour.* We refer to Example 19.1. You are to come up with a rough sketch, a description and a prescription of what you can logically think of as a factual, respectively a desirable 24-hour behaviour of a ship/shore (i.e., quay) container crane.

Exercise 19.16 *A 24-hour Container Truck/Chassis Behaviour.* We refer to Example 19.1. You are to come up with a rough sketch, a description and a prescription of what you can logically think of as a factual, respectively a desirable 24-hour behaviour of a container truck/chassis.

20

Requirements Acquisition

- The **prerequisite** for studying this chapter is that you now know what should go into a requirements model: prescriptions of its (i) domain, (ii) interface and (iii) machine requirements, and, within these (i) of projections, determinations, instantiations, extensions and fittings, respectively, (ii) of shared data initialisation and refreshment, computational data and control, man-machine dialogues, man-machine physiological, and machine-machine dialogues, and (iii) of performance, dependability, maintenance, platform, and documentation requirements, respectively.
- The **aims** are that you will then know how to gather, to acquire, facts about the requirements, and to organise those facts for subsequent analysis.
- The **objective** is that you will, competently, lead and carry out thorough acquisition of requirements facts from requirements stakeholders.
- The **treatment** is pragmatic and systematic.

Throughout the acquisition process, recall:

The "Golden Rule" of Requirements Engineering

Prescribe only those requirements that can be objectively shown to hold of the designed software.

Stakeholders may not know whether a requirements that they express is objectively demonstrable or implementable. It is for the requirements engineer to guide the stakeholders on these matters.

20.1 Requirements Acquisition Versus Domain Models

Software development was in the past based only on requirements. There was then no need, in a textbook on software engineering, for a prior chapter on domain requirements facets. But there would be a need for a chapter on

requirements acquisition. Such a chapter on requirements acquisition would be very much similar to the chapter, Chap. 12, on domain acquisition. If a software developer had no domain description to build on she would need to do requirements acquisition following basically the principles and techniques of Chap. 12. But that chapter would have to be rewritten: wherever the term domain (etc.) appears in Chap. 12 a reformulation in terms of requirements (etc.) would have to appear. And this rewriting would, in numerous cases, appeal to the domain. But those more or less implicit references to the domain would only be to a small part of the domain, namely those which relate very specifically to the requirements being acquired.

The present chapter will steer a course somewhere between an "old-fashioned" requirements acquisition chapter (one that assumed that there was no knowledge of the domain) and a "modern" requirements acquisition chapter (i.e., one that assumes that there is a full domain description). Thus this chapter is formulated such that readers can read it without having read Chap. 12 (and then skipping Sect. 20.2). Readers with a prior knowledge of domain engineering can skip smaller parts of some examples.

20.2 Domain Model-Based Requirements Acquisition

To read this section we assume that you have studied Part IV, domain engineering.

20.2.1 Domain Requirements Acquisition, a Preview

Assumptions

A major part of requirements acquisition is based on, that is, assumes that both the client stakeholders and the requirements engineer have carefully studied the description of the (application) domain in question.

Domain Requirements Acquisition, Basic Steps

Characterisation. By *domain requirements acquisition* we mean a process where all decisions with respect to domain requirements are based on both the client stakeholders and the requirements engineer carefully and together going through that domain description. The *domain requirements acquisition* process includes that questions are asked of every described domain entity, function, event and behaviour: *should this phenomenon and/or concept "somehow" be part of the requirements?* And if so *how should the "somehow" be manifested?* The process further includes that answers to these questions are recorded. Thus the domain requirements acquisition is based on the domain requirements facets of *projection, determination, instantiation, extension* and *fitting*. ∎

20.2.2 Remaining Requirements Acquisition, a Preview

Assumptions

A remaining part of the requirements acquisition assumes that the requirements engineer and the client stakeholders "somehow" decide on possible platforms on which to implement the machine. This somehow is guided by possibly existing or potential client platforms, by economical and human factors considerations, by possible deviation in development costs among platforms, and so on. A remaining part of the requirements acquisition is based on, that is, assumes that the client stakeholders and the requirements engineer roughly decide on the possible platforms on which to implement the machine, and that the requirements engineer informs the client stakeholders of the technological possibilities offered by those platforms. By a platform we mean the hardware and its systems and other software packages by means of which the client and the software developers agree to implement the required software.

Machine Requirements Acquisition, Basic Steps

Characterisation. By *machine requirements acquisition* we mean a process where the requirements engineer works together with client stakeholders and examines those possible issues of *performance, dependability, maintenance, platform* and *documentation* that spring from considerations of domain requirements, and poses the questions *which degrees of performance and dependability, which kinds of maintenance, what kind of platform* and *which kinds of documentation* are expected from the required software and records, as machine requirements prescription units, client answers to these. ∎

Interface Requirements Acquisition, Basic Steps

Characterisation. By *interface requirements acquisition* we mean a process which, based on the domain requirements acquisition identifies all shared phenomena, and for each of these goes through the basic concepts of *shared data initialisation requirements, shared data refreshment requirements, computational data and control requirements, man-machine dialogue requirements, man-machine physiological interface requirements* and *machine-machine dialogue requirements,* and poses the questions *which are the shared data initialisation and refreshment requirements, which are the computational data and control requirements,* and *which are the man-machine and the machine-machine dialogue requirements* and *which are the man-machine physiological interface requirements*, and records, as interface requirements prescription units, client answers to these. ∎

20.2.3 Further Issues

Some of the concepts, principles and techniques of requirements acquisition are near-identical to those of domain acquisition. We shall anyway "repeat", in edited form, material from the chapter on domain acquisition, Chap 12, however with the term domain replaced by the term requirements, and with many subsidiary terms of domain acquisition replaced by those of requirements acquisition. We kindly expect the reader to carefully compare the two sets of terms and development processes.

20.3 Overview of Concepts

Characterisation. By *requirements acquisition* we shall understand the process of obtaining prescriptions of the requirements, that is, of *eliciting* (i.e., capturing) these from requirements stakeholders, of writing them down and of roughly structuring (i.e., roughly organising, roughly classifying) these prescriptions. ▪

The term *elicitation* is thus not synonymous with *acquisition*. But stands for part of the acquisition process. The *structuring* (organising or *classifying*) is meant as not being based on any serious, nontrivial analysis — that comes later! It is just an *indexing* for purposes of making easy some process of *search* for *requirements prescription units* having specified *properties*.

Acquisition involves *stakeholders*, and usually many of them. And acquisition involves writing something down: *requirements prescription units* in the form of *rough sketches*.

Elicitation is difficult. So we shall further investigate that concept. But first let us examine what it is we have to elicit, to extract, to capture, namely *requirements*.

20.3.1 Requirements

Characterisation. By *requirements* we shall understand that which "expresses" *what* is expected from the desired computing system, i.e., the machine, that is, the hardware plus the software. ▪

The term expresses is crucial. How the expressions are formulated, whether stored as mental images, or verbalised amongst stakeholders, or written down in some form or another, is crucial. In this and the next sections we shall be concerned about "forms of written expression".

20.3.2 Elicitation of Requirements

Characterisation. By *elicitation of requirements* we shall understand a multifaceted process: (i) of *reading about* requirements posed to previous computing systems of the same domain; (ii) of *talking with*, i.e., *interviewing*, requirements stakeholders; (iii) and of yourself, or stakeholders, *recording* (possibly aided by *questionnaires*) your/their conception — i.e., perspective, views — of the requirements. That is, we record the properties of the entities, the functions, the behaviours, including the events, of the machine to be designed — where these entities, functions, events, behaviours are now *conceptual*, i.e., are intellectualised. ∎

20.3.3 Recording Requirements

Characterisation. By *recording requirements* we shall understand a not necessarily systematic, coherent and complete writing down of bits and pieces of rough-sketch descriptions of one or another requirements, such that each of these bits and pieces forms a *unit of requirements prescription*, which is then suitably *indexed* (i.e., *classified*, categorised, named). ∎

Requirements Index

Roughly, an *index* of a *unit of prescription* is a marking as to one or more categories of the name of the requirements being recorded, and/or the names of the stakeholders, and/or a category name for the stakeholder group, and/or the type of the concept (data, function, event, process), and/or the type of the requirements facets. Section 20.3.4 will elaborate further on requirements indices.

Requirements Type

By the "type of a requirements facet" we mean whether it is a domain, an interface or a machine requirement.

- If it is a domain requirements, then by the type of the requirements facet we further mean whether it is a projection, a determination, an instantiation, an extension or a fitting requirement.
- If it is an interface requirements, then by the type of the requirements facet we further mean whether it is a shared data intialisation, a shared data refreshment, a computational data and control, a man-machine dialogue, a man-machine physiological or a machine-machine dialogue requirement.
- If it is a machine requirements, then by the type of the requirements facet we further mean whether it is a performance, a dependability, a maintenance, a platform or a documentation requirement.

Requirements Units

Characterisation. By *unit of requirements prescription* we shall understand some informal or formal, usually rough-sketch text which starts prescribing something, or which adds to a prescription of something, with the unit prescription being annotated by some initial classification, such that the unit prescription forms a whole (e.g., a complete sentence). ▪

A unit of prescription is, deliberately, a loose notion. It reflects the nature of the elicitation process. This process is exploratory, even experimental: The persons involved in the elicitation process, normally, at the outset of elicitation, do not know where the elicitation process will take them, that is, what will evolve! Eventually, a large body of units of requirements prescriptions will evolve.

Example 20.1 *Units of Requirements Prescriptions:* We give three examples of units:

(i) All railway lines shall be represented within the machine. *Initial (rough) classification:* stakeholder: *Mr. AA, railway signalling officer;* concept type: *entity;* facet: *projection;* concept names: *line.*

(ii) The machine shall be able to support the following operations involving simple demand/deposit bank accounts: opening, deposit, withdrawal, transfer, obtaining a statement (of past operations), and closing. *Initial (rough) classification:* stakeholder: *Ms. BB, bank teller;* primary concept types: *entity;* secondary concept types: *functions;* facet: *determination* (for primary and secondary concept); primary concept name: *demand/deposit bank account;* secondary concept names: *demand/deposit account, open, deposit, withdrawal, transfer, statement, close.*

(iii) The machine shall support airline timetables. *Initial (rough) classification:* stakeholder: *Mr. CC, pilot (captain);* concept type: *entity;* facet: *projection;* concept names: *airline timetable.* ▪

We observe, from the above examples, that often the unit text is (far) shorter than the classification. And that the classifications need to be refined into primary and secondary concept types, and facets.

We refrain from detailing possible forms of units and their classifications. Such a detailing would be appropriate for a specific instantiation of an acquisition process, typically one for which specific tools are then provided. Here it suffices to indicate the issues involved.

20.3.4 Indexing Requirements Prescription Sketches

Characterisation. By an *index* we shall understand a naming of concept names; a stakeholder group and one or more persons of that group; the kind of primary and secondary concepts: entities, functions, events, or behaviours; relevant type of the requirements facet. ▪

Indexing is basically an informal endeavour which is carried out in order to facilitate search for some properties as to the name of the entity, function, or process (including event); or as to the stakeholder group or person; or as to the attributes of the entity, function, event, or behaviour; or as to the facet type. That is, we envisage that the requirements engineer, in the analysis process, may wish to review several requirements prescription units with respect to some (logical) criteria that involve one or more of the above-listed classification categories.

Characterisation. By *indexing requirements sketches* we shall understand a process of equipping, i.e., of annotating, requirements prescription units, with one or more classification indexes, where these distinct indexes cover concept names, stakeholder kinds and person names, attributes (type and range of values of the) entity, function, or process (including event) kind, and type of requirements facets. ∎

20.4 The Acquisition Process

The acquisition process now, typically, proceeds, in one or (usually) more rounds (i.e., cycles) of each of the steps as now listed:

(i) *Review of stakeholder index, etc.:* (i.1) Is such a listing of all relevant stakeholders established? (i.2) If so, is it adequate? (i.3) If not, then establish it, and review. (i.4) Establishment of contact to, i.e., liaison with, identified requirements stakeholder groups and persons.

(ii) *Study of domain documents:* (ii.1) Are such documents established? (ii.2) If so, read them, and ask: are they adequate? (ii.3) If not, then augment, so as to become adequate. (ii.4) Make sure you (and clients) have understood the domain description documents well enough to serve as the basis for requirements questionnaires and development of domain requirements.

(iii) *Study of requirements documents:* (iii.1) retrieval of existing requirements documents (concerning same domain applications); (iii.2) evaluation of the relevance of such documents; (iii.3) reading of relevant such documents; and (iii.4) preparation for the recording of requirements prescription units (see below, item (vi)).

(iv) *Casual talks with stakeholders:* (iv.1) Initial, loosening-up talks, i.e., chats, in person, with stakeholders (iv.2) for the purposes of establishing rapport, i.e., confidence, trust, (iv.3) including preparation for the recording of requirements prescription units (see item (vi) below).

(v) *Systematic, questionnaire-based interviews of stakeholders:* (v.1) Formulation and printing of questionnaires (see below); (v.2) distribution, or personal handing out, in connection with casual talks, of questionnaires; (v.3) gathering of more or less completed questionnaires; (v.4) and preparation for the recording of requirements prescription units (see next item).

(vi) *Recording of requirements prescription units:* (vi.1) There is the recording, on paper, or electronically, done by either the interviewed and questioned stakeholders, or by the (requirements engineer) interviewers; (vi.2) and there is the result, on paper, or on some storage medium, of this recording: a document consisting of one or more requirements prescription units.

(vii) *Classification of requirements prescription units:* (vii.1) Each requirements prescription unit is then briefly examined, individually, (vii.2) i.e., separately from the examination of other requirements prescription units, (vii.3) and to each is affixed as many of the requirements prescription index unit attributes as are relevant and as can be ascertained.

(viii) *Review of requirements acquisition process:* (viii.1) Once, what is believed to be, at some point in time, all requirements prescription units have been received and indexed, (viii.2) they are examined as to whether those that have been elicited form a necessary and sufficient collection, (viii.3) whether some may need to be rejected, (viii.4) or whether more may be needed. This review process takes into account such issues as: Is the collection of requirements prescription units gathered representative (have all selected stakeholder groups replied with similar or uneven care)? Are the requirements prescription unit texts understandable, etc.? The review is not analysing the collection for consistency, conflicts or completeness. Such an analysis follows the requirements acquisition process.

We will now treat several of the above steps in some detail.

20.4.1 Stakeholder Liaison

We remind the reader of Sect. 9.3: stakeholder perspectives, and Sect. 12.2.1: stakeholder liaison. We can thus assume that a list of all possibly relevant stakeholders has been established and reviewed by all parties to a contract of development of a requirements prescription. Such a list was already established if a prior phase involved the engineering of domain models. And such a list can serve as a basis for the new list. We expect, normally, a requirements stakeholder list to be a subset of the domain stakeholder list, omitting just a few stakeholders — perhaps.

Now comes the effectuation, the liaison and interaction with what is considered appropriate, identified members of those requirements stakeholder groups that are deemed relevant for a successful completion of that development. The contract must secure timely availability of both the requirements development engineers and the identified requirements stakeholders. Let us refer to the latter as the requirements stakeholder representatives. A pair of managers, from, respectively, the requirements development engineers and the requirements stakeholders, must ensure an ongoing, free and timely use of the time resources of the requirements development engineers and the identified requirements stakeholders. It is that assurance we could call the requirements stakeholder liaison.

20.4.2 Elicitation Studies

Well-nigh every application has its existing requirements literature, one that in *one form or another more or less* explicitly hints at requirements prescriptions for software for *one or another* segment of the application area. Please observe the *italicised* hedges. By this we mean that it is not always that obvious to find good, written existing requirements prescriptions for software for every application — even within well-established (i.e., reasonably mature) software houses. To give the reader an idea of possible requirements prescription sources, besides the in-house, internal sources, we refer to the list of examples of external, organisational sources given in Sect. 12.2.2.

20.4.3 Elicitation Interviews

The purpose of elicitation talks is to establish rapport and build confidence between each requirements stakeholder group and the requirements engineers assigned to capture requirements knowledge from respective requirements stakeholder group members. Instead we shall, in this section, more closely examine the processes of interviews and their follow-up.

To repeat, the requirements engineer embarks on interviews by first preparing a customised questionnaire, one that is "fitted" to the specific domain at hand. The requirements engineer meets one or more representatives of the designated requirements stakeholder group in order to introduce them to the questionnaire. This introduction ("familiarisation") is done by "walking" through the questionnaire with these representatives, explaining terms to them and by answering questions that might arise during this interview. Then these representatives are left to fill in their own copy of an identical questionnaire, either individually, or as a group. Members of the group are encouraged, before filling out the questionnaire, to contact their requirements engineer for resolution of any "metaquestions", that is, questions about how to interpret the questions in the questionnaire. Finally, the questionnaire is believed completed and is returned to the group's requirements engineer.

20.4.4 Elicitation Questionnaires

But how is such a questionnaire formulated? And: Is there only one round of questionnaire interviews? We can right away suggest that there should be as many rounds of questionnaire interviews as the requirements engineer thinks necessary to cover the various requirements stakeholder groups' perspectives in necessary and sufficient depth. That is: We can usually not plan for how many rounds are needed. For an unfamiliar application target, one that has not been requirements prescribed before, requirements acquisition is a research undertaking — and for such research we cannot beforehand determine the number of "rounds" needed. But we can now give some general guidelines for how to formulate a questionnaire, and, after that, we will give an example of such a questionnaire.

General Guidelines: Questionnaire Structure and Contents

We know, now, what a requirements prescription should look like, what it should contain. Namely, at the least, an enumeration of requirements stakeholders: their names and categories. It should contain a prescription, within the chosen span and scope, of requirements within applicable *facets:*

1. business process reengineering:
   (a) support technology review and replacement,
   (b) management and organisation reengineering,
   (c) rules and regulation reengineering,
   (d) human behaviour reengineering, and
   (e) script reengineering;
2. domain requirements:
   (a) projection,
   (b) determination,
   (c) instantiation,
   (d) extension, and
   (e) fitting;
3. interface requirements:
   (a) shared data initialisation,
   (b) shared data refreshment,
   (c) computational data and control prompts and input,
   (d) man-machine dialogue,
   (e) man-machine physiological interface, and
   (f) machine-machine dialogue; and
4. machine requirements:
   (a) performance expectations (storage, time, and equipment),
   (b) dependability expectations (accessibility, availability, integrity, reliability, safety, and security),
   (c) maintenance (adaptive, corrective, perfective, preventive and extensional),
   (d) platforms (development, execution, maintenance, and demonstration),
   (e) documentation, etc.

That is, a set of requirements prescription units which, for each facet, attribute, and "thing", provides the type and some details of the concept, and specifically of what *kind* the concept is, entity, function, event or behaviour. So our questions should be such as to tempt the stakeholders to convey these *facet* kinds to the requirements engineer.

Special Guidelines: Questionnaire Structure and Contents

When outlining the questions concerning requirements facets, the requirements questionnaire must be tailored — "tuned" — with examples that are

pertinent to the domain. In our next example we shall indicate that tuning with *italic font*.

Example 20.2 *An Example Questionnaire:* We show, over the next pages, a long example of a questionnaire directed at healthcare workers (nurses, porters, medical doctors, and the like) in a hospital.

(0) Dear stakeholder, you are please asked to "collect your thoughts", i.e., to concentrate on thinking — thoroughly, conceptually — about your work. (0.1) Not your work as it is, (0.2) but how you would like it to be, or to become,

(1) First we ask you (see the questions a little further down) to tell us about what we call the *entities* of your work, the things you can point to or the things that you can otherwise conceive.

> Typical entities of your work are such as (and now we enumerate a long list of such things) *people: patients, nurses, medical doctors, technicians, next-of-kin (of patients), etc.; materials: medicine, bandages, distilled water, etc.; tools: syringes, blood pressure meter, thermometer, etc.; documents: patient medical records, etc.; other facilities: wards (with beds, etc.), operating rooms (with their equipment), etc. And so forth.*

(1.1) Please list, by naming them, the entities that you meet in your daily work, whether frequently, sometimes or seldomly. (1.2) For each entity listed please describe its characteristics, its properties and its use — not whether it is a good entity or a bad one. (1.3) Now make up your mind, for each and every entity that you have identified, whether you expect the required computing system to represent, to reflect (i.e., to maintain and store) that entity.

(2) Then we ask you (see the questions a little further down) to tell us about what we call the *functions* of your work, the things you do with entities.

> Typical functions of your work are such as (and now we enumerate a long list of such things) *inject, with a syringe, penicillin into a patient; create, edit or copy a patient medical record; transfer a patient from a ward bed to a movable bed, etc.*

(2.1) Please list, by naming them, the functions that you perform in your daily work, whether frequently, sometimes or seldomly. (2.2) For each function listed please describe its characteristics: Which things enter into doing the function, and which things result from having performed the function. (2.3) Now make up your mind, for each and every function that you have identified, whether you expect the required computing system to support, in one form or another, that function, and how.

(3) Finally, we ask you (see the questions a little further down) to tell as about what we call the *behaviours* of your work, the procedural sequences of functions involving entities as well as *events* that trigger your work in different directions, etc.

A typical behaviour of your work is, for example, *the process of interviews, analyses, diagnostics, treatment (surgery, medication, rehabilitation, etc.), such as a patient undergoes during hospitalisation*.

(3.1) Please list, by naming them, the behaviours that you observe, or participate in, in your daily work, whether frequently, sometimes or seldomly. (3.2) For each behaviour listed please describe its characteristics. (3.3) Now make up your mind, for each and every behaviour that you have identified, whether you expect the required computing system to support, in one form or another, that behaviour, and how (this is an aspect of projection etc.).

(4) At this moment we show the requirements stakeholder one or another model of the stakeholders' domain. Maybe one presentation per requirements stakeholder group. And we then ask the questions:

(4.1) Which of those entities do you wish was "supported" by the machine (and how)? (4.2) Which of those functions do you wish was "supported" by the machine (and how)? (4.3) Which of those events do you wish was "supported" (i.e., observed or generated) by the machine (and how)? (4.4) Which of those behaviours do you wish was "supported" by the machine (and how)?

Obviously, the requirements engineer shall explain the concept of "support".

(5) We repeat item (4), but in a more systematic fashion, we ask:

(5.1) "Tuned" business process reengineering questions about: (5.1.1) Which new technologies shall be used (and how), and which old ones shall be phased out? (5.1.2) Which new management and organisational structures shall be used (and how), and which old ones shall be phased out? (5.1.3) Which new rules and regulations shall be used (and how), and which old ones shall be phased out? (5.1.4) Which changed human behaviours should we encourage be used, and which old ones shall be discouraged? (5.1.5) What scripting needs to be done (and how)?

The requirements engineer shall, obviously, explain the above terms.

(5.2) "Tuned" domain requirements questions about: (5.2.1) Which parts of the domain model shall be kept in, and which shall be omitted from the requirements? (5.2.2) Which nondeterminacies shall be kept, and which shall be made deterministic in the requirements (and how)? (5.2.3) Which generic ideas shall be kept, and which shall be instantiated in the requirements (and how)? (5.2.4) Which new (domain) functionalities shall extend the requirements (and how)? (5.2.5) To what other existing requirements shall the present ones be fitted (and how)?

The requirements engineer shall, obviously, explain the above terms.

(5.3) "Tuned" interface requirements questions about: (5.3.1) Which mass initialisation of data shall be done and how? (5.3.2) Which regular updates or refreshment of data shall be done, how regularly and how? (5.3.3) Which computational data and control inputs shall be provided and how? (5.3.4) How would you like the man-machine dialogue to proceed? Here the requirements engineer

may need give prototype examples. (5.3.5) What man-machine interface physiological "gadgets" shall be deployed and how? (5.3.6) Which machine-machine dialogues are necessary, and how should they proceed?

The requirements engineer shall, obviously, explain the above terms.

(5.4) "Tuned" machine requirements questions about: (5.4.1) Which performance figures must be achieved: response times, storage consumption, equipment facilities? (5.4.2) Which dependabilities (accessibility, availability, levels of integrity, reliability, safety, security) must be guaranteed (and to which degrees)? (5.4.3) Which issues of maintainability must be secured (adaptive, corrective, perfective, preventive and extensional)? (5.4.4) On which platforms shall the software be developed, on which shall it be executed, on which shall it be maintained, and on which, if applicable, shall it be demonstrated? (5.4.5) What documentation shall accompany the product?

The requirements engineer shall, obviously, explain the above terms. ∎

From application to application, and from case to case (i.e., contract to contract) the requirements engineers may extend or shorten the above exemplified questionnaire.

20.4.5 Elicitation Reports

The requirements engineer now assembles an elicitation report. Roughly it consists of all the reports received from solicited stakeholders, together with the requirements engineer's "quick" assessment of these reports: their trustworthiness, as well as their indexing. We expect an elicitation report to be both electronically available, and available in paper form.

20.5 Discussion

20.5.1 Concept and Process Review

We have outlined important concepts of and steps in the requirements acquisition process: the concepts of requirements prescription units and their indices; the identification of and liaison with requirements stakeholder groups and representatives; the concept and formulation of requirements acquisition questionnaires — as based on what we know of how a requirements model is structured and what it must contain. We also include in this list the process of requirements engineer studies of the domain; the process of requirements stakeholder interviews and requirements stakeholder completion of questionnaires — cum writing prescription units; the process of indexing; and the process of evaluating whether a requirements acquisition should be ended or continued.

20.5.2 Process Iteration

Requirements acquisition, especially for "untrodden" requirements, i.e., for "new", unfamiliar computing applications, is an art. It hinges on being research-oriented. As already indicated it will certainly evolve in iterations, and hopefully it can be quickly ascertained as to whether they are converging rapidly enough, or not — maybe even diverging.

20.5.3 Delineation: Acquisition and Analysis

We have decided here to present the requirements acquisition process as separate from the requirements analysis process. As two processes they interact: The latter may result in a call for resumption of the former. One might therefore, as the term 'resumption' indicates, treat them as coroutines.

To us, the essential difference is: the acquisition process is very stakeholder-intensive, and is centred around rough-sketching description units, whereas the analysis process is not stakeholder-intensive, and is centred around analysing prescription documents, and in concept formation.

20.5.4 Principles, Techniques and Tools

We summarise:

Principles. The principle of *requirements acquisition* is that of providing material which helps one to identify requirements, and to uncover inconsistencies and conflicts in human perceptions of the requirements. ▪

This, like domain acquisition, is truly a daunting task. It is, in its very nature, a task requiring training, more in science than in engineering. Commensurate with that, we formulate the next point.

Techniques. The techniques of *requirements acquisition*, besides all those clerical ones mentioned earlier in this chapter — (i) review of and liaising with stakeholders; (ii) reading domain descriptions; (iii) reading (other) requirements prescriptions; (iv) casual, explorative talks with stakeholders; (v) formulation of questionnaires, and questionnaire-based interviews with stakeholders; (vi) recording the results of these interviews; and (vii) collecting and indexing (i.e., classifying) requirements description units — are also the (viii) review of these indexed requirements description units, i.e., the "vetting" of their contents, so to speak. That is, (viii.1) ensuring that the set of indexed requirements description units is necessary and sufficient, and (viii.2) that the set is accurate, and properly reflects bona fide stakeholder perspectives. ▪

Tools. Since the requirements acquisition process is necessarily an informal one, the *tools* are: (a) human reading and interviews, (b) text processing

the recording of requirements description units (c) in a way that facilitates storage and retrieval, (d) and which allows for specialised programs (queries) to be (later) expressed and performed with the aim of facilitating requirements analysis. ▪

20.6 Exercises

20.6.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

20.6.2 The Exercises

Exercise 20.1 *Requirements Prescription Units.* For the fixed topic, selected by you, suggest some 12 requirements prescription units — such as might have been elicited from 4 different stakeholder groups.

Exercise 20.2 *Requirements Prescription Unit Indexes.* Based on your answer to Exercise 20.1, suggest (the possibility of) some indexing scheme.

Exercise 20.3 *Requirements Questionnaire.* For the fixed topic, selected by you,

1. suggest a short, precise narrative that can form the basis for a requirements questionnaire;
2. then formulate such a requirements questionnaire.

21

Requirements Analysis and Concept Formation

- The **prerequisite** for studying this chapter is that you have studied the chapters on stakeholders, requirement facets and requirements acquisition.
- The **aims** are to introduce concepts of 'concept formation', and to introduce concepts of 'requirements consistency,' 'completeness' and 'conflict'.
- The **objective** is to bring you further along the road to becoming a professional software engineer — being versatile in requirements engineering.
- The **treatment** is informal, yet systematic.

This chapter follows, almost line-by-line, Chap. 13: 'Domain Analysis and Concept Formation'.

21.1 Introduction

Characterisation. 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) the aim of evaluating whether a requirements can be objectively shown to hold, and if so what kinds of tests (etc.) ought be devised. ▪

In preparation for bringing in some concept formation examples, we first show some requirements prescription units:

Example 21.1 *Some Logistics System Requirements Prescription Units:* We bring in some (non-indexed) requirements prescription units gathered when requirements determining a freight logistics application:

"Freight delivered to trucking depots, railway freight terminals, harbours, and air cargo centres, shall be recorded, by the machine, as so delivered."

"Freight transported by trucks, freight trains, ships, and aircraft, shall be recorded, by the machine, as being so deliverable, as being so delivered, and, subsequently, as having been delivered."

"Trucks moving along highways, trains along railway lines, ships along sea lanes, and aircraft along air corridors shall be so recorded, as moving, etc."

"Freight transferred between trucks, trains, ships and aircraft at trucking depots, railway freight terminals, harbours, and air cargo centres, and between trucks, trains, ships and aircraft at such places where trucking depots coincide with railway freight stations, harbours and air cargo centres, shall be recorded, as about to be so transferred, as being "in the midst" of being so transferred, and, subsequently, as having been transferred".

And:

"Freight picked up from trucking depots, railway freight terminals, harbours, and air cargo centres, shall be so recorded: As available for being picked up, as being picked up, and, respectively, as having been picked up." ▪

Characterisation. By *machine concept formation* we understand the abstraction of requirements concepts — hinted at by requirements acquisition prescription units — into machine concepts. ▪

Whereas requirements prescription units may refer to categories of immediately instantiable, that is designatable values, concepts usually refer to types of such values.

Example 21.2 *A Logistics System Analysis and Concept Formation:* We analyse and form concepts from the above exemplified prescription units.

Trucking depots, railway freight terminals, harbours, and air cargo centres shall all be represented as hubs.

Trucks, trains, ships and aircraft shall all be represented as conveyor vehicles.

Highways, railway lines, sea lanes, and air corridors shall all be represented as routes. ▪

Concept formation usually leads to much simpler prescriptions.

Example 21.3 *A Partial Logistics System Narrative:*

Freight delivered to hubs shall be so registered by the machine.

Freight transported by conveyor vehicles shall be so registered by the machine.

Trucks moving along routes shall be so registered by the machine.

Freight transferred from one conveyor vehicle to another conveyor vehicle at hubs shall be so registered by the machine.

And:

Freight picked up at hubs shall be so registered by the machine. ▪

We now deal in more detail with two main aims of requirements analysis: Concept formation, and consistency, conflict and completeness analysis.

21.2 Concept Formation

We refer to Sect. 13.2. In that section we treated the notion of domain concept formation — and in more detail than below.

Concept formation is for the experienced, and it is an art — but some principles and techniques can be taught and learned. Principles and techniques of abstraction apply here — to their fullest — and form the major principles and techniques used in concept formation. As we shall have covered these principles and techniques of abstraction elsewhere[1] we shall not cover abstraction much here, but say: In reading requirements prescriptions we shall seek out texts wherever a concept can be abstracted (into a [further abstracted] concept), or two or more concepts can be "merged", i.e., abstracted into one concept.

Each abstracted concept need be carefully defined. Concept formation (i.e., identification) then necessitates a rewriting of requirements prescriptive text units into narratives where the concepts replace those which they abstract, as well as the insertion of the defined concepts into a terminology.

21.3 Consistencies, Conflicts, and Completeness

Conflicts are one form of inconsistencies. "Remaining" inconsistencies can be resolved between requirements stakeholders — as aided by the requirements engineers. Completeness is a relative issue. We shall cover these three ideas now.

21.3.1 Inconsistencies

Characterisation. By *inconsistency* of a *requirements prescription* we shall understand some pairs (or more) of texts where one text prescribes one (set of) property (properties), while another text (of the pair or more) prescribes (prescribe) an "opposite" property (set of properties), that is: Property P and property $\mathsf{not}\ P$. ∎

Example 21.4 *A Requirements Prescription Inconsistency:* The following two non-indexed prescription units express an inconsistency: *"Between 5 am and 2 pm the machine shall monitor and control the despatch of trains at 12 minute intervals;"* and *"Between 11 am and 7 pm the machine shall monitor and control the despatch of trains at 15 minute intervals."* ∎

[1] We cover abstraction principles, techniques and tools in Vols. 1–2 of this series of textbooks on software engineering where the current volume is the third.

The P in the above example is: *"Between 11 am and 2 pm the machine shall monitor and control the despatch of trains at 12 minute intervals;"* and the not P in above example is: *"Between 11 am and 2 pm the machine shall monitor and control the despatch of trains at 15 minute intervals."*

21.3.2 Conflicts

Characterisation. By *conflict* of a *requirements prescription* we shall understand a requirements prescription inconsistency, in which some requirements stakeholders strongly adhere to one set of requirements prescriptions, inconsistent with another set of requirements prescriptions, strongly adhered to by other requirements stakeholders — and such that this conflict can only be resolved through negotiation, including possible business process reengineering, between up to several levels of management. ▪

Inconsistencies which can be resolved (i.e., removed) by discussion between requirements stakeholders are not conflicts. Conflicts are usually deeply rooted — and are not within the prerogative of the requirements engineers to solve.

21.3.3 Incompleteness

Characterisation. By *incompleteness* of a *requirements prescription* we shall understand a prescription which leaves completely open the values of entities, the function argument/result value relation, or the process behaviour; or which indicates some alternative possibilities of entity attributes, function argument/result pairs, or behaviour alternatives, without indicating (i.e., describing) all (obvious) such. ▪

Example 21.5 *A Requirements Prescription Incompleteness:* We give a few examples:

"The machine shall monitor that adult passengers pay full fare, children half. The machine shall sound an alarm when passengers do not pay accordingly". An incompleteness could be that there is no requirements prescription for what is meant by passengers and hence by adult and child passengers, or that — say — soldier, student and pensioner transport fees are not covered.

"A ***free*** *rail unit shall be available for scheduling. A* ***locked*** *train unit is one which has been scheduled but is as yet not occupied by a train. An* ***occupied*** *train unit has a train passing it."* An incompleteness could be that there is no prescription for — say — rail units that have just been passed by a train but possibly not yet (made) available for scheduling. ▪

Incomplete requirements prescriptions may be made complete, or left as such. Final incomplete requirements prescriptions may be a source for expressing domain requirements that "repair" any conceived incompletenesses.

21.3.4 Looseness and Nondeterminism

Characterisation. By a *loose requirements prescription* we shall understand the same as an incomplete requirements prescription. ▪

Characterisation. By a *nondeterministic requirements prescription* we shall understand a requirements prescription which deliberately leaves open such things as function argument/result values, choice amongst alternative behaviours, ordering of events, etc. ▪

Example 21.6 *Nondeterministic Requirements Prescriptions:* We refer to Example 13.7. ▪

21.4 From Analysis to Synthesis

Once a reasonably thorough analysis and concept formation has been done, the requirements engineer can create a requirements prescription, i.e., a requirements model, according to the principles and techniques outlined in earlier chapters.

21.5 Discussion

21.5.1 General

Which tool support is available for discovering inconsistencies and incompleteness? To answer that question in the positive, we need reflect, a bit, on the form of our prescription units: If they are formulated in an informal language, a notation without any formal semantics or proof system, then there is only the human brain and informal, but precise reasoning to resort to. If they are formulated in a formal language, a notation with a formal semantics, or a proof system, then model checking and theorem proving may be attempted. We shall, in the present edition of these volumes, not venture further into this important area, other than saying that there do indeed exist tools and techniques for assisting in the discovery of requirements inconsistencies and incompleteness.

21.5.2 Principles, Techniques and Tools

We summarise:

Principle. *Requirements Analysis:* The principle applies to requirements descriptions and shall ensure that prescriptions are consistent, relatively complete and based on the "narrow bridge" principle of pleasing sets of a relatively small set of designations and defined concepts. ▪

Principle. *Concept Formation:* This principle applies also to requirements prescriptions and shall ensure that requirements prescriptions are based on "telling", i.e., pleasing concepts. ▪

Techniques. *Requirements Analysis:* When applied to informal requirements prescriptions, including informal prescription units, the analysis techniques are likewise informal. When applied to formalised requirements prescriptions, including informal prescription units, the preferred techniques focus on 'abstract interpretation': From static data and control flow analysis, to symbolic "execution" of the formal prescriptions texts, followed by human interpretation of the results. ▪

Techniques. *Concept Formation:* The techniques of discovering concepts are, naturally, those of exploration and experimentation, of skepticism, and of conjecturing and refuting. These investigative techniques again require further techniques, usually those of an inquisitive, scientific mind. ▪

The reader will have discerned, in the characterisation of 'Concept Formation', that (human) ingenuity is called for. So be it.

Tools. *Requirements Analysis:* When requirements analysis is applied to informal requirements prescriptions, including informal prescription units, the tools are likewise informal: human brain power, and good text-processing facilities. When domain analysis is applied to formalised domain descriptions, including informal description units, the tools include such as can perform abstract interpretation, i.e., flow analysis on formal texts. ▪

Tools. *Concept Formation:* Human brain power. ▪

21.6 Bibliographical Notes

The following researchers have contributed significantly to the field of requirements engineering — and are mentioned in this chapter due to the tool-oriented nature of their work: B. Nuseibeh, A. Finkelstein, A. Hunter, and J. Kramer: [177, 263]; John Mylopoulos, A. Borgida, L. Chung, S.J. Greenspan, and E. Yu: and [126, 127, 253–255, 380]; Joseph A. Goguen, M. Girotka and C. Linde: [123,124]; Axel van Lamsweerde, A. Dardenne, R. Darimont, M. Feather, S. Fikas, R. De Landtsheer, E. Letier, C. Ponsard, and L. Willemet: [76–78, 102, 211, 214, 215, 360–366].

21.7 Exercises

21.7.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

21.7.2 The Exercises

Exercise 21.1 *Requirements Inconsistencies.* For the fixed topic, selected by you, formulate three to four examples of inconsistent requirements prescription units.

Exercise 21.2 *Requirements Conflicts.* For the fixed topic, selected by you, formulate two examples of conflicting requirements prescription units.

Exercise 21.3 *Requirements Concepts.* For the fixed topic, selected by you, formulate four or more requirements prescription units, from which, by simple analysis, you can come up with at least two concepts.

22

Requirements Verification and Validation

- The **prerequisite** for studying this chapter is that you have a reasonable grasp of the previous stages of requirements engineering: From requirements acquisition, via analysis and concept formation, to requirements prescription (i.e., requirements modelling).
- The **aims** are to briefly introduce the concepts of requirements verification (including model checking and testing) and validation, and to cover some of the attendant principles and techniques.
- The **objective** is to complete your education and training so as to become a professional requirements engineer.
- The **treatment** is informal.

This chapter follows, almost line by line, Chap. 14: 'Domain Verification and Validation'.

22.1 Introduction

Let us first review where we are, in the current chapter, in the process of describing the requirements development process and its method principles and techniques:

(i) First we focused on the core aspects of requirements modelling: The "whats" and "hows" of a requirements model, we could call it the "production technology": (i.1) First some preliminaries, Chap. 17, (i.2) included an overview of the requirements development process, Sect. 17.5. (i.3) Then we restated the notion of stakeholders, Chap. 18. (i.4) The major part of this chapter, Chap 19 then covered principles and techniques for constructing a requirements prescription.

That coverage explained "what" a requirements model should contain, the facets "mirrored", and — notably — with respect to — the stakeholders and the perspectives to be dealt with.

(ii) Then we focused more on "how". In contrast to "production technology" we could call this (the "how") the "process technology":

(ii.1) First the process, principles and techniques, of requirements acquisition, Chap. 20, that which "begins" the requirements development work. (ii.2) Then the process, principles and techniques of requirements analysis and concept formation, Chap. 21.

After requirements acquisition and requirements analysis and concept formation followed the requirements modelling proper — that which we covered earlier, as mentioned above under (i.2) in Chap. 19. Finally follows domain validation and verification — the topic of this chapter.

The purpose of the above review has been to put the somehow "reverse" ordering of the chapter sections "straight" wrt. the ordering of the requirements development processes.

We can now summarise, even before we have covered the notions of validation and verification, the requirements development process: After producing appropriate informative documents — needs and ideas, concepts, scope and span, assumptions and dependencies, implicit cum derivative goals, synopsis, and contracts — one proceeds to identifying requirements stakeholders and establishing liaison with members of requirements stakeholder groups. Then on to requirements acquisition: interviews, studies, questionnaire formulation and requirements stakeholders' replies to these, ending with requirements prescription unit indexing and an elicitation report. This (acquisition) being followed by requirements analysis and concept formation. Then the actual requirements modelling. And finally requirements validation and verification.

22.2 Requirements Verification

In this chapter (as in Chap. 14) we use the term 'verification' to also cover the concepts of model checking and testing.

Characterisation. By *requirements verification* we shall understand a process, and the resulting (analytic) documents, in which some requirements prescriptions are being analysed in order to ascertain whether what is being described satisfies certain (claimed or otherwise expected) properties. ▪

So what — really — is the difference between requirements validation and requirements verification?

In validation we examine the requirements model to make sure we are modelling what the requirements stakeholders think that domain is: *"Validation gets the right requirements model"*. In verification we examine whether our requirements model "hangs together," such as the requirements engineers want it to be: *"Verification gets the requirements model right"*.

(The above, oft-quoted distinction was, it seems, first formulated by Barry Boehm in [42].)

Verification is adjoint to validation: Both validation and verification are needed. Usually verification precedes validation.

Verification work typically proceeds as follows: Desired properties of the requirements model, properties that do not transpire immediately from the requirements prescription, are formulated, informally or formally, and then "proofs" by "verbal" arguments, or some form of symbolic testing, or formal proofs, or model checking, is, or are, performed in order to check that the desired property holds of the requirements model.

So verification, to us, includes, rearranging the terms a bit: informal reasoning, that is, "proofs" by "verbal" arguments and testing; and formal reasoning, that is, formal proofs and model checking.

By informal reasoning we shall, however, mean "proofs" by "verbal" arguments.

22.2.1 Informal Reasoning

Characterisation. *Informal Reasoning:* By *informal reasoning* we shall understand a carefully phrased series of arguments, which, as a whole, convinces an audience to these arguments of the validity of what is concluded. . ∎

Human beings often reason, but are not always careful in doing so. Informal reasoning demands great care.

22.2.2 Testing

Characterisation. *Testing:* By *requirements testing* we shall understand that a requirements prescription is provided with set values for all relevant arguments (the test data), with the prescription then being evaluated ("executed") for those arguments. The test then results in a final value of the prescription for those arguments. ∎

Such a final value may be a complicated quantity. Typical final values could be: an execution sequence, a trace of prescription points, with sets of variable values for each step in the sequence (i.e., trace).

Another way of phrasing it: testing is a systematic search for a counterexample to a claim (of proof) of correctness.

Testing has till now basically been a heuristics-based science. An important ingredient in performing testing is (formal) text analysis. If requirements prescription parts have been formalised, then theory-based testing technologies have or can be developed and can be used for testing. Section 29.5.3 (of Chap. 29) covers testing in more detail.

22.2.3 Formal Proofs

Characterisation. By a *formal proof* we shall understand a given requirements prescription, a statement (a theorem) to be proved, and the proof that the requirements prescription satisfies the statement: This proof refers to a proof system for the language in which the requirements prescription is expressed (axioms and inference rules), and is otherwise a sequence, composed from steps, where each step in the sequence is like a theorem (a lemma), a statement, and where pairs of steps in the proof sequence are related by the axioms, and the inference rules. ▪

Since we are, at this stage, not exemplifying or relying seriously upon a formal specification language, we shall not exemplify any formal proofs. Such formal proofs are exemplified elsewhere.

22.2.4 Model Checking

Characterisation. *Model Checking:* By *model checking* we shall understand "a method for formally verifying usually concurrent systems whose usually extremely large, practically speaking infinite state systems, have been reduced to manageable finite state systems". ▪

(The quoted phrase, above, is taken from the home web page of the Carnegie Mellon University Model Checking group.[1])

Requirements prescriptions about such finite state systems are typically expressed as temporal logic formulas. Efficient symbolic algorithms are used to traverse the model defined by the system and check if the requirements prescription holds or not — for a "reduced" set of possible states of systems. Extremely large state spaces can often be traversed in minutes.

22.3 Requirements Validation

Characterisation. By *requirements validation* we shall understand a process, and the resulting (analytic) documents, in which some requirements prescriptive documents are being inspected by both requirements stakeholders and requirements engineers, and in which, whatever is being prescribed, is being positively and/or negatively reviewed (i.e., positively and/or negatively criticised) with reference to the elicitation report and with respect to whatever the requirements stakeholders might now realise about their expectations, including the pointing out, if necessary, of inconsistencies, incompletenesses, conflicts and errors of prescription that may change the elicitation report.

Requirements validation is possibly interwoven with requirements verification work. See below. ▪

[1] http://www-2.cs.cmu.edu/~modelcheck/

22.3.1 The Requirements Validation Documents

In order to perform requirements validations, the validators need the following (input) documents: (i) The list of requirements stakeholders; (ii) the requirements acquisition documents: Questionnaire, and the collection of indexed prescription units; (iii) the rough sketch, terminology, narrative, and possibly — if produced — the formalisation documents that constitute the requirements prescription proper; and (iv) the requirements analysis and concept formation documents. That is: Basically all documents produced (so far) in the requirements modelling effort.

In order to complete requirements validation, the validators produce the following (output) documents: (i) A possibly updated requirements stakeholder document; (ii) possibly updated requirements acquisition documents; (iii) possibly updated rough sketch, terminology, narrative, and — if relevant — the formalisation documents; (iv) possibly updated requirements analysis and concept formation documents; and (v) a requirements validation report.

We now cover some aspects of the necessarily informal validation process.

22.3.2 The Requirements Validation Process

Requirements validation proceeds as follows: Requirements engineers "sit together" with requirements stakeholders and review, line-by-line, the domain model, holding it up against the previously elicited requirements prescription units, while then noting down any discrepancies.

In doing requirements validation requirements stakeholders usually read the informal, yet precise and detailed narrative prescriptions. No assumption is made as to their ability to read formalisations. On the contrary: It is assumed that they cannot read formal specifications.

For reasonably large-scale projects the customer may hire professional consultants who can also study the formalisations — just like future ship owners hire Lloyd's Register (www.lr.org/ [224])[2] to check ship designs in preparation for insurance companies to take on insurance risks.[3]

Requirements validation (and verification) ends with a requirements validation (and verification) report which either accepts the requirements model, or which points out needs to correct the elicitation report, the requirements analysis and concept formation report, and the requirements model.

[2] Or such similar companies as Bureau Veritas (www.bureauveritas.com/ [51]), Norwegian Veritas (DNV: Den Norske Veritas, www.dnv.no/ [262]), or the German TÜV (www.tuev-sued.de/ [356]).

[3] The staff of these, and similar design quality assurance companies are oftentimes very sophisticated software engineers, well-versed in formal software development and verification methods.

22.3.3 Requirements Development Iterations

Thus requirements validation (and verification) can be expected to be an iterative process alternating possibly with further requirements verification, possibly with further requirements elicitation report work, possibly with further requirements analysis and concept formation work, and with further requirements modelling work; and ending with with further requirements validation (and verification) work.

22.4 Discussion

22.4.1 General

This chapter was a repetition, almost line-by-line, of Chap. 14: 'Domain Verification and Validation'.

We have treated aspects of requirements validation and verification — and in the same chapter. We have done so since they relate in many ways.

And we have used the term 'verification', primarily to stand for formal proofs, but, secondarily, also for 'model checks' and 'tests'.

22.4.2 Principles, Techniques and Tools

We summarise:

Principle. *Requirements Validation:* To ensure that the requirements prescribed are the right requirements. ▪

Principle. *Requirements Verification:* To uncover a domain theory, i.e., to get the requirements prescriptions right. ▪

Techniques. *Requirements Validation:* In summary: Human, collaborative document inspection. See Sect. 14.3.2. ▪

Techniques. *Requirements Verification:* Verification techniques, based on formal descriptions, include such which enable formal verification (of posed lemmas and theorems), model checking, and tests, while requirements verification techniques, based on informal descriptions, basically amount to informal, concise reasoning. ▪

Tools. *Requirements Validation:* Since requirements validation is basically an informal process, the tools are tools that support document cross-referencing (between requirements description units and narrative requirements descriptions and requirements terminologies), and data mining based on such documents. ▪

Tools. *Requirements Verification:* Requirements verification based on formal requirements prescriptions requires such tools as for example proof assistants and theorem provers, model checkers, and test generators and tester monitors; whereas requirements verification based on informal prescriptions basically requires human reasoning. ▪

22.5 Bibliographical Notes

Seminal books on model checking are [59, 173].

22.6 Exercises

22.6.1 Preamble

The first 4 exercises (22.1–22.4) of this chapter are *closed book* exercises. That means that you are to try write down a few lines of your solution before you check with the appropriate section for our answer to the questions. Exercises 22.5 and 22.6 test the problem-solver's ability to lead a group of two or more requirements validators, respectively requirements verifiers.

22.6.2 The Exercises

Exercise 22.1 *Requirements Validation Documents.* Which are the requirements development documents that are needed in order to commence proper requirements validation, and which are the resulting documents?

Exercise 22.2 *Requirements Validation Process.* Outline, in brief, i.e., in a few itemised lines, the requirements validation process.

Exercise 22.3 *Requirements Verification, Model Checking and Testing.* Explain, in brief, in a few itemised lines, the concepts of formal verification, of model checking and of testing.

Exercise 22.4 *Requirements Validation Versus Domain Verification.* Explain in two itemised lines the difference between the objectives of requirements validation and requirements verification.

Exercise 22.5 *Specific Domain-Specific Requirements Validation.* For the fixed topic, selected by you, and on the background of your solutions to some of exercises 19.4 to 19.6, suggest, preferably three or more, issues that may need special attention during requirements validation.

Exercise 22.6 *Specific Domain-Specific Requirements Verification.* For the fixed topic, selected by you, and on the background of your solutions to some of exercises 19.4 to 19.6, suggest, preferably three or more, issues that may need special attention during requirements verification.

23

Requirements Satisfiability and Feasibility

- The **prerequisite** for studying this chapter is that you have studied the previous three chapters.
- The **aims** are to introduce the concepts of satisfiability, feasibility and compliance; to relate satisfiability to properties of requirements documents and feasibility to implementability of the required machine, and to relate compliance to implied and implicit, i.e., metagoals.
- The **objective** is to round off your education as a professional requirements engineer.
- The **treatment** is informal and systematic.

23.1 Introduction

Four kinds of questions may be asked about a requirements document.

- Is it well-formed? We take this to mean: is it objectively demonstrable? That is, can, and are, reasonably comprehensive tests devised for which stated requirements can objectively be shown to hold?
- Is it satisfactory? We take this to mean: is it *correct*, *unambiguous*, *complete*, *consistent*, *stable*, *verifiable*, *modifiable*, *traceable* and *faithful*? We shall investigate these questions in Sect. 23.2.
- Is what it prescribes feasible? We take this to mean: is it technologically feasible (can it be built) and is it economically feasible (is it within reasonable costs)? We shall investigate these questions in Sects. 23.3 and 23.4.
- Do the requirements address implicit/derivative goals? We shall investigate this last question in Sect. 23.5.

Please note that the four questions are metaquestions. They are not substitute questions dealing with verification and validation directly, but they somehow imply these — however implicitly.

23.2 Satisfaction Study

Characterisation. *Requirements Satisfiability:* We enumerate the criteria that a requirements document must satisfy in order to be satisfactory: *correctness (validated), unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability* and *faithfulness,* i.e., meets assumptions and dependencies. ▪

This edited list, except for "faithfulness", is taken from Hans van Vliet's delightfully refreshing book [369] (pp. 225–226). We will next treat each of these criteria separately.

23.2.1 Correct (Validated) Requirements Document

For us, for a requirements document to be correct means that it has been thoroughly validated with respect to all requirements stakeholders, and that the client has accepted the final document.

23.2.2 Unambiguous Requirements Document

For us, for a requirements document to be unambiguous means that no inconsistencies, no vaguenesses and no double meanings remain in the final requirements document. In other words: It is precise.

23.2.3 Complete Requirements Document

For us, for a requirements document to be complete means that no (unintentionally placed) holes can be pointed out, that is, everything needed to be prescribed has been prescribed. Completeness is thus relative. It is only wrt. what "needs" be described, not what "can" be described.

23.2.4 Consistent Requirements Document

For us, for a requirements document to be consistent means the same as being unambiguous.

23.2.5 Stable Requirements Document

Another term for "stability" is "importance". A simple ranking of individual (itemised) requirements into (i) essential requirement (must be implemented), (ii) worthwhile requirement (would be very nice if implemented) and (iii) optional requirement (implement if not too costly), and with this ranking being stable during the requirements engineering phase, to us characterises a stable requirements document.

23.2.6 Verifiable Requirements Document

This criterion relates to the implementation. For a requirements to have been met by an implementation means that it can be proven or tested (i.e., checked) that a given requirement has been implemented and meets expectations. Some requirements cannot be so tested, at least not objectively. Certainly today (year 2005) rather many machine requirements, and some interface requirements are hard to quantify, let alone measure.

23.2.7 Modifiable Requirements Document

It is often claimed that *requirements change all the time*. Whenever such changes actually do happen one needs to modify the existing requirements document. To do so, it is of paramount importance that the requirements prescription document follows a hopefully existing domain description document. This makes it easier to find, we claim, where changes to the requirements prescription document need be made, and, given that the final criterion (*traceability*, see next) is met, to trace repercussions of the change.

We claim the following: With carefully worked out domain descriptions you will find that domain and interface requirements do not "change all the time". The domain is as it is. It changes very little. Its intrinsics are ultimately stable. With domain and, to an important extent, also interface requirements prescriptions being "derived" from the domain descriptions you will find that many potentially unclear situations — which in the past led to "changing requirements" — do not arise.

23.2.8 Traceable Requirements Document

For us, for a requirements document to be traceable means that every requirements (statement), r_s, is annotated with its origin (whom, when, where), and that the reason (rationale) for the requirements is well-documented. Furthermore, traceability means that one can simply find all those other requirements (statements), $r_{s_1}, r_{s_2}, \ldots, r_{s_n}$, on which the meaning of the given requirement depends, that is, r_s relies on $r_{s_1}, r_{s_2}, \ldots, r_{s_n}$, or whose meaning depends on the given requirement, that is, $r_{s_1}, r_{s_2}, \ldots, r_{s_n}$ rely on r_s. Suitable recording schemes must therefore be provided by suitable requirements documentation tools.

23.2.9 Faithful Requirements Document

Assumptions and dependencies (Sect. 2.4.5) expressed in informative documents need to be implied by the requirements. To put it differently: If the requirements development, from the outset, was judged to build upon some assumptions about the environment of the domain, that which lies outside

the domain, and hence, in a sense, outside the requirements, then those assumptions must somehow have found their way into constraints on entities or preconditions of functions, events and behaviours. The similar case holds for dependencies. These assumptions and dependencies must be so checked.

23.2.10 Discussion of Satisfiability

Thus satisfiability is more of a syntactic criterion. In review, it has to do with: correct (validated), unambiguous, complete, consistent, stable, verifiable, modifiable, traceable and faithful documents. Satisfiability is clearly a software engineering concern. The way in which the requirements are developed must a priori secure satisfiability. Yet some "outside" satisfiability auditing must be performed. That is mostly a management issue.

23.3 Technical Feasibility Study

Given an otherwise satisfactory requirements document, the question is now: *Is it technically feasible?* We see three aspects in this: *Is business process reengineering feasible? Are hardware solutions feasible?* And: *Are software solutions feasible?*

Characterisation. *Requirements Feasibility, Technical:* A requirements is technically feasible if it is possible to implement the business process reengineering, to implement a hardware solution and a software solution. ▪

23.3.1 Feasibility of Business Process Reengineering

Is the business process reengineering feasible in the given (customer, user) environment? If not, how should we modify the requirements?

To find out whether a business process reengineering requirements is possible, the requirements engineer and the client's requirements stakeholder liaisons need discuss the business process reengineering requirement with all staff involved to ascertain their willingness and ability to adapt.

23.3.2 Feasibility of Hardware

Can hardware (be found, acquired, and operated to) meet interface and machine requirements? If not, how do we modify the requirements?

23.3.3 Feasibility of Software

Can software (be found, acquired or developed, and operated to) meet domain, interface and machine requirements? If not, how should we modify the requirements?

23.3.4 Discussion of Technical Feasibility

To answer each of the three kinds of technical feasibility questions in a believable manner implies serious software engineering cum systems engineering studies of the requirements document, software engineering cum systems engineering studies of potential hardware and software designs and studies of potential business process reengineering alternatives. The latter typically requires "management engineering" capabilities. Thus we shall not offer any guidelines for these three areas. Objective technical feasibility studies, although also part of software engineering, are still basically untrodden land, scientifically.

23.4 Economic Feasibility Study

Given an otherwise satisfactory requirements document, one which meets the technical feasibility criteria, the question is then: *Is it economically feasible?* We see three aspects in this: *Can the development costs be funded? Can the development costs be amortised over a reasonable period? Do the advantages of using the desired computing system outweigh the development and operating costs?*

Characterisation. *Requirements Feasibility, Economics:* A requirements is economically feasible if trusted development cost estimates can be estimated and funded, if these costs can be amortised and if these costs outweigh disadvantages of not having the required software (i.e., *are worth it*). ∎

23.4.1 Feasible Development Costs

Are we satisfied that further development cost estimates are believable and can be funded? If not, how do we modify the requirements?

23.4.2 Feasible Write-off Costs

Is the amortisation cost commensurate with the lifetime of the computing system? If not, how should we modify the requirements?

23.4.3 Gains Outweigh Costs?

Do we gain more than we have to pay, and are the gains worth it? If not, how do we modify the requirements?

23.4.4 Discussion of Economic Feasibility

To answer each of the three kinds of economic feasibility questions in a believable manner implies serious software engineering cum economics studies of the requirements document, software engineering cum economics studies of potential hardware and software designs, and studies of potential business process reengineering alternatives. The latter typically requires "economics engineering" capabilities. Thus we shall not offer any guidelines for these three areas. Objective economic feasibility studies, although also part of software engineering, are still basically untrodden land, scientifically.

23.5 Compliance with Implicit/Derivative Goals

23.5.1 Review of Implicit/Derivative Goals

We refer to Sect. 2.4.6 for an explanation of what is meant by implicit/derivative goals. Briefly, by an implicit/derivative goal we mean: a goal which is not explicitly expressed in a requirements, but which is nonetheless expected to be met by the computing system to be developed from the requirements, for example, *greater company profit, larger market share, fewer workplace accidents,* and *improved customer satisfaction.*

23.5.2 Discussion of Implicit/Derivative Goals

As is obvious from the above, implicit/derivative goals cannot be quantified in a way that "points" to, or indicates what the prescribed computing system should offer. We therefore decided to place the statement of such goals in the informative documentation part, but should provide a "back-reference" from the requirements document to the appropriate (implicit/derivative goal) section of the informative documentation. The actually prescribed requirements stand in no rational relationship to these metagoals. Hence it is up to a review board of the parties to the requirements development contract to review — and "sign off" on — whether the requirements are indeed believed to be a best thinkable way to achieve the metagoals.

23.6 Discussion

23.6.1 General

To answer each of the three kinds of economic feasibility questions in a believable manner implies serious experience in estimating software design costs, as well as economic and management studies of lifetime and trade-off issues. We refrain from offering such guidelines wrt. software development cost estimation.

23.6.2 Principles, Techniques and Tools

We summarise:

Principles. The principle of *requirements satisfiability* is basically one of delivering an orderly, or "neat" set of requirements documents: Correct (validated), unambiguous, complete, consistent, stable, verifiable, modifiable, traceable and faithful documents. ∎

Principles. The principle of *technical satisfiability of requirements* is basically one of checking whether the requirements be implemented. As such, the principle implies that "best practices" of both software engineering and business process reengineering (BPR) be applied. Thus, besides ("best software engineering practices") proper domain engineering, proper requirements engineering, and inspired and innovative software design, BPR is important. Some BPR can be carried out by the software engineers, "the rest" by management engineering. ∎

Principles. The principle of *economic satisfiability of requirements* is basically one of checking: Would a computing system implemented according to requirements meet economic expectations? As such, the principle implies that best practices of "economics management and engineering" be applied. This is basically untrodden land. It is not (yet, really) part of software engineering. So, we give little advice on the matter. ∎

Techniques. *Requirements Satisfiability:* Any techniques that will check for correctness (validated), ambiguity, completeness, consistency, stability, verifiability, modifiability, traceability and faithfulness will do. Basically, some of the above are separately handled by respective processes: validation, verification, testing, model checking, etc. Others can be, and are, 'checked' by the practice of actually working with these documents — so checking satisfiability for these ("others") is a matter of asking for experience. ∎

Techniques. *Requirements Technical Feasibility:* As can be gleaned from our statement concerning *technical satisfiability of requirements*, the techniques are those that are needed in order to answer the double question: *Have proper domain and requirements engineering led to a requirements which can be implemented* (i) *by a computing system (hardware and software),* and (ii) *if needed, by suitably reengineered business processes?* The first question needs to be answered by the leader of a team of available software engineers (SE), and requires their competence in software design coupled with the team leader's confidence in the team's abilities. The (SE) techniques are really metatechniques. The latter question, BPR, must be answered by BPR engineers. Again, the BPR techniques are really metatechniques. ∎

Techniques. *Requirements Economic Feasibility:* As can be gleaned from our statement concerning *economic satisfiability of requirements*, the techniques are those that are needed in order to answer the *economic feasibility* question. These techniques are basically outside the established regime of software engineering, at least as covered in the present three volumes. So we have to refer you to technical articles, if and when they exist and/or appear on the issue of checking economic feasibility. ▪

23.7 Exercises

23.7.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

23.7.2 The Exercises

Exercises 23.1–23.4 test the problem-solver's ability to lead a group of two or more requirements satisfiability, feasibility and indirect goal compliance checkers. The exercises require that you have worked out suitable solutions to Exercises 19.4–19.6.

Exercise 23.1 *Requirements Satisfiability.* For the fixed topic, selected by you, outline, in at most one third of a page, specific issues, from your selected topic, where expressed requirements may need special satisfiability attention. That is, are there specific correctness, ambiguity, completeness, consistency, stability, (especially perhaps) verifiability, modifiability, traceability, or faithfulness issues that may cause satisfiability problems?

Exercise 23.2 *Requirements Technical Feasibility.* For the fixed topic, selected by you, outline, in at most one half-page, specific issues, from your selected topic, where expressed requirements may need special technical feasibility attention. That is, could there be requirements (that perhaps are not expressed in your solution) which are not technically feasible? Try to list two or three such (i.e., you may have to "dream them up").

Exercise 23.3 *Requirements Economic Feasibility.* For the fixed topic, selected by you, outline, in at most one half-page, specific issues, from your selected topic, where expressed requirements may need special economic feasibility attention. That is, could there be requirements (that perhaps are not expressed in your solution) which are not economically feasible? Try to list two or three such (i.e., you may have to "dream them up").

Exercise 23.4 *Implicit/Derivative Requirements.* For the fixed topic, selected by you, outline, in at most one half-page, specific issues, from your selected topic, where expressed implicit/derivative goals may need special compliance attention. That is, could there be implicit/derivative goals which are not met by your requirements. Try to list two or three such (i.e., you may have to "dream them up").

24

The Requirements Engineering Process Model

- The **prerequisite** for studying this chapter is that you have completed the study of the previous chapters in this, the requirements engineering part of this volume.
- The **aim** is to summarise the essence of the previous seven chapters.
- The **objective** is to make sure you have a good overview of all stages and steps of the requirements engineering phase.
- The **treatment** is a summary and systematic.

24.1 Introduction

We have completed, almost, covering a crucial set of principles and techniques for software development. Sandwiched in-between domain development and software design, requirements development serves as one of the three parts of our software engineering triptych.

24.2 Review of Requirements Development

We itemise the major stages and steps of requirements development:

- Stakeholder liaison: Chap. 18
  All relevant stakeholders must be identified, formal contacts established, and regular liaison kept ongoing.
- Requirements acquisition: Chap. 20
  Studies of examples of previous requirements from similar applications, interviews with stakeholders, questionnaire preparation and prescription unit indexing.
- Requirements analysis and concept formation Chap. 21
- Requirements modelling Chap. 19
  - ⋆ Business process reengineering: review of domain facets

  - ⋆ Domain requirements: projection, instantiation, determination, extension, fitting
  - ⋆ Interface requirements: identification of shared phenomena, shared data initialisation, shared data refreshment, man-machine dialogue, physiological interface, machine-machine dialogue
  - ⋆ Machine requirements: performance, dependability, maintainability, portability, documentation
- Requirements validation and verification — Chap. 22
- Satisfiability and feasibility — Chap. 23

24.3 Review of Requirements Documents

In Sect. 17.6 we commented upon what, ideally, a complete set of requirements documents would consist of. The aim of requirements engineering is to create informative, descriptive and analytic documents about and constituting the requirements. Therefore it is important to always keep in mind what a possible contents listing could be of such a complete set of documents. We have now covered all the principles, techniques and tools necessary, but not necessarily sufficient, to actually construct requirements documents for large-scale computing systems.

24.4 The Repeat Table of Contents Listing

We repeat this requirements documents listing from Sect. 17.6.2.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas (Eurekas, I)
   (e) Concepts and Facilities (Eurekas, II)
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis (Eurekas, III)
   (j) Standards Compliance
   (k) Contracts, with Design Brief
   (l) The Teams
      i. Management
      ii. Developers
      iii. Client Staff
      iv. Consultants
2. Prescriptions
   (a) Stakeholders
   (b) The Acquisition Process
      i. Studies
      ii. Interviews
      iii. Questionnaires
      iv. Indexed Description Units
   (c) Rough Sketches (Eurekas, IV)
   (d) Terminology
   (e) Facets:
      i. BPR
         - Sanctity of Intrinsics
         - Support Technology
         - Management and Organisation
         - Rules and Regulations
         - Human Behaviour
         - Scripting
      ii. Domain Requirements
         - Projection

- Determination
- Instantiation
- Extension
- Fitting

iii. Interface Requirements
- Shared Phenomena and Concept Identification
- Shared Data Initialisation
- Shared Data Refreshment
- Man-Machine Dialogue
- Physiological Interface
- Machine-Machine Dialogue

iv. Machine Requirements
- Performance
  - ⋆ Storage
  - ⋆ Time
  - ⋆ Software Size
- Dependability
  - ⋆ Accessibility
  - ⋆ Availability
  - ⋆ Reliability
  - ⋆ Robustness
  - ⋆ Safety
  - ⋆ Security
- Maintenance
  - ⋆ Adaptive
  - ⋆ Corrective
  - ⋆ Perfective
  - ⋆ Preventive
- Platform (P)
  - ⋆ Development P
  - ⋆ Demonstration P
  - ⋆ Execution P
  - ⋆ Maintenance P
- Documentation Requirements
- Other Requirements

v. Full Requirements Facets Documentation

3. Analyses
   (a) Requirements Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompletenesses
      iv. Resolutions
   (b) Requirements Validation
      i. Stakeholder Walkthroughs
      ii. Resolutions
   (c) Requirements Verification
      i. Theorem Proofs
      ii. Model Checks
      iii. Test Cases and Tests
   (d) Requirements Theory
   (e) Satisfiability and Feasibility
      i. Satisfaction: correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
      ii. Feasibility: technical, economic, BPR

24.5 Discussion

Some people like to have a diagram, like a flow chart, or a workflow model, in front of them. Figure 24.1 shows one that purports to "picture" the requirements development process model. Figure 24.2 shows the requirements modelling box of Fig. 24.1 in detail.

The essential property of Fig. 24.1 is to emphasise the iterations that are normally needed in order to reach an acceptable requirements model.

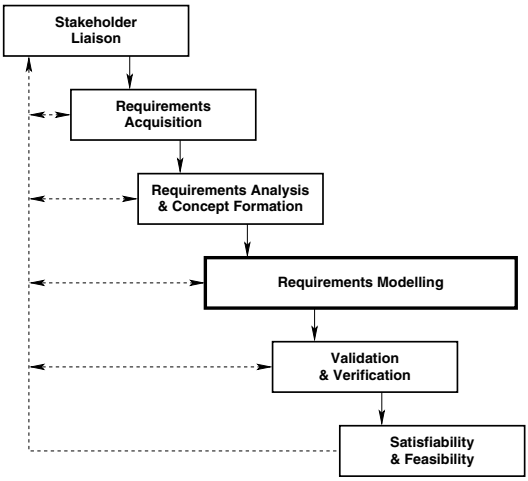


Fig. 24.1. Diagramming a requirements process model

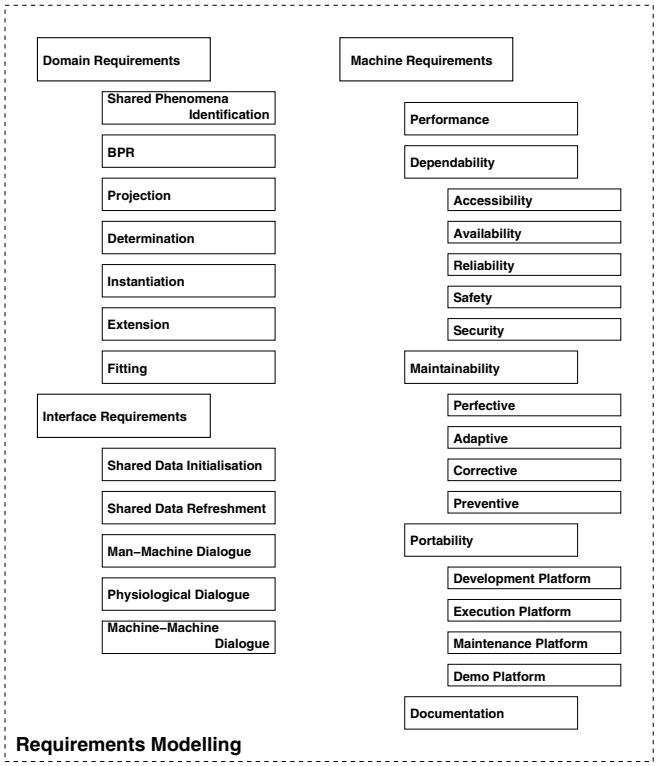


Fig. 24.2. The requirements modelling stage

Part VI

COMPUTING SYSTEMS DESIGN

In this part, Chaps. 26–30, we shall, at long last, at least for this volume, cover software design. We do so by "lifting" software design, in Chap. 26, to computing systems, or as we rephrase it, hardware/software codesign. For the rest of Part VI we then focus, more narrowly, on software design.

In Chap. 26 we stepwise unfold the component structure and component interfaces of an example timetable system — the example that has appeared, in one form or another, in earlier chapters on domains and on requirements.

In Chap. 27 we stepwise unfold the component structure and component interfaces of an "old-fashioned" file system as implemented in terms of main backing storage and disk storage. That long, but illustrative example, also shows the way in which, in a software design process, one can extend the facilities offered by that software. That is, in requirements engineering we did have a notion of 'extension': joining functionalities to a domain requirements. In Chap. 27 we join functionalities to a software design, that is, "discover" software function necessities or possibilities that were, or might not have been, obvious during requirements development.

Chapter 28 covers what is known in "the trade" (of software engineering jargon) as domain-specific architectures. That chapter is inspired very much by Michael Jackson's notion of problem frames [189,191]. The chapter sketches the issues. That is, it does not go into design details — since much of that has been covered extensively in earlier volumes of this series. Suffice it to say, as an "appetiser", that Chap. 28 covers such problem frames as translators (interpreters and compilers), repository (i.e., information cum database) systems frames, client/server (including workflow systems) frames, workpiece frames, and connection frames. Some are sketched, and some are treated in more than sketch detail. The reader is referred to specialist textbooks on respective frames.

Chapter 29 covers a selection of the classical chores of coding: how to go from RSL specifications to executable code in such programming languages as, for example, SML, C++, Java and C#, and other "mundane" issues.

Chapter 30 finally covers a process model for computing systems design. That is, we summarise this part.

25

Hardware/Software Codesign

- The **prerequisites** for studying this chapter are that you have a basic grasp of Parts IV and V, Domain and Requirements Engineering, and also a good grasp of the main formal specification language of these volumes, RSL.
- The **aim** is to overview basic issues of computing systems design.
- The **objective** is to prepare the reader for the — basically software — design issues to be covered in Part VI.
- The **treatment** is systematic.

25.1 Introduction — On Architecture

We recapitulate important characterisations from Chap. 1.

Characterisation. By a *computing systems architecture* we mean a first kind of specification of hardware/software — after requirements — one which indicates **how** the hardware/software is to handle the given requirements in terms of hardware equipment and software components and their interconnection, though without detailing (i.e., designing) this equipment and these components. ∎

Characterisation. By a *hardware architecture* we mean a first kind of specification of hardware after requirements — one which indicates **how** the hardware is to be configured from equipment in order to handle some of the given requirements in terms of equipment components and their interconnection, though without detailing (i.e., designing) these components. ∎

Characterisation. 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 components and their interconnection, though without detailing (i.e., designing) these components. ∎

Characterisation. By a *software architecture design* we mean the development process of going from existing requirements, and possibly some component design, to the software architecture — producing all appropriate architecture documentation. ∎

25.2 Hardware Components and Modules

We assume, in this book, that the hardware design aspect of the computing systems design is handled by mere selection of existing, i.e., available equipment: central processors, their speed, their storage; peripheral equipment, displays, terminals, printers, etc.; data communication equipment; etc. We refer to Sect. 25.4 where we will comment on the larger aspects of hardware/software codesign.

25.3 Software Components and Modules

We refer to Sect. 1.2.4 where we cover a view of what components and modules are.

Characterisation. By a *software component structure* we mean a second kind of specification of software after determining the software architecture — one which also indicates **how** the software is to implement individual components and modules. ∎

Characterisation. By *software component design* we mean the development process of going from existing requirements, and with a software architecture design in mind, to the detailed component modularisation — producing all appropriate component and module documentation. ∎

Thus component design makes decisions about data structures and particular algorithms, and component structure then implies how procedures of one module or component invokes procedures of the same or other modules and components.

25.4 Hardware/Software Codesign

We refer to [344]. That book covers a different aspect of hardware/software codesign than was very briefly mentioned earlier in this chapter.

Normally, when bringing up the topic of hardware/software codesign, one does not (usually) mean that which we implied earlier in this chapter. Earlier, by hardware/software codesign, we meant to examine a requirements prescription and then decide, mostly from the machine requirements, which degrees

of liberty and/or constraints they gave us, respectively imposed on us with respect to our choice of the hardware part of the machine. We consider the usual aspects: speed, storage capacity, data communication bandwidth, etc.

Characterisation. By *hardware/software codesign*, in this very cursory section, we shall now mean almost something in the form of "the opposite" of the above: which of the specific functions, or data capture, or event mechanisms, should be implemented in hardware, say, by application-specific integrated circuits (ASICs, [174, 259]), or by field-programmable gate arrays (FPGAs, [50,240]). Or whether the whole thing should be hardware — as for systems-on-chips (SoC, [195, 302]). ▪

Other than this brief mention we shall not go into this topic, but refer the reader to special course and text and handbooks [50, 174, 195, 240, 259, 302].

25.5 Stepwise Refinement of Architectures

The above four characterisations (architectures, components, and, for both of them, their design) indicate an interplay between taking care of the domain requirements, the interface requirements and the machine requirements.

That interplay not only alternates between these three requirements facets, but also between the component structures, that is, components seen as "black" boxes, connected (via interfaces) to one another, and the component functions, that is, components seen as "white" boxes, whose contents and mode of operation are revealed by some module composition, and their interfaces.

25.6 Discussion

In the rest of this part, i.e., Part VI, we shall touch upon some of the issues mentioned in the present chapter.

25.7 Principles, Techniques and Tools

Principles. The basic ideas underlying the concept of, and possible need for, *hardware/software codesign* are those of seeking a balance between performance, dependability, etc., of hardware solutions as compared to software solutions, and the flexibility that software solutions give over hardware solutions: easier to modify once "a solution" is installed. ▪

Techniques. The techniques of *hardware/software codesign* have only been hinted at: Examination of each and every requirements, and/or initial architecture design, having in mind to answer the question: *Shall this requirements, this architectural design be implemented in hardware or software?* Subsidiary techniques revolve around evaluation of performance, dependability, etc., issues. ▪

As for tools, for software design they are the ones otherwise referred to throughout these volumes. For hardware/software codesign there are already some tools. We refer to [195, 302] for some references.

26

Software Architecture Design

- The **prerequisites** for studying this chapter are that you have studied and grasped the chapters on domain facets, Chap. 11, and on requirements facets, Chap. 19, and are curious to see how the further development from domains via requirements to software can evolve.
- The **aims** are to show how domain and interface requirements evolve into software architecture design, and to show, in particular, how machine requirements determine special software components.
- The **objective** is to finally, with the next two chapters, bring you on the road to becoming an all-around, professional software engineer.
- The **treatment** is from systematic to formal.

26.1 Introduction

The requirements prescriptions cover four areas: business process reengineering, domain requirements, interface requirements and machine requirements. Only the last three sets of requirements influence the design of the computing system, i.e., the machine. The business process reengineering prescriptions are meant to materially influence the behaviour of people using that machine. In this chapter we shall primarily focus on architectural consequences of some machine requirements.

The chapter is predominantly based on formalisations. The reader who wishes to study this book only informally is thus left to not be able to enjoy one of the many advantages of using formal specifications. The "closer" one gets to actual program implementation, the more one has to express the software design in a programming notation. The formalisms used are, however, simple and mostly based on the `CSP` of `RSL/CSP`. We covered that subset of `RSL` in Vol. 1, Chap. 21. So get used to it!

26.2 Initial Domain Requirements Architecture

Usually, when developing a domain requirements, we can formalise, incrementally, the resulting domain requirements, not just by their properties, but we can also give model-oriented prescriptions. The same holds for some interface requirements. But for some other interface requirements, and for most, if not all, machine requirements, we cannot formalise the properties asked for, but one can formalise their possible, claimed implementation, that is, an architectural software design.

Model-oriented requirements prescriptions amount to partial software architecture specifications. Let us "slowly" unfold such a software architecture specification.

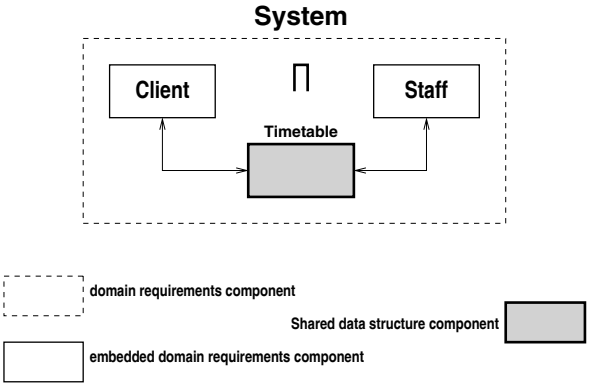


Fig. 26.1. Timetable application components

Example 26.1 *Component Diagram of a Simple Timetable System:* We refer to Example 19.14. We diagram that formalisation as shown in Fig. 26.1. The arrows here were not thought of as channels.

Formal Presentation: A Simple Timetable System

We can model the diagram of Fig. 26.1 by:

system: TT → **Unit**
system(tt) ≡ client(tt) ∥ staff(tt)

client: TT → **Unit**
client(tt) ≡ **let** q:Query **in let** v = $\mathcal{M}_q$(q)(tt) **in** system(tt) **end end**

staff: TT → **Unit**
staff(tt) ≡

```
    let u:Update in
    let (r,tt′) = 𝓜u(u)(tt) in system(tt′) end end
```

Both the airline client and the airline staff make use of the (shared phenomenon) airline timetable. It is therefore to be considered a data structure that is *shared* not only between the domain and the machine, but also between client and staff. ▪

We call the component shown in Fig. 26.1, a domain requirements *component*. In this case, we may claim that it consists of three embedded such.

We now, increasingly, since software design is our subject, turn to model-oriented specifications. In this section, we almost exclusively develop and enrich process-oriented specifications.

Example 26.2 *Formal Model of Simple Timetable System Process:* We refer to Example 26.1 (above), but assume arrows as designating channels. Each of the three subcomponents of Fig. 26.1 are now considered to be separately evolving behaviours, that is, processes.

Formal Presentation: Simple Timetable System Processes

```
channel
  ctt:QU, ttc:VAL, stt:UP, ts:RES
value
  system: TT → Unit
  system(tt) ≡ client() || time_table(tt) || staff()

  client: Unit → out ct in tc Unit
  client() ≡ let qc:Query in ctt!𝓜q(qc) end ttc?;client()

  staff: Unit → out st in ts Unit
  staff() ≡ let uc:Update in st!𝓜u(uc) end let res = ts? in staff() end

  time_table: TT → in ctt,stt out ttc,tts Unit
  time_table(tt) ≡
      let qf = ctt? in ttc!qf(tt); time_table(tt) end
    ⌷ let uf = st? in let (tt′,r)=uf(tt) in ts!r; time_table(tt′) end end
```

We can read the framed formulas above aloud for the reader who otherwise cannot read these formulas: There are two connections, two interfaces, between the client and the time_table, One in each direction. Similarly, there are two connections, two interfaces, between the staff and the time_table, One in each direction. The system behaviour is the parallel composition of

three behaviours: client, staff and time_table. Only the time_table possesses the timetable. All three behaviours, client, staff and time_table, are cyclic: Recurse indefinitely.

The client behaviour sends query requests to the time_table behaviour and awaits response before recycling. The staff behaviour sends update requests to the time_table behaviour and awaits response before recycling. In either case, the client and staff behaviours — before resuming their behaviour — ignore the response.

The time_table behaviour, as an obedient server, is ready, in each round, each cycle, to engage in an event with either the client or the staff behaviours. The time_table behaviour expresses this by an external nondeterministic choice ([]). We refer to Example 19.32. ▪

• • •

In Examples 19.27 and 19.28 we exemplified aspects of interface requirements for the example of the present chapter. One could claim, and with some justification, that what Example 19.28 illustrated could as well be said to constitute a software design specification. Other than this fleeting reference to interface software design, we shall not, in this chapter, illustrate interface design.

26.3 Initial Machine Requirements Architecture

In general we can always claim that one can continue, after such software design which "implements" domain requirements concerns, with software design concerned with implementing machine requirements.

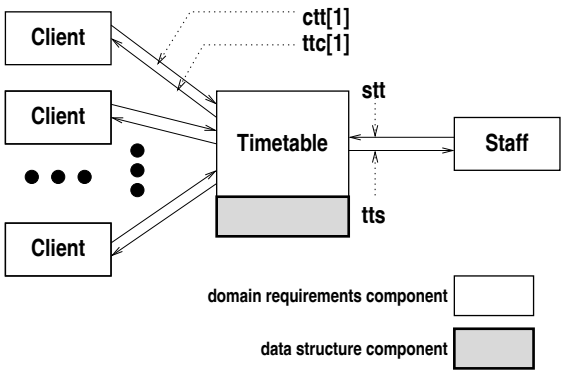


Fig. 26.2. Timetable application components

Example 26.3 *Component Diagram and Formal Model of a Timetable System:* We shall exemplify a software design which can be said to implement domain requirements. We refer to Example 19.32, but now, as it is, with n client processes (Fig. 26.2).

This will be the last figure in Chap. 26 in which we explicitly show data structure components.

Formal Presentation: Timetable Application Components

```
type
  CIdx /* Index set of, say, 1000 terminals */
channel
  { ct[i]:QU,tc[i]:VAL | i:CIdx }
  st:UP,ts:RES
value
  system: TT → Unit
  system(tt) ≡ ||{client(i)|i:CIdx} || time_table(tt) || staff()
```

The individual processes are defined next.

Formal Presentation: Timetable Application Components

```
  client: i:CIdx → out ct[i] in tc[i]  Unit
  client(i) ≡ let qc:Query in ct[i]!M_q(qc) end tc[i]?;client(i)

  staff: Unit → out st in ts  Unit
  staff() ≡ let uc:Update in st!M_u(uc) end let res = ts? in staff() end

  time_table: TT → in {ct[i]|i:CIdx},st  out {tc[i]|i:CIdx},ts  Unit
  time_table(tt) ≡
    [] {let qf = ct[i]? in tc[i]!qf(tt) end | i:CIdx}
    [] let uf = st? in let (tt′,r)=uf(tt) in ts!r; time_table(tt′) end end
```

We refer to Example 19.32.

For the readers who cannot read the formulas we "read" them "aloud": There is an index set of client (names), CIdx. For each client there is a separate pair of channels, ct[i] and tc[i], i.e., means of communicating with the time_table process. This is just a generalisation of the model given in Example 26.2. And, as in that model, there is a pair of channels, st and ts, between the staff and the time_table process. The system is the parallel, comprehended composition of as many client processes as there are elements in CIdx, composed, also in parallel, with one staff and one time_table process. The only difference between the model of the present example and that of Example 26.2 is, first, that communications between client processes and the time_table process

takes place over indexed channels, and, second, that the time_table process nondeterministically, externally, is ready to engage with any client process. ▪

Thus we enter, in the continuing examples of this section, a stage where we are now concerned with the implementation, i.e., the software architectural issues as determined by machine requirements. But first let us exemplify an issue of analysis.

26.4 Analysis of Some Machine Requirements

We have chosen, in this section, to focus on just a few machine requirements issues. Although it may not be realistic of actual developments, it is sufficiently illustrative of what goes on in actual software design developments concerned with the implementation of machine requirements.

The machine requirements issues selected are *performance, availability, accessibility* and *adaptive maintainability*.

26.4.1 Performance

We refer to Examples 19.31 and 19.32. The performance issue chosen was the simple one of making sure that n clients could be online simultaneously. And the software design issue that we wish to look at is how to design a machine requirements component, or a set of such components, that separate out from the time_table process the choice among n client processes and one staff process.

26.4.2 Availability

We refer to Example 19.35. The time_table process does not guarantee "fair" choice between handling input from clients and input from staff processes (f. nondeterministic external choice ($\Box$) of Example 26.3). The internal nondeterministic choice that we are referring to above is that between the timetable process's handling of the n:Index client process inputs and its handling of the staff process inputs. The RSL semantics of $\Box$ allows one side of the $\Box$ operator, i.e., one operand, to be selected indefinitely. "Fairness" is an issue of making sure that both process operands of $\Box$ are "inquired" as to willingness to "progress".[1] That is, both the client and the staff processes should be given a fair chance of communicating with the time_table process.

[1] Our reasoning would be the same for the nondeterministic internal choice operator $\sqcap$.

26.4.3 Accessibility

We refer to Example 19.34. The current software architecture can be said to prescribe strict, mutually exclusive serialisation of client and staff processes wrt. time_table process. This may be acceptable for zero time processing, but it is not acceptable for time-consuming timetable operations (like, for example, connection queries)! "Small, quick" query processing, such as journey, could be interleaved with the processing of "large, time-consuming" queries, such as connections.

26.4.4 Adaptive Maintainability

We refer to Example 19.40. We focus on the direct channels between time_table and the client and staff processes. These direct channels, if also implemented as channels ("ad verbatum"), might hinder the development (i.e., refinement) of several distinct implementations of the client process.

26.5 Prioritisation of Design Decisions

The design decisions include a prioritisation of which machine requirements shall first, then subsequently, determine program organisation design decisions. Our example prioritisation is:

- First *performance*, then
- *availability*, then
- *accessibility*, and finally
- *adaptive maintainability*.

We do not motivate the specific prioritisation. Specific machine requirements prioritise one requirements over another. There may be various reasons for a prioritisation. These prioritisation reasons are usually given in informative documents. We shall not go into a discussion of machine requirements prioritisation.

26.6 Corresponding Designs

So, on one hand, there is the issue of deciding how to suggest a software architecture design (i.e., "a component or two") which implement[s] a machine requirement — a design which can, eventually, be shown, somehow, to satisfy the (usually property-oriented) machine requirements. And, on the other hand, there is the issue of choosing a way in which to represent this design decision.

The emphasis in this section is on expressing designs in terms of diagrams: (i) with some boxes designating domain requirements entities and functions,

i.e., domain requirements components; (ii) with other boxes designating machine requirements design choices, i.e., machine requirements components and (iii) with arrows connecting these boxes designating means of invoking functions in respective components.

26.6.1 Design Decision wrt. Performance

We illustrated in Sect. 26.4.1 just one aspect of the larger machine requirements concern: performance. It was based on Examples 19.31 and 19.32. We shall now illustrate a design decision which is then recorded in the form of a "boxes and arrows" diagram. Throughout this and the next design decisions (Sects. 26.6.2–26.6.4) we shall use this rather informal mode of "design and reasoning", which is based on understanding boxes as usually cyclic processes and arrows as one- or two-directional input/output event channels. That is, we shall omit the crucial specification of the protocols which monitor and control events (synchronisations and communication "along" the arrows) in (and between) processes (i.e., the boxes).

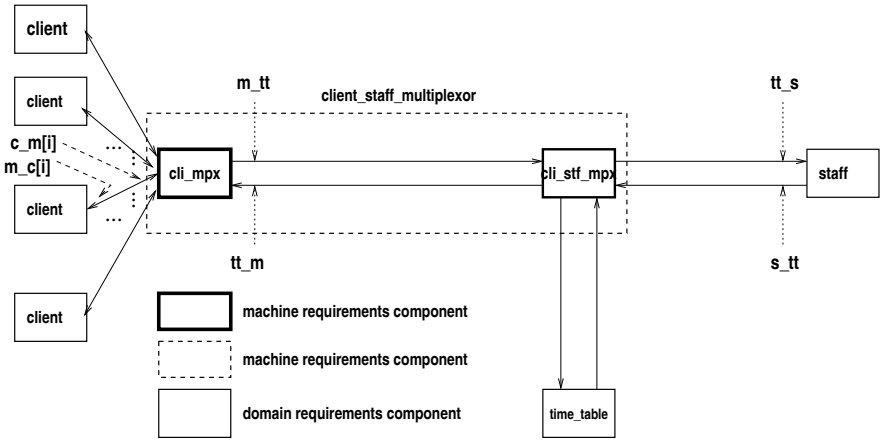


Fig. 26.3. Client/staff multiplexor components

Example 26.4 *Performance Component Design:* We refer to Fig. 26.2. We observe that it is the time_table process which performs the choice between n client requests and one staff request. We decide to "factor" this aspect — of choice — out from the time_table process, and into one machine requirements component, cli_mpx (for client multiplexor component), and one machine requirements component, cli_stf_mpx (for client/staff multiplexor component). We refer to Fig. 26.3. For the moment we might consider the two machine

requirements components as a pair of "inseparable, back-to-back" components. Informally speaking, what goes on in the boxes and on the channels is as follows: Assume a client, say client i, wishes to communicate a query to the timetable. The client multiplexor then decides between this, client i, request and possibly other such, client $j, i \neq j$, requests, and selects one, say client j. The client multiplexor now passes that request on to the client staff multiplexor. Assume that the staff, also at such a time as client requests $i, \ldots, j, \ldots, k$ are issued, issues an update request. Now the client multiplexor decides on one of these two: client j query and the staff update. Assume that the client multiplexor decides to choose the staff request. That staff request is passed on to the timetable, is serviced, and a result (response) is returned to the client staff multiplexor, which returns it to the staff. Right after this the client staff multiplexor may choose to service the client (j) query. It could choose a further staff update should such a request have been issued "right on the heels" of a former and serviced update request. Now we assume that the client staff multiplexor services the ("outstanding") client (j) query. It is passed on to the timetable and is serviced, and a result (value) is returned to the client staff multiplexor, which returns it to the client multiplexor (whereupon the client staff multiplexor is freed to service staff updates). The client multiplexor returns the result of the client j query to that client and the client multiplexor is then freed to service either "outstanding" or new client requests. Notice that at this stage of design we have chosen to let the two multiplexor processes await completion, by them, of the most recently serviced request. ∎

26.6.2 Design Decision wrt. Availability

Example 26.5 *Availability Component Design:* First, we refer to Example 19.35 and then to the discussion of Sect. 26.4.2. To resolve nondeterminism that unfairly chooses among client query requests, as in the client multiplexor, or as among a chosen client request and a staff request, we must modify both multiplexors. Our design choice is to "equip" both multiplexors each with their own arbitration procedure embodied in an *arbiter component*.

Thus these arbiters shall secure a "more fair" choice — perhaps "less nondeterministic" — within and between the two categories of users. A solution could be to have an internal clock (or "bit") secure alternate sampling of client queries (to the client staff arbiter), respectively secure alternate sampling of chosen client and staff requests (to the timetable process). We refer to Fig. 26.4. ∎

Please observe across the two examples just given, Example 26.4 and Example 26.5, that the boxes of a former design decision change "nature" (i.e., meaning) as a result of a subsequent design decision. It is because we wish to

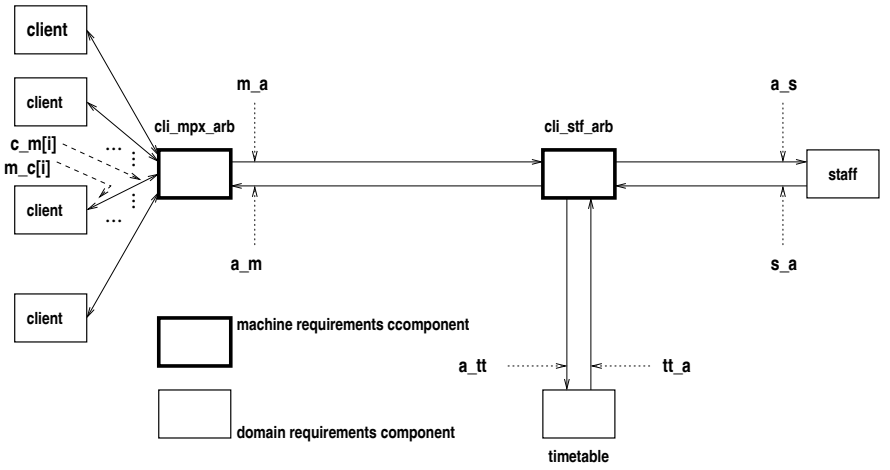


Fig. 26.4. Arbiter component

have the design freedom (i.e., the design options) to let this happen often, that we refrain at this thereby informal step from detailing the "inner workings" of the boxes etc. Now we need only fully specify (incl., possibly formalise) a last design decisions accumulated step.

26.6.3 Design Decision wrt. Accessibility

Example 26.6 *Accessibility Component Design:* First, we refer to Example 19.34 and then to Sect. 26.4.3. The current software architecture (still) prescribes strict, mutually exclusive serialisation of client and staff processes wrt. time_table process. This is not acceptable for time-consuming time_table operations. Our design decision, in this step, is therefore to prescribe that the time_table process is to be a time-shared process. Thus the time_table process is to accept up to several requests, in any order, and to service each request, for example, in some "round robin" fashion, in time slices, such that zero, one or more, but a finite, small number of time slices, allocated to one particular request, produces partial, and eventually full results for that request.

To thus interleave client requests implies the passing of client identity — all the way to the time_table process. There, at the time_table process, they are associated with the time-shared processing, and affixed the partial or full, computed, results when these are being communicated back in order to "sort" out which partial or completed (i.e., full) results "belong to which request".

This is an acceptable solution even when it comes to the design of a time_table process which is as general as possible! Either the operating system handles the time-sharing, and that system then handles the client (and now also staff) request identities, or the timetable subsystem must!

We decide therefore to make sure that two or more client processes can be serviced in overlapping time intervals. This will be effected by a client queue process (cli_q) inserted between the client multiplexor processes (i.e., components) and the client staff arbiter process (i.e., component), and by a client staff queue process (cli_stf_q), i.e., client staff queue component, inserted between the client staff arbiter process (i.e., component) and the time_table process (i.e., component). The latter queue secures that also otherwise serially serviced staff updates can be time-shared and "computed" in a piecemeal fashion. We refer to Fig. 26.5. The previous client multiplexor and arbiter processes (i.e., components) have to be redefined in light of "adding" client and (whether client i or) staff identities to the requests being forwarded to the time_table process.

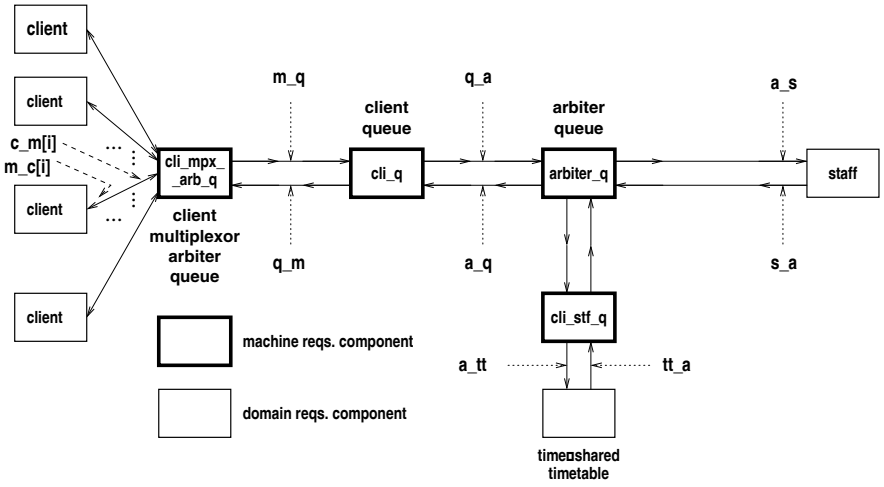


Fig. 26.5. Time-sharing interleave components

A journey query handling by the timetable process could thus be interleaved with the handling of a connection query from another client. The former may "arrive" at the time_table process before the latter, but that process may decide to first service the latter. A more detailed specification of what goes on during the processing of requests can be given:

We first specify the orderly flows of requests from clients and staff towards the time_table process. Thereafter we specify the orderly flows of partial and/or completed results from the time_table process back to respectively client and staff processes. These two flows must themselves be interleaved and not interfere with one another.

Each client request, after having been arbiter-chosen by the client multiplexor arbiter, is annotated with its origin ($client_i$) and passed on to the client queue. Meanwhile the client multiplexor arbiter is freed to accept other requests. The client queue notes that a request is being made by $client_i$ and passes that request on to the client staff arbiter, and is otherwise freed to accept further annotated client requests. The client staff arbiter selects client and staff requests and passes selected such on to the client staff queue, while being freed to arbitrate between subsequent client and staff requests. The client staff queue process notes the identity of the request and passes it on to the time_table process.

In every time-slice that the time_table process has a partial or a completed result associated with a client or a staff request, it returns that result to the client staff queue process. If the result is marked as completing a request, then the client staff queue process removes it from its list of pending requests, as now having been fully serviced by it and the time_table process, while, in any case returning the result to the client staff arbiter. That process, inspecting the identity affixed to the returned result (by the time_table process), decides where to route that result: to the staff process, or to the client staff queue process. In the former case the returned (partial or completed) result has been fully handled. In the latter case the client staff queue process inspects the returned value to see whether it represents a partial or a completed request value. If the latter, then the client staff queue process removes it from its list of pending client requests, as now having been fully serviced by it and the time_table process. In both cases the returned value is returned via the client multiplexor arbiter to the client.

The above scheme allows inspection, at any time, by a *client/staff timetable service system* (which we have not spoken of before) as to the state of pending requests. ▪

26.6.4 Design Decision wrt. Adaptability

Example 26.7 *Adaptive Maintainability Connector Design:* First, we refer to Example 19.40 and then to Sect. 26.4.4. To secure that the developer does not take "white box" advantage of knowledge of how the client, staff, time_table and any other component processes are implemented, it is suggested to insert *connectors* between these and the (newly introduced) arbiter process. The existence of these connectors forces a "standard" ("black box") interface between connected processes. The idea is that the protocol for communication between domain and machine requirements component processes, on one side, and connector processes, on the other side, is made such that previously neighbouring component processes are "effectively" shielded from one another wrt. "white box" knowledge. We refer to Fig. 26.6.

The shaded circles and rounded corner boxes (both kinds without a black edge) designate these connectors. The insertion of connectors between components makes no change to the flow of messages across the network of processes. So we basically inherit the detailed, informal specification that was given for the flow of requests and results across the network of processes as given at the end of Example 26.6.

The meaning of the connectors is subject to a wide range of interpretations. Take an example. Let the double-arrow lines between clients and the client_multiplexor_arbiter stand for wide area communication lines. In some implementation there is trust with respect to these lines: no noise, no intrusion. The connectors on these double-arrow lines can therefore be very simple. In other implementations there is noise, but no threat of intrusion. Now these connectors need to implement some protocol that ensures uncorrupted receipts of messages. In yet other implementations there is threat of intrusion. Now the connectors must implement some encryption scheme. And so forth. ∎

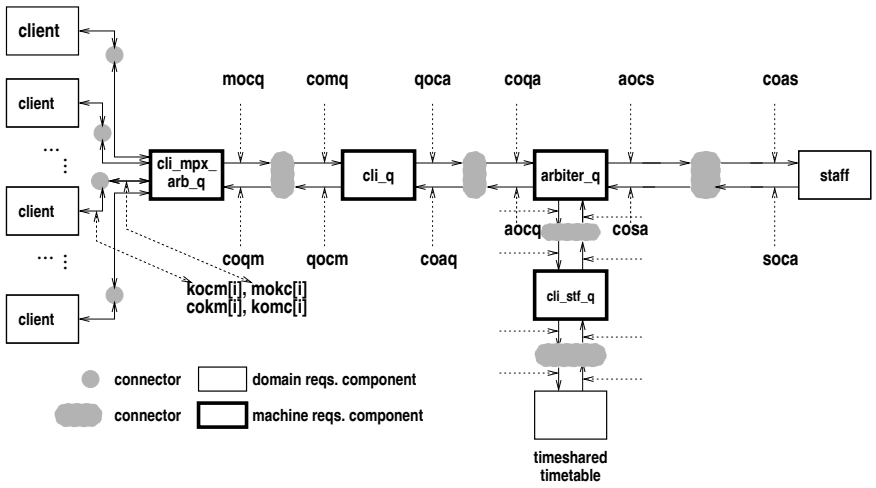


Fig. 26.6. Adaptivity connectors

26.7 Discussion

26.7.1 General

We have concluded a stage of development. From a set of requirements we have developed, informally, a software architecture, a first stage of software

design. This software design is informal, and, in a sense, incomplete. It is incomplete in the sense that we have yet to specify the individual behaviours of each of the domain and machine requirements components as well as of the connector components. It is "complete" in the sense that we now have a "picture" which specifies: There are those and those processes, and there are those and those channels, and no more!

This stage of architecture design deployed a technique — used a design approach — which was informal. It is based on "boxes and arrows". The technique is based on inserting, and perhaps renaming (i.e., redefining) "boxes and arrows". In each design decision step we presented (at the end of the corresponding examples) a detailed specification of how the network of processes was to behave. But only in the last step did we commit ourselves to a specification that eventually has to be implemented.

26.7.2 Principles and Techniques

We summarise:

Principles. The principles of *software architecture* are to sketch a first software design, to handle what is considered the most crucial requirements and to get the long process of software design "on the road". ∎

Principles. The principles of *software architecture design* are basically those of *divide and conquer,* i.e., separation of concerns, of determination of overall component structuring, and hence of determining major interfaces, so as to allow separate design groups to tackle separately the development of components. ∎

Principles. The idea of a *component* is that it serves as a suitably free-standing collection of modules, which together offer either functionalities (i.e., can perform functions), or data storage, or monitoring and/or control of processes, possibly in response to events, in order to implement a clearly identifiable (preferably small set of) requirements. ∎

Principles. Components emerge as a result of software architecture design. Other than that, *component design* follows all the principles of design according to which any software is designed. ∎

Techniques. Major *architecture design techniques* revolve around decomposing the requirements into possibly separable major subsystems (which was not illustrated in this chapter); and deciding, for each such subsystem decomposition, upon definition of resulting interfaces between subsystems, their protocol of communication over these interfaces and the types of data exchanged. Techniques within subsystems also included exploring (experimenting with) different prioritisations of related requirements in order to ascertain

whether one or another prioritisation results in an architecture cum component structure (and interface) design that allows a reasonable sequence of steps of extension, each addressing subsequent, i.e., remaining (still related) requirements. ▪

Subsequent chapters will uncover additional principles and techniques.

26.8 Bibliographical Notes

The Carnegie Mellon University group around David Garlan (G.D. Abowd, R. Allen, M. Shaw, C. Shekaran, and others) has contributed rather significantly to the clarification of many software architecture issues, notably such that relate to components and their connections: [1, 2, 7–9, 113–116, 333].

26.9 Exercises

26.9.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

26.9.2 The Exercises

The last four exercises, i.e., Exercises 26.5–26.8, are specific to this chapter. The first exercises assume that you have selected a topic and are carrying through a series of solutions around this selected topic, i.e., answers to exercises in previous chapters, and now also this chapter.

Exercise 26.1 *Process Software Architecture.* For the fixed topic, selected by you, assume some domain requirements concerning numbers of behaviours (of each class of such behaviours that you can identify in your selected topic). Then diagram, as in Example 26.3 (see Fig. 26.2), these behaviours as a first attempt at a software architecture design.

Hint: You may seek inspiration from the diagrams of Sect. 11.2.3 (i.e., Figs. 11.1–11.47).

Exercise 26.2 *Implementing Accessibility.* For the fixed topic, selected by you, postulate some meaningful accessibility (i.e., machine dependability) requirements, and then suggest a restructuring of your solution to Exercise 26.1.

Exercise 26.3 *Implementing Availability.* For the fixed topic, selected by you, postulate some meaningful availability (i.e., machine dependability) requirements, and then suggest a restructuring of your solution to Exercise 26.2.

Exercise 26.4 *Implementing Adaptability.* For the fixed topic, selected by you, postulate some meaningful adaptability (i.e., machine maintenance) requirements, and then suggest a restructuring of your solution to Exercise 26.3.

Exercise 26.5 *Performance Component Design.* We refer to Example 26.4. Please formalise the model implied by that example.

Exercise 26.6 *Availability Component Design.* We refer to Example 26.5. Please formalise the model implied by that example.

Exercise 26.7 *Accessibility Component Design.* We refer to Example 26.6. Please formalise the model implied by that example.

Exercise 26.8 *Adaptive Maintainability Component Design.* We refer to Example 26.7. Please formalise the model implied by that example.

27

A Case Study in Component Design

- The **prerequisite** for studying this chapter is that you are reasonably fluent in reading, if not in writing, formal specifications — as this chapter is optimally appreciated when one also studies its rigorous development. But the chapter can be read — and an essence conveyed — by skipping the framed formalisations.
- The **aims** are to illustrate the gradual introduction of additional user functionalities when securing a software system's robustness requirements, to thus show that "new requirements" may "pop up" as a result of software design, and to illustrate data refinement techniques, abstraction functions and one kind of correct criteria.
- The **objective** is to make you versatile in your ability to design systems.
- The **treatment** is from rigorous to formal.

27.1 Overview Introduction

This section presents one large example. In it we show how a software architecture — which satisfies some initial domain requirements — is developed in steps alternating with the development of a component structure. This component structure satisfies some further machine requirements, requirements that are not really "discovered" till "halfway" through architecture design. The example also illustrates the use of *data refinement* techniques for the purpose of conquering the architectural complexity of a system.

27.1.1 System Complexity

That is, we are oftentimes faced with the problem of having to design a system with very many properties, too many to be grasped in any one presentation. Instead we show a technique whereby the architectures, i.e., the full variety of all properties, can be stepwise developed. From a small architectural

specification — exemplifying what is considered the very basic properties — one arrives, in steps, at increasingly more complex designs. At each step a new, small set of properties is "added" to the previous description.

Sometimes software systems contain unnecessarily many, seemingly independent concepts. Occasionally a large number of such concepts are, however, necessary. Their presence is required in order to cope with varieties of domain requirements, interface requirements and, especially, machine requirements.

In all cases it is rather hard to grasp all the concepts, sort them out and interrelate them properly. In many cases this ability to dissect a software architecture into its many constituent notions is seriously hampered by opaque presentations of their interdependencies.

27.1.2 Proposed Remedies

> You can design software in either of two ways:
> Either you make it so clear and simple such that it obviously has no bugs,
> of you make it so complex such that it has no obvious bugs.
>
> *Sir Tony Hoare*

Three possibilities for "solving" the apparent complexity problem exist: two extremes, and a "middle road". These choices are either not to design such multiconcept systems at all, or go on designing them in the old "hacker" fashion. We shall sometimes choose the first extreme, sometimes the "middle road" approach outlined below, but never the second "compromise"!

27.1.3 Stepwise Development

Characterisation. By *stepwise development* — other terms, used interchangeably, are *stepwise refinement, stepwise reification, stepwise transformation* — of a software design, we shall understand the following: First a model is established which exhibits, as abstractly as deemed reasonable, the intrinsic concepts and facilities for which the software was intended, that is, a model which satisfies the domain requirements. Then this model is subjected to *data* and/or *operation refinement*. The choice of refinements is determined so as to satisfy those domain requirements which were not all taken care of in the first step, and so as to — similarly — satisfy (remaining) interface requirements, and so as to satisfy machine requirements. ▪

We shall, in this chapter, emphasize variations of *data refinement* and *data reification* referred to by the collective term *data transformation*. A sequence of transformations may be needed. Each step introduces further properties and/or details, none, some or all of which are exploited in exposing them to an external world. The order of the steps and their nature is dictated, for example, by technological and/or product strategic considerations.

27.1.4 Stagewise Iteration

By stagewise iteration — other terms, used interchangeably, are stagewise evolution or stagewise spiralling — of a software design, we shall understand the following:

- *One or more steps of development within a stage, s, are performed.* ("Start" with stage s.)
- *Then one or more steps of development within a next stage, s', are performed.* (Forward to stage s'.)
- *Then one or more steps of development within stage s are performed.* (Back to stage s.)
- *And so on, alternating between stages $s, s', s'', \ldots, s'''$.* (Iterating, forward and backward.)

Our example exhibits stagewise iteration.

27.2 Overview of Example

Our example is that of a file-handler system:

0. At the top level (step 0) of architecture we focus our attention on files, file names, pages and page names as *data* and the creation, and erasure, of files, and the writing, updating, reading, and deletion of pages as *operations*. At this step files are named and consist of named pages.

At the top level no concession is made to the possible storing of files and their pages in such diverse storage media as foreground (fast access, "core"), or background (slow access, "disk") storage. The decision, which is hence recorded, of eventually implementing the storing of files and pages on disk-like devices, predicates a need to be able to "look up", reasonably fast, where, on possibly several disks, files and pages are stored.

1., 2. In the next two steps we therefore introduce first the notions of catalogues and directories, and, subsequently, as a further step of development, abstractions of the notions of main storage and disks.

Catalogues eventually record disk addresses of file directories, one per file. Directories eventually record disk addresses of pages. Our file system at this level has one catalogue. We think, at the level of main storage and disks, of the one catalogue as always residing in main storage, whereas all directories are normally only stored on disks.

To speed up access to disk pages we operate on main storage copies of directories. The intention to operate on a file is then indicated by its opening, which is an "act" that brings a disk directory copy into main storage. The intention to not operate further on a file is then indicated by its closing, an "act" which reverses the above copying.

3. Hence open and close operations are introduced in step 3.

Opening and closing are file-related concepts primarily brought upon us by *efficiency* considerations. These efficiency concerns are rooted in insufficient technologies. Thus they represent machine performance requirements.

Neither at the top, nor at the second level (i.e., in steps 2 and 3) of the file-handler "architecting", did we bother about the machine requirements issue of *reliability*. We here define the reliability of our file handler as its ability to survive *crashes*.

By a "crash" we restrictively mean anything which renders main storage information (catalogue and opened directories) useless. By total "survival" we mean the ability to continue (some time) after a "crash" as if no "crash" had occurred. By "partial survival" we mean the ability to continue with at least a nonvoid subset of the files after a "crash" — with the complement set of files being clearly identified.

4. In the fourth step, building upon redundancies in catalogue, directory and page recordings, we therefore introduce notions of *checkpointing* files and automatic *recovery* from "crashes".

5.–6. Final steps — as presented here — hint at space management of storage and disk: We introduce free lists of unused, available disk storage, etc.

27.3 Methodology Overview

We paraphrase the above by giving an overview of the principles and techniques to be deployed.

27.3.1 Principles

We can summarise the principles as follows:

1. *Stepwise unfolding of software architecture:* Instead of going, in "one fell swoop", from (all of) the requirements to (all of) the architecture, we decompose this stage of the development of the software architecture of a simple file handler into steps, each step taking care of one concern.
2. *Interweaving domain and machine requirements implementation:* Instead of taking care, first, of all domain requirements, we alternate between considering domain and machine requirements.
3. *Invariance:* Usually abstract type definitions, i.e., sorts, are subject to axioms which express properties of the type, i.e., of its values. These axiomatic sort properties are usually expressed in relation to (i.e., in terms of) the various functions that are applicable to (otherwise well-formed) values of the type.
   In stage- and stepwise refinement one usually represents abstract (i.e., sort) types in terms of concrete types (e.g., sets, Cartesians, lists, maps,

etc.). In doing so the concrete type is usually capable of expressing "more" values than are desired, i.e., values that do not properly represent any corresponding abstract (sort) value — which was and is the idea. Hence we need to express invariance of values of a type, i.e., a subtype. We do so by defining explicit invariance predicates.

4. *Abstraction, adequacy and sufficiency:* While adhering to the above principles, we also adhere to principles of considering functions that abstract from later design steps "back" to earlier design steps, or that express the adequacy of a representation, that is, of a later design step (i.e., of a design decision), wrt. an earlier, or that express the sufficiency of a representation.
5. *Correctness:* When "performing" a step of development, from a "more" abstract to a "more" concrete design, one has to argue why the chosen step implements the abstraction. Usually a formal proof is required. Often an informal, but precise reasoning is sufficiently convincing.

27.3.2 Techniques

We summarise the techniques, referring to Sect. 27.2, as they are invoked in different steps:

- Intrinsic domain requirements:
  - ⋆ Step 0: Intrinsic architecture: (Sect. 27.4)
    files, create, erase, pages, write, update and delete

  A prescription is given of what we could consider the domain requirements of the file handler. At the same time we might consider this prescription to also be a specification of the basic software architecture from which we further develop the full software architecture. Had we, instead, prescribed the domain requirements by means of sorts, of the signatures of the file handler command functions, and of axioms over these, then we could say that the specification given below is truly that of a, or the, basic software architecture.
- Machine requirements:
  - ⋆ Step 1: Catalogue (Sect. 27.5)
  - ⋆ Step 2: Disk (Sect. 27.6)

  Two steps of data refinement now follow: We choose a "more" concrete representation (of the system) than given in step 0; we define an invariance predicate; we redefine the file handler command functions; and we express adequacy, sufficiency and correctness. We do not prove correctness.
- Domain and machine requirements:
  - ⋆ Step 3: open and close commands (Sect. 27.7)

  We argue design decisions based on considerations of technology, and, accordingly (again) choose an "even more" concrete representation of the file-handler system than presented in step 1. We then define invariance

and abstraction functions, redefine some of the file-handler command functions, and leave the expression of adequacy, sufficiency and correctness to the reader. Again we do not prove correctness.

- Detailed component structure:
  - ⋆ Step 4: Crash robustness: check and crash (Sect. 27.8)
  - ⋆ Step 5: "Flat" storage (Sect. 27.9)
  - ⋆ Step 6: Space management (Sect. 27.10)

27.4 Step 0: Files and Pages

The next four subsections present an abstraction of a file system architecture.

27.4.1 A "Snapshot"

Figure 27.1 abstracts a file system of three files named f1, f2 and f3. The first file contains two pages, the second is empty and the third file contains three pages.

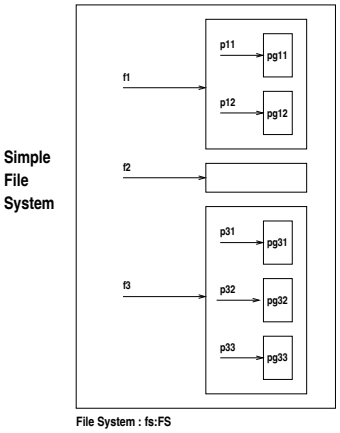


Fig. 27.1. Base file system

27.4.2 An Abstract Formal Model

Based on the immediately following English wording of what the type of the state of our top-level file-handler is, we "derive" informally the formal type definitions.

The sole data structure of our file-handler consists of a set of uniquely named files. Each file consists of a set of uniquely named pages. Let *Fn*, *Pn*

and *PAGE* denote the further unspecified types of respectively file names, page names and pages. Then:

Formal Presentation: Step 0: Architecture: Files and Pages

```
type
    [Step 0]
  FileSystem, File, Fn, Page, Pn, FILE, PAGE
  FS0 = Fn ~m~> FILE
  FILE = Pn ~m~> PAGE
value
  obs_Fns: FileSystem → Fn-set
  obs_Files: FileSystem ~→ File-set
  obs_FS0: FileSystem ~→ FS0
  obs_Fn: File ~→ Fn
  obs_FILE: File ~→ FILE
  obs_Pns: Page ~→ Pn-set
  obs_Pages: File ~→ Page-set
  obs_Pn: Page ~→ Pn
  obs_PAGE: Page ~→ PAGE
```

We have completed our first task: that of specifying the most important aspects first, namely the semantic types. We need to express an invariance: The file names of the system are those of the files of the system. And the page names of the system are those of the pages of the system.

Continuation – Formal Presentation: Step 0: Architecture: Files and Pages

```
axiom
  ∀ fs:FileSystem •
    let fns = obs_Fns(fs), files = obs_Files(fs) in
    fns = { obs_Fn(f) | f:File • f ∈ files } ∧
    ∀ f:File • f ∈ files •
      let pns = obs_Pns(f), pages = obs_Pages(f) in
      pns = { obs_Pn(p) | p:Page • p ∈ pages }
    end end
```

27.4.3 Abstract Versus Concrete Basic Actions

To *create* an initially empty file (of no pages) we need to specify a new, hitherto unused file name. To *erase* an existing file we need to specify the name of a file already in the system. To put a page into a file we need to

specify the names of the file and page, and the page itself. To *get* a page from a file we to need specify the names of the file and page. Finally, to *delete* a page we need to specify the same:

Formal Presentation: Abstract Versus Concrete Basic Actions

- Create file:
  - ⋆ **Abstract:**

    value
    crea: Fn $\xrightarrow{\sim}$ FileSystem $\xrightarrow{\sim}$ FileSystem
    crea(fn)(fs) **as** fs′
    pre: fn ∉ obs_Fns(fs)
    post: obs_FS0(fs′) = obs_FS0(fs) ∪ [fn ↦ [ ]]

    or:

    axiom
    empty: File → **Bool**
    empty((crea(fn)(fs))(fn)),
    ∼empty((crea(fn)(put(fn,pn,pg)(fs)))),
    undef(empty((crea(fn)(eras(fn)(fs)))))
  - ⋆ **Concrete:**

    value
    crea0: Fn $\xrightarrow{\sim}$ FS0 $\xrightarrow{\sim}$ FS0
    crea0(fn)(fs) ≡ fs ∪ [fn ↦ [ ]]
    pre: fn ∉ obs_Fns(fs)
- Put file:
  - ⋆ **Abstract:**

    value
    put: Fn × Page $\xrightarrow{\sim}$ FileSystem $\xrightarrow{\sim}$ FileSystem
    put(fn,pg)(fs) **as** fs′
    pre: fn ∈ obs_Fns(fs)
    post: **let** cfs = obs_FS(fs), cfs′ = obs_FS(fs′),
    pn = obs_Pn(pg), cpg = obs_PAGE(pg),
    cfile = obs_FILE((obs_FS(fs))(fn)) **in**
    cfs′ = cfs † [fn ↦ pgs † [pn ↦ cpg]] **end**

    or:

    axiom
    get(fn)(put(fn,pn,pg)(fs)) = pg,
    undef(get(fn)(del(fn)(fs)))
  - ⋆ **Concrete:**

    value
    put0: Fn × Pn × PAGE $\xrightarrow{\sim}$ FS $\xrightarrow{\sim}$ FS

put0(fn,pn,pg)(fs) ≡ fs†[fn ↦ fs(fn)†[pn ↦ pg]]
pre: fn ∈ **dom** fs

27.4.4 Concrete Actions

Formal Presentation: Concrete Actions

value
  eras0: Fn $\xrightarrow{\sim}$ FS0 $\xrightarrow{\sim}$ FS0
  eras0(fn)(fs) ≡ fs \ {fn}
    pre: fn ∈ **dom** fs

  get0: Fn × Pn $\xrightarrow{\sim}$ FS0 $\xrightarrow{\sim}$ PAGE
  get0(fn,pn) ≡ (fs(fn))(pn)
    pre: f ∈ **dom** fs ∧ p ∈ **dom** (fs(fn))

  del0: Fn × Pn $\xrightarrow{\sim}$ FS0 $\xrightarrow{\sim}$ FS0
  del0(fn,pn)(fs) ≡ fs † [fn ↦ (fs(fn)) \ {pn}]
    pre: f ∈ **dom** fs ∧ p ∈ **dom** (fs(fn))

We have completely specified the basic, major functions of a simple file handler system. The abstraction is just that: We have abstracted from any concern about how actual input of commands, including input of pages, and of how output of pages take place. We have also abstracted "away" considerations of what kind of diagnostics to use in case of erroneous input — we have only defined, in preconditions, what we mean by erroneous input. We have abstracted from any representation of files, and, in fact, the entire file system. Finally, we have not been, and shall not, in this entire example, be concerned with what pages are.

27.5 Step 1: Catalogue, Disk and Storage

We divide the next development into three steps. First, we introduce the data notions of *catalogues* and *directories*, then the data notion of *disk*, and finally the data notions of *storage* and *disk*. The single aim of this level is to introduce the operation notions of *open* and *close*.

27.5.1 Catalogue Directories

Figure 27.2 instantiates a first step of concretisation of Fig. 27.1. A catalogue and some file directories have been "inserted" as a means of obtaining access to the file pages.

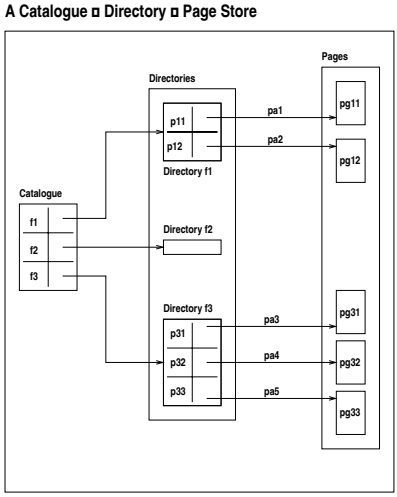


Fig. 27.2. Catalogue + directories + pages

Data Structure

We now adorn our type names according to the number of the step of development. The zeroth step (which was the top level) gave us FS0.

To each file in FS1 we now associate a page directory. Each directory records where pages are stored. Directories are named, and these names are recorded in a catalogue.

Formal Presentation: Catalogue + Directories + Pages

type

[Step 0]
FS0 = Fn $\overrightarrow{m}$ (Pn $\overrightarrow{m}$ PAGE)

[Step 1]
FS1 = CTLG1 × DIRS1 × PGS1

You may (justifiably) think of directories "translating" user-oriented page names into system-oriented page addresses, and *PGS1* to be a disk-like space within which all pages of all files are allocated. Let:

$$\left[\begin{array}{l} f_1 \mapsto \left[\begin{array}{l} p_{11} \mapsto g_{11}, \\ p_{12} \mapsto g_{12} \end{array}\right], \\ f_2 \mapsto \left[p_{21} \mapsto g_{21}\right], \\ f_3 \mapsto [\,] \end{array}\right]$$

be an abstract, *FS0*, file system. Its counterpart in *FS1* is:

$$\left(\begin{array}{c} \left[\begin{array}{l} f_1 \mapsto d_1, \\ f_2 \mapsto d_2, \\ f_3 \mapsto d_3 \end{array}\right], \\ \left[\begin{array}{ll} d_1 \mapsto & \left[\begin{array}{l} p_{11} \mapsto a_{11}, \\ p_{12} \mapsto a_{12} \end{array}\right], \\ d_2 \mapsto & \left[p_{21} \mapsto a_{21}\right], \\ d_3 \mapsto & [\,] \end{array}\right], \\ \left[\begin{array}{l} a_{11} \mapsto g_{11}, \\ a_{12} \mapsto g_{12}, \\ a_{21} \mapsto g_{21} \end{array}\right], \end{array}\right)$$

Invariant

The type definitions define too much. Not all combinations of catalogues, directories and pages go together. We must require that there is a distinct directory in *DIRS1* for each file catalogued in *CTLG1;* that pages addressed in *PGS1* are actually recorded in directories; and that every page, understood as page-address, is described in exactly one directory (that is belongs to exactly one file).

Formal Presentation: Catalogue + Directory + Page Invariants

```
type
    [Step 0]
  FS0 = Fn ~m~> (Pn ~m~> PAGE)

    [Step 1]
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1
  CTLG1 = Fn ~m~> Dn
  DIRS1 = Dn ~m~> DIR1
  DIR1 = Pn ~m~> Pa
  PGS1 = Pa ~m~> PAGE

value
  inv_CTLG1: CTLG1 → Bool
  inv_CTLG_1(ctlg) ≡ card dom ctlg = card rng ctlg

  inv_DIRS1: → Bool
  inv_DIRS1(dirs) ≡
    card dom dirs = card rng dirs ∧
    ∀ dir:DIR1 • dir ∈ rng dirs ⇒ inv_DIR1(dir)

  inv_DIR1: → Bool
  inv_DIR1(dir) ≡ card dom dir = card rng dir
```

```
inv_PGS1: → Bool
inv_PGS1(pgs) ≡ card dom pgs = card rng pgs

inv_FS1: FS1 → Bool
inv_FS1(ctlg,dirs,pgs) ≡
    inv_CTLG_1(ctlg) ∧inv_DIRS1(dirs) ∧ inv_PGS1(pgs) ∧
    rng ctlg = dom dirs ∧
    ⋃ { rng dir | dir:DIR1 • dir ∈ rng dirs } = dom pgs ∧
    ∀ pa:Pa•pa ∈ dom pgs•∃! dn:Dn•dn ∈ dom dirs•pa ∈ rng dirs(dn)
```

Annotations:

- The more concrete catalogue is well-formed if it is a bijection: To each file name there corresponds a unique directory name.
- The collection of more concrete directories is well-formed if it is a bijection: To each directory name there corresponds not only a unique directory, but each of these directories is well-formed, i.e., is also a bijection. Directories map each page name to a unique page.
- The collection of more concrete pages is well-formed if it is a bijection: To each page address there corresponds a unique page.

The previous three items were concerned only with the well-formedness of respective components of the overall more concrete file system. What is missing, namely the constraints that "cut across" the triplet structure is now formulated:

- The more concrete file system is well-formed:
  - ⋆ if each of its parts is well-formed,
  - ⋆ if the directory names mentioned in the catalogue correspond exactly to those mentioned in the collection of directories, and
  - ⋆ if, for every page address in the collection of pages, there exists a unique directory name in whose directory that page address is mentioned.

27.5.2 Abstraction

Given an *FS1* file system we can abstract a "corresponding" *FS0* from it. Abstraction is a function.

Formal Presentation: Catalogue + Directory + Page Abstraction

```
type
    [Step 0]
  FS0 = Fn ~m~> (Pn ~m~> PAGE)

    [Step 1]
```

```
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1
  CTLG1 = Fn ~m~> Dn
  DIRS1 = Dn ~m~> DIR1
  DIR1 = Pn ~m~> Pa
  PGS1 = Pa ~m~> PAGE

value
  abs_FS0: FS1 ~~> FS0
  abs_FS0(ctlg,dirs,pgs) ≡
    [ fn ↦ [ pn ↦ pgs((dirs(ctlg(fn)))(pn))
                | pn:Pn • pn ∈ dom dirs(ctlg(fn)) ]
      | fn:Fn • fn ∈ dom ctlg ]
  pre: inv_FS1(ctlg,dirs,pgs)
```

We can only retrieve (i.e., abstract) well-formed file systems.

- To abstract, i.e., to retrieve, from a more concrete file system, its abstract counterpart is
  - ⋆ for every file name, in the concrete catalogue,
  - ⋆ to reconstruct named pages:
    - · namely, for every page address in the directory for that file
    - · to map it into its page in the collection of pages.

As an aside: We could also, in addition to abstraction functions, define their "inverse", injection functions:

```
type
  A, B
value
  wf_A: A → Bool, wf_B: B → Bool
  abs_A: B ~~> A, inj_B: A ~~> B-infset
axiom
  ∀ a:A • wf_A(a) ⇒ ∀ b:B • wf_B(b) ⇒ b ∈ inj_B(a) ⇒ abs_A(b)=A
```

27.5.3 Actions

We rewrite the action functions in terms of the new semantic types:

Action Signatures

Formal Presentation: Action Signatures

```
value
```

```
crea1: Fn → FS1 ~→ FS1
eras1: Fn → FS1 ~→ FS1
put1: (Fn × Pn × PAGE) → FS1 ~→ FS1
get1: (Fn × Pn) → FS1 ~→ PAGE
del1: (Fn × Pn) → FS1 ~→ FS1
```

There are five commands: crea1, eras1, put1, get1 and del1 (create, erase, put, get and delete). To create a file all the syntax that is needed is a new file name. To do the update requires the entire file system — and results in a new file system. We leave the "reading" of the remaining signatures for the reader to decipher.

Create and Erase File Actions

Formal Presentation: Create and Erase File Actions

```
value
  crea1(fn)(ctlg,dirs,pgs) ≡
    let d:D • d ∉ dom dirs in (ctlg ∪ [fn↦d],dirs ∪ [d↦[]],pgs) end
  pre: f ∉ dom ctlg

  eras1(fn)(ctlg,dirs,pgs) ≡
    (ctlg \ {fn},dirs \ {ctlg(fn)},pgs \ rng dirs(ctlg(fn)))
  pre: f ∈ dom ctlg
```

To create a named file is to "fetch" a new directory name, to let that directory name be the designation of the file name in the catalogue, to initialise the named directory to an empty such, and to not change the collection of pages.

Put Page Action

Formal Presentation: Put Page Action

```
value
  put1(fn,pn,pg)(ctlg,dirs,pgs) ≡
    if pn ∈ dom dirs(ctlg(fn))
      then
        (ctlg,dirs,pgs † [(dirs(ctlg(fn)))(pn) ↦ pg])
      else
        let pa:Pa • pa ∉ dom pgs in
        let dirs′ = dirs ∪ [ctlg(fn)↦(dirs(ctlg(fn)))]∪[pn↦pa],
```

```
            pgs′ = pgs ∪ [pa ↦ pg] in
        (ctlg,dirs′,pgs′) end end end
  pre: f ∈ dom ctlg
```

To put a page into the file system is to overwrite that named page in the collection of pages if there was already one by that name (and hence address). Otherwise it is to "fetch" a new page address, to extend the appropriate directory with the page name to page address association, and then to extend the collection of pages accordingly. The former is an update, and the latter is a write.

Get and Delete Page Actions

Formal Presentation: Get and Delete Page Actions

```
value
  get1(fn,pn)(ctlg,dirs,pgs) ≡ pgs(dirs(ctlg(fn))(pn))
  pre: f ∈ dom ctlg ∧ pn ∈ dom(dirs(ctlg(fn)))

  del1(fn,pn)(ctlg,dirs,pgs) ≡
    (ctlg,
     dirs † [ctlg(fn) ↦ (dirs(ctlg(fn))) \ {pn}],
     pgs \ {(dirs(ctlg(fn)))(pn)})
  pre: f ∈ dom ctlg ∧ pn ∈ dom(dirs(ctlg(fn)))
```

To get a named page from a named file is to look it up in the collection of pages under the address in the directory as so directed by the catalogue. To delete a named page from a named file is to remove the page name to page address association from the directory and the page from the collection of pages.

27.5.4 Adequacy and Sufficiency

Correctness of the above realisations of the semantic actions with respect to the realisation of *FS0* in terms of *FS1* is expressed by means of the (i.e., an) abstraction function.

We "divide" our correctness concern into three parts: adequacy of chosen concrete data representation, sufficiency of the same, and correctness of each concrete action specification with respect to the corresponding abstract action specification.

Adequacy

Formal Presentation: Adequacy

```
axiom
    ∀ fs0:FS0, ∃ fs1:FS1 • inv_FS1(fs1) ⇒ fs0 = abs_FS0(fs1)
```

A concrete file system model is adequate with respect to an abstract file system model if for every abstract file system there corresponds a more concrete well-formed one which abstracts, i.e., which retrieves to the abstract file system.

Sufficiency

Formal Presentation: Sufficiency

```
axiom
    ∀ fs1:FS1 • inv_FS1(fs1) ⇒ abs_FS0(fs1) ∈ FS0
```

A concrete file system model is sufficient with respect to an abstract file system model if every well-formed concrete file system abstracts to an abstract file system.

27.5.5 Correctness

Comparable Results

To express correctness of concrete action specifications with respect to concrete action specifications we need to define an abstraction function on results (RES).

Formal Presentation: Comparable Results

```
type
      [Step 0]
    RES0 = FS0 | PAGE
      [Step 1]
    RES1 = FS1 | PAGE
value
    abs_RES0: RES1 ~→ RES0
    abs_RES0(r) ≡
      if r ∈ FS1
```

then if inv_FS1(r) **then** abs_FS0(r) **else** undef **end**
else r **end**

The abstraction of a result which is an entire concrete file system requires that that concrete file system is invariant, i.e., well-formed, and is then its abstraction. Concrete pages do not differ, in this development, from abstract pages.

The Correctness Statement

Formal Presentation: Correctness

axiom
[adequacy] ∧ [sufficiency] ∧
abs_RES0(crea1(fn)fs1) = crea0(fn)fs0 ∧
abs_RES0(eras1(fn)fs1) = eras0(fn)fs0 ∧
abs_RES0(put1(fn,pn,pg)fs1) = put0(fn,pn,pg)fs0 ∧
abs_RES0(get1(fn,pn)fs1) = get0(fn,pn)fs0 ∧
abs_RES0(del1(fn,pn)fs1) = del0(fn,pn)fs0

Where '=' "extends" to: undef = undef!

Correctness of this step of development is now that the semantic types, i.e., concrete data representation, at this step, are adequate, that its concrete data representation is sufficient, and that every concrete operation yields a result which is comparable to that of the corresponding abstract operation. This can be diagrammed as the commutation of two algebras. See Fig. 27.3.

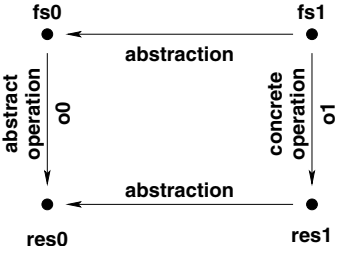


Fig. 27.3. Correctness: commutation of two algebras (FS0, FS1)

Let us express the correctness theorem a bit more precisely. For every pair of abstract (fs0:FS0) and concrete (fs1:FS1) file systems such that the concrete one abstracts to the abstract one, it shall be the case that for respective, and corresponding operations (o0, o1) that the results (o0(fs0), o1(fs1)) are comparable.

27.6 Step 2: Disks

27.6.1 Data Refinement

The data refinement of this step involves the "gathering" (into one component of *FS2) of directories and pages, that is, of the above DIRS1 and PGS1 components of FS1, called DSK2. DIRS1* and *PGS1* are modelled as maps, and *DSK2* will hence be a "merged" type of similar maps. Where before catalogue and directory map range types were directory names, respectively page addresses:

$$\left(\begin{bmatrix} f_1 \mapsto d_1, \\ f_2 \mapsto d_2, \\ f_3 \mapsto d_3 \end{bmatrix}, \begin{bmatrix} d_1 \mapsto \begin{bmatrix} p_{11} \mapsto a_{11}, \\ p_{12} \mapsto a_{12} \end{bmatrix}, \\ d_2 \mapsto \begin{bmatrix} p_{21} \mapsto a_{21} \end{bmatrix}, \\ d_3 \mapsto [\,] \end{bmatrix}, \begin{bmatrix} a_{11} \mapsto g_{11}, \\ a_{12} \mapsto g_{12}, \\ a_{21} \mapsto g_{21} \end{bmatrix}\right)$$

The "merged" (or "gathered") type will only have addresses in its map definition set. We think of *DSK2* as modelling "actual" disks.

27.6.2 Disk Type

A "Snapshot"

Figure 27.4 is intended to show a "monolithic" state which consists of three components: catalogue, disk directories and disk pages. The catalogue is intended to be (foreground) storage-bound, whereas the directories and pages are to be disk-bound — as shown by the rectangle drawn around the latter. The formalisation captures this grouping.

27.6.3 FS0, FS1 and FS2 Types

Concrete Semantic Types

Formal Presentation: Concrete Semantic Types

type
  [Step 0]
  Fn, Pn, PAGE
  FS0 = Fn $\overset{m}{\rightarrow}$ (Pn $\overset{m}{\rightarrow}$ PAGE)
  [Step 1]
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1

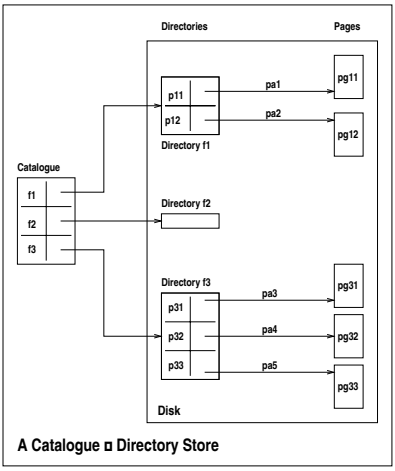


Fig. 27.4. A "snapshot"

CTLG1 = Fn $\overrightarrow{m}$ Dn
DIRS1 = Dn $\overrightarrow{m}$ DIR1
DIR1 = Pn $\overrightarrow{m}$ Pa
PGS1 = Pa $\overrightarrow{m}$ PAGE
[Step 2]
Adr = Dn | Pa
FS2 = CTLG2 × DISK2
CTLG2 = Fn $\overrightarrow{m}$ Adr
DISK2 = Adr $\overrightarrow{m}$ (DIR2 | PAGE)
DIR2 = Pn $\overrightarrow{m}$ Adr

that is:

type
DISK2 = (Dn $\overrightarrow{m}$ DIR2) $\bigcup$ (Pa $\overrightarrow{m}$ PAGE)

Here addresses *Adr* (like file names, *Fn*, and page names, *Pn*, and pages, *PAGE*) are further undefined. The $\bigcup$ operator on map types is not proper RSL, but could have been so without much trouble.

27.6.4 Disk Type Invariant

Again, the type definitions define too much. In addition to the invariants ["carried over" from the very similar definitions of *FS1*], we must (first) make sure that directory addresses (listed in the catalogue) really denote directories on the disk, respectively that page addresses listed in directories really denote pages on the disk. Once this is established we can retrieve *FS1* data from such

"tentatively well-formed" *FS2* data, and this abstracted data must satisfy the earlier stated constraints.

Formal Presentation: Disk Type Invariant

```
value
  inv_FS2: FS2 → Bool
  wf_Dirs: FS2 → Bool

  inv_FS2(fs2) ≡ wf_Dirs(fs2) ∧ inv_FS1(abs_FS1(fs2))

  wf_Dirs(ctlg,disk) ≡
    ∀ a:Adr • a ∈ rng ctlg
      ⇒ a ∈ Dn ∧ disk(a) ∈ DIR2 ∧ ∀ a′:Adr • a′ ∈ rng disk(a)
        ⇒ a′ ∈ Pa ∧ disk(a′) ∈ PAGE
```

27.6.5 Disk Type Abstraction

We leave as an exercise to narrate the abstraction function from FS2 to FS1. But here, at least, is the formalisation:

Formal Presentation: Disk Type Abstraction

```
value
  abs_FS1: FS2 ~→ FS1
  abs_FS1(ctlg,disk) ≡
    (ctlg,[ a ↦ disk(a) | a:Adr • a ∈ rng ctlg ],disk \ rng ctlg)
```

27.6.6 Adequacy, Sufficiency, Operations and Correctness

We leave as an exercise to define adequacy and sufficiency; semantic actions: crea2, eras2, put2, get2, and del2; and correctness.

27.7 Step 3: Caches

27.7.1 Technology Considerations

We enumerate some technology constraints as they help motivate our next design decisions.

- Storage space is expensive. Disk space is less so.

- Storage access is fast. Disk access is less so.
- Hence some data are in storage; most are on disk.
- Hence accessible data must first be "opened".

We shall then see (i.e., next) our "design decision response" to the above technology constraints.

27.7.2 Cached Directory and Page Access

We now face the reality of storages and disks. By a storage we shall understand a memory medium for which access to information is orders of magnitude faster than to information on what we shall then call disks! Access to pages (on disk) goes via catalogue and directories, where the latter are also on disk. Thus two disk accesses per page access. (In this discussion we think of the catalogue as residing in storage.) To cut down on disk accesses we therefore decide to copy into storage the directories of those files whose pages we wish to access. In the resulting model all pages will still be thought of as stored only on the disk.

Figure 27.5 shows how some directories are opened, that is, are cached (hence copied) in fast access storage.

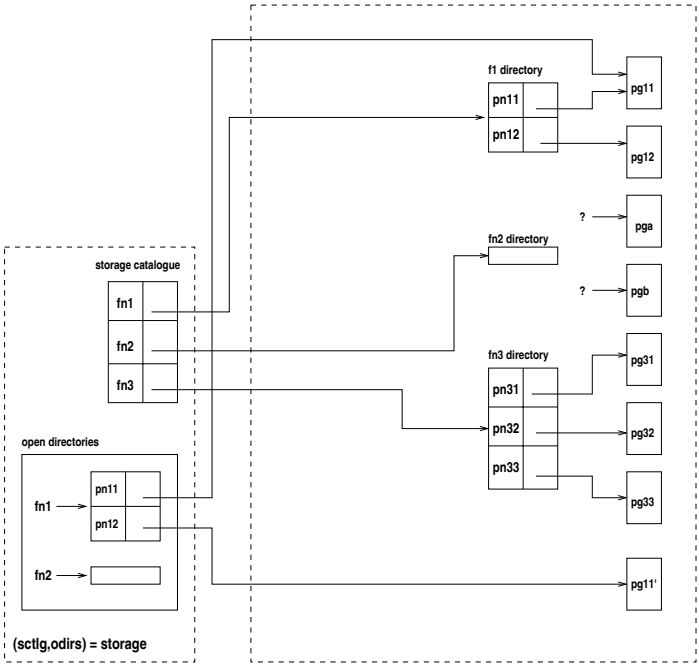


Fig. 27.5. Cached directory and page access

Formal Presentation: Semantic Data Types

```
type
    [Step 0]
  Fn, Pn, PAGE
  FS0 = Fn ~m~> (Pn ~m~> PAGE)

    [Step 1]
  Dn, Pa
  FS1 = CTLG1 × DIRS1 × PGS1
  CTLG1 = Fn ~m~> Dn
  DIRS1 = Dn ~m~> DIR1
  DIR1 = Pn ~m~> Pa
  PGS1 = Pa ~m~> PAGE

    [Step 2]
  Adr = Dn | Pa
  FS2 = CTLG2 × DISK2
  CTLG2 = Fn ~m~> Adr
  DISK2 = Adr ~m~> (DIR2 | PAGE)
  DIR2 = Pn ~m~> Adr

    [Step 3]
  FS3 = STG3 × DISK3
  STG3 = CTLG2 × (Fn ~m~> DIR2)
  DISK3 = DISK2
value
  open3: Fn → FS3 ~> FS3
  clos3: Fn → FS3 ~> FS3
```

So the data system consists now of a part residing in storage and another part residing on disk. The storage part has two parts: the only catalogue (there is), and the directories of opened files. The disk part consists of all directories and all pages, merged into one map, as in the previous step.

27.7.3 Invariance

Formal Presentation: Invariance

```
value
  inv_FS3: FS3 → Bool
  wf_StgDiskOverlap: FS3 → Bool
```

```
inv_FS3(fs3) ≡ wf_StgDiskOverlap(fs3) ∧ inv_FS2(abs_FS2(fs3))

wf_StgDiskOverlap((ctlg,odirs),disk) ≡
  dom odirs ⊆ dom ctlg ∧ ∀ fn:Fn • fn ∈ dom ctlg
    ⇒ odirs(fn)/dom disk(ctlg(fn)) = disk(ctlg(fn))/dom odirs(fn)
```

The well-formedness of the new file system has two parts: First, there must be "identity" (referred to as "overlap") between those (opened) directories residing in storage and those of the same files residing on the disk. Second, the file system abstracted from the now more concrete new file system must be invariant. We see that the opened (storage-bound, or residing) directories take precedence over the similar file directories residing on disk. That is, updates on the opened directories are not propagated "back" onto the disk before closing the respective directories.

This is reflected in the abstraction function, which retrieves "more abstract" file systems (step 2) from more concrete file systems (step 3); see next.

27.7.4 Abstraction

Formal Presentation: Abstraction

```
value
  abs_FS2: FS3 ~→ FS2
  abs_FS2(stg,dsk) ≡
    (ctlg,disk † [ctlg(fn) ↦ odirs(fn) | fn:Fn • fn ∈ dom odirs])
```

We leave the narration of the abstraction function to the reader.

27.7.5 Actions

We leave the annotation of the more concrete (step 3) action specifications to the reader.

Formal Presentation: Actions

```
Open and Close Actions

type
  FS3 = STG3 × DISK3
  STG3 = CTLG2 × (Fn -m→ DIR2)
  DISK3 = Adr -m→ (DIR2 | PAGE)
```

```
  CTLG2 = Fn ~m~> Adr
  DIR2 = Pn ~m~> Adr

value
  open3: Fn → FS3 ~→ FS3
  clos3: Fn → FS3 ~→ FS3

  open3(fn)((ctlg,odirs),disk) ≡
    ((ctlg,odirs ∪ [fn ↦ disk(ctlg(fn))]),disk)
  pre: fn ∈ dom ctlg ∧ fn ∉ dom odirs

  clos3(fn)((ctlg,odirs),disk) ≡
    ((ctlg,odirs \ {fn}),disk † [ctlg(fn) ↦ odirs(fn)])
  pre: fn ∈ dom ctlg ∧ fn ∈ dom odirs
```

Create and Put Actions

```
value
  crea3: Fn → FS3 ~→ FS3
  crea3(fn)((ctlg,odirs),disk) ≡
    let dn:Adr/Dn • a ∉ dom disk in
    ((ctlg ∪ [fn ↦ dn],odirs),disk ∪ [dn ↦ [ ]]) end
    pre: fn ∉ dom ctlg

  put3: Fn × Pn × PAGE → FS3 ~→ FS3
  put3(fn,pn,pg)((ctlg,odirs),disk) ≡
    if pn ∈ dom odirs(fn)
      then
        ((ctlg,odirs),disk † [(odirs(fn))(pn) ↦ pg])
      else
        let pa:Adr/Pa • pa ∉ dom disk in
        let odirs′ = odirs † [ fn ↦ odirs(fn) ∪ [pn ↦ pa]],
            disk′ = disk ∪ [pa ↦ pg] in
        ((ctlg,odirs′),disk′)
    end end end
    pre: f ∈ dom ctlg ∧ fn ∈ dom odirs
```

Erase, Get, and Delete Actions

These are left as exercises!

27.7.6 Adequacy, Sufficiency and Correctness

These are also left as exercises! Hint: Recall nondeterministic selection of Dn's and Pa's in FS2 and FS3. Therefore postulate the existence of one-to-one mapping(s) between (pairs of) Dn's, between (pairs of) Pa's and between Adr's and Dn's or Pn's.

27.8 Step 4: Storage Crashes

By storage crash we mean that information, i.e., data, kept by storage, as apart from being kept by a disk, is corrupted and can no longer be relied upon.

27.8.1 Storage and Disk

The catalogue is maintained in storage and if a crash occurs it cannot be used in order to gain access to the disk. Thus, to safeguard against loss of data a copy of the catalogue is kept on disk. Every so often the storage ("master") catalogue is copied — "checkpointed" — onto the disk. When a crash occurs, the disk is considered intact, and the disk copy of the catalogue can be copied "back" to storage. Certain actions performed between the most recent checkpoint and cashe restore must be repeated.

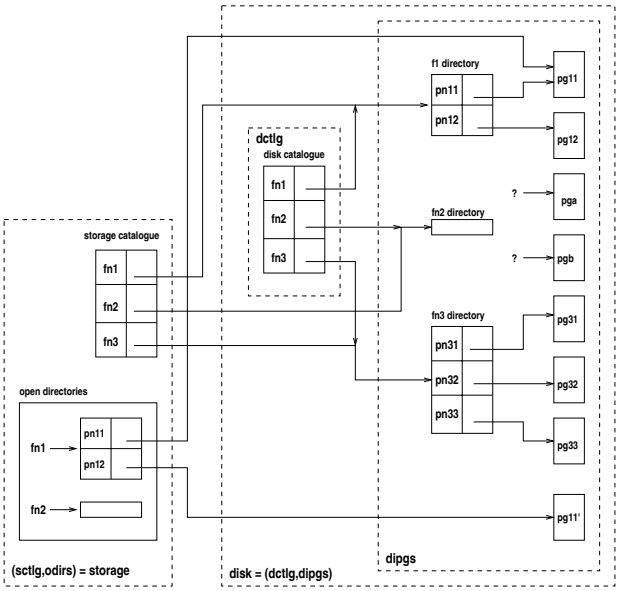


Fig. 27.6. Replicate catalogue

Figure 27.6 shows, relative to Fig. 27.5, the insertion of a copy on disk of the storage (hence the disk) catalogue.

27.8.2 Concrete Semantic Types

Formal Presentation: Concrete Semantic Types

```
type
    [Step 3]
  FS3 = STG3 × DISK2
  STG3 = CTLG2 × (Fn ~m~> DIR2)
  DISK2 = Adr ~m~> (DIR2 | PAGE)
  CTLG2 = Fn ~m~> Adr
  DIR2 = Pn ~m~> Adr

    [Step 4]
  FS4 = STG3 × DISK4
  DISK4 = CTLG2 × DISK2

value
  inv_FS4: FS4 → Bool
  inv_FS4(stg.disk) ≡ consSTG(stg,disk) ∧ consDISK(disk)

  consSTG: FS4 → Bool
  consDISK: DISK4 → Bool
```

27.8.3 Invariance

Well-formedness of the fourth step file system design is the conjunction of a consistent storage and consistent storage-disk pages.

27.8.4 Consistent Storage and Disks

Consistent Storage

A consistent storage has the names of all the opened (hence storage-bound) directories be a subset of the disk directories. Further, when retrieving the disk system — from the storage directories and the additionally defined disk directories — that current disk system, restricted (/) to those pages that can be reached from storage directories (as, and hence excluding void, old

pages), but extended (†) with the disk system (ds) reachable from the storage directories, shall be disk system 2 (i.e., DIR2) invariant.

Formal Presentation: Consistent Storage

```
value
  consSTG: FS4 → Bool
  consSTG((ctlg,odirs),(,dipgs)) ≡
    dom odis ⊆ dom ctlg
    ∧ inv_FS2(ctlg,currentSDiPgs((ctlg,odirs),dipgs))

  currSDiPgs: (STG3×(Fn ~m~> DIR2))×DISK2 → DISK2
  currSDiPgs((stg,odirs),dipgs) ≡
    let as = currSAddrs((ctlg,odirs),dipgs) in
    let ds = [ctlg(fn)↦odirs(fn)|fn:Fn•fn ∈ dom odirs]
    in (disk / as) † ds end end

  currSAddrs: (STG3×(Fn ~m~> DIR2))×DISK2 ~→ Adr-set
  currSAddrs((stg,odirs),dipgs) ≡
    let das = rng ctlg,
        opas = ⋃ { rng dir | dir:DIR2 • dir ∈ rng odirs},
        cpas = ⋃ { rng dipgs(a) | a:Adr • a ∈ { ctlg(fn)
                                | fn:Fn • fn ∈ dom ctlg \ dom odirs}} in
    das ∪ opas ∪ cpas end
```

Consistent Disk

The current disk system includes only those pages which can be "reached" (i.e., addressed) from the disk catalogues.

Formal Presentation: Consistent Disk

```
type
  DISK4 = CTLG2 × DISK2
        = (Fn ~m~> Adr/Dn) × ((Adr/Pa) ~m~> ((Dn ~m~> Adr) | PAGE))
value
  consDISK: DISK4 → Bool
  consDISK(dctlg,dipgas) ≡ inv_FS2(dctlg,currDDiPgs(dctlg,dipgas))

  currDDiPgs: DISK4 → Bool
  currDDiPgs(dctlg,dipgs) ≡ dipgs / currDAddrs(dctlg,dipgs)

  currDAddrs: DISK4 ~→ Adr-set
  currDAddrs(dctlg,dipgs) ≡
```

```
    let das = rng dctlg in
    let pas = ⋃ { rng dipgs(a) | a:Adr • a ∈ das } in
    das ∪ pas end end
```

27.8.5 Abstractions

One can abstract, i.e., retrieve to step 3 file systems either on the basis of storage catalogues, or on the basis of disk catalogues. The corresponding retrieve functions use the same restriction functions as defined for consistencies of storage, respectively disk subsystems above.

Formal Presentation: Retrieval Functions

From Storage:

```
value
  abs_FS3_STG: FS4 ~→ FS3
  abs_FS3_STG((sctlg,odirs),(,dipgs)) ≡
    ((sctlg,odirs),dsk/CurrSAddrs(sctlg,dipgs))
```

From Disk:

```
value
  abs_FS3_DSK:  FS4 ~→ FS3
  abs_FS3_STG(,(dctlg,dipgs)) ≡
    ((sctlg,[ ]),dipgs/CurrDAddrs(sctlg,dipgs))
```

27.8.6 Garbage Collection

In garbage collection we delete all those pages which can no longer be "reached" from the current storage and disk directories.

Formal Presentation: Garbage Collection

```
value
  GarbColl: FS4 ~→ FS4
  GarbColl((sctlg,odirs),(dctlg,dipgs)) ≡
    let sas = currSAddrs((sctlg,odirs),dipgs),
        das = currDAdrs(dctlg,dipgs) in
    ((sctlg,odirs),(dctlg,dipgas/sas ∪ das)) end
```

27.8.7 New Actions

Check and Crash Actions

To checkpoint a file means to update the disk with the latest version of that file's storage directory. For that a new, i.e., a "fresh" address is "fetched" and storage and disk catalogues suitably updated.

Formal Presentation: Check and Crash Actions

```
value
  check: Fn → FS4 ~→ FS4
  check(fn)((sctlg,odirs),(dctlg,dipgs)) ≡
    let a:Addr • a ∉ dom dipgs in
    ((sctlg † [fn ↦ a],odirs),
        (dctlg † [fn ↦ a],
            dipgs ∪ [a ↦ odirs(fn)])) end
  pre: fn ∈ dom sctlg ∧ fn ∈ dom odirs

  crash: () → FS4 ~→ FS4
  crash()(,(dctlg,dipgs)) ≡ ((dctlg,[]),(dctlg,dipgs))
```

To crash here means to render the storage catalogues void.

27.8.8 Some Previous Commands

Open and Close Actions

We leave it to the interested reader to "trace" the changes to the specifications of the open and close commands with respect to the file system of step 3.

Formal Presentation: Open and Close Actions

```
value
  open4: Fn ~→ FS4 ~→ FS4
  open4((sctlg,opdirs),(dctlg,dipgs)) ≡
    ((sctlg,odirs ∪ [ fn ↦ dipgs(sctlg(fn))]),(dctlg,dipgs))
  pre: fn ∈ dom sctlg ∧ fn ∉ dom odirs

  close4: Fn ~→ FS4 ~→ FS4
  close4((sctlg,opdirs),(dctlg,dipgs)) ≡
    let a:Adr • a ∉ dom dipgs in
    ((sctlg † [fn ↦ a],odirs \ {fn}),(dctlg,dipgs ∪ [a ↦ odirs(fn)])) end
  pre: fn ∈ dom odirs
```

Put Action

Formal Presentation: Put Action

```
value
  put4: Fn × Pn × PAGE ~→ FS4 ~→ FS4
  put4(fn,pn,pg)((sctlg,opdirs),(dctlg,dipgs)) ≡
    let a:Adr • a ∉ dom dipgs in
    ((sctlg,odirs † [fn ↦ odirs(fn) † [pn ↦ a]]),
          (dctlg,dipgs ∪ [a ↦ pg])) end
```

The put action resembles the check file action.

27.9 Step 5: Flattening Storage and Disks

27.9.1 "Flat" Storage and Disk

The former step of development modelled the disk as a pair consisting of a catalogue and the "previous" disk model. We now "merge" the former into the latter.

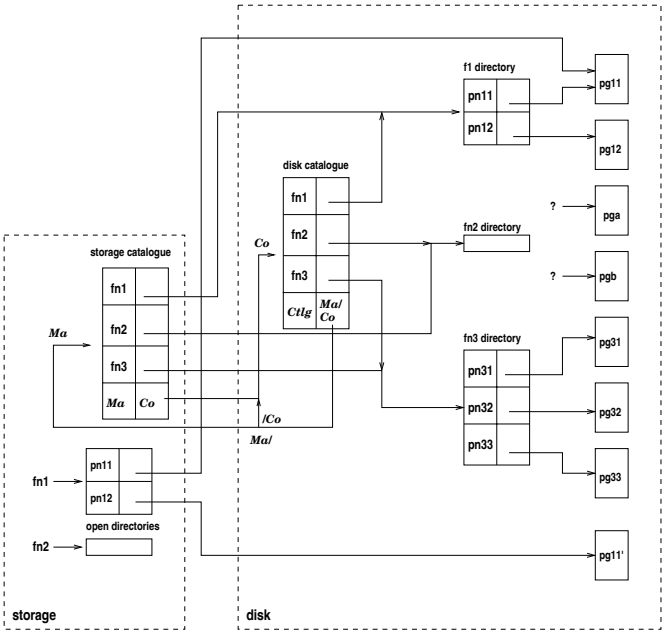


Fig. 27.7. "Flat" storage and disk

Figure 27.7 is really very much like Fig. 27.6. In that former figure some dashed (disk) boxes (disk catalogue and disk pages) and a fully drawn storage box (open directories) indicated separate accessible disk and storage areas. These are now "merged" into being generally addressable.

Formal Presentation: "Flat" Storage and Disk

From the former models we have:

type

[Step 3]
FS3 = STG3 × DISK2
STG3 = CTLG2 × (Fn $\overrightarrow{m}$ DIR2)
DISK2 = Adr $\overrightarrow{m}$ (DIR2 | PAGE)
CTLG2 = Fn $\overrightarrow{m}$ Adr
DIR2 = Pn $\overrightarrow{m}$ Adr

[Step 4]
FS4 = STG3 × DISK4
DISK4 = CTLG2 × DISK2

Defined in terms of some of the former types we get:

[Step 5]
Loc, Adr
FS5 :: STG5 × DISK5
STG5 = ({master} $\overrightarrow{m}$ SCTLG5) $\bigcup$ (Loc $\overrightarrow{m}$ DIR5)
DISK5 = ({copy} $\overrightarrow{m}$ DCTLG5) $\bigcup$ (Adr $\overrightarrow{m}$ (DIR5|PAGE))
SCTLG5 = ({master} $\overrightarrow{m}$ {copy}) $\bigcup$ (Fn $\overrightarrow{m}$ DAdr)
DCTLG5 = ({ctlg} $\overrightarrow{m}$ {master,copy}) $\bigcup$ (Fn $\overrightarrow{m}$ Adr)
DAdr = Adr × Ref
Ref == nil | Loc

27.9.2 "The Rest"

We leave the definition of invariants, abstraction function, actions, adequacy, sufficiency, and correctness — as an exercise — to the reader.

27.10 Step 6: Disk Space Management

27.10.1 The Issue

We refer to Fig. 27.8. We can usually consider both storage and disk to both consist of (thus two) finite sets of segments. At any one time some of these

pages are being used — for storage and disk catalogues, directories and file pages. And, at any one time the remaining, thus unused, segments are "free". We decide, therefore, to maintain "lists" (actually sets) of references to free segments.

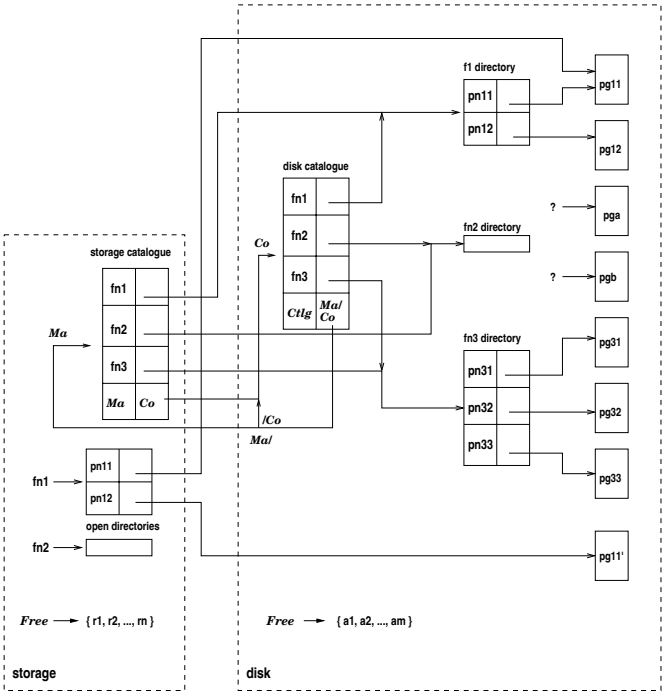


Fig. 27.8. Space management

Free segments can be allocated for new file (disk) pages, or new file (storage or disk) directories. Such allocation removes a reference to a free segment from the appropriate "free list". Whenever directories and file pages have been deleted *and* storage and disk catalogues and directories made to correspond, then garbage collection can be applied. Garbage collection now "returns" the reference to the garbage-collected segment to the "free list".

27.10.2 "The Rest"

We leave the definition of types, invariants, abstraction function, actions, adequacy, sufficiency and correctness — as an exercise — to the reader.

27.11 Discussion

27.11.1 General

A long odyssey has ended. We have "slowly", in small, "measured" steps unfolded a software component design. From the basis of a "small" system component, we added, "one-by-one", additional properties. Some properties were determined by domain requirements considerations. Other properties were determined by machine requirements considerations. Some "new, added" properties could be invoked by new commands. Other "new, added" properties are just "always" there. To properly read and grasp this chapter the reader must carefully read and make sure to understand every detailed step, its design decisions, i.e., each and every line of specification, and its micro steps. Proper understanding requires patience, and that the reader solves the posed exercises.

We have also, intertwined with the incremental presentation of design details, presented method principles and techniques. The stepwise unfolding is one such principle. In each step, especially in the transition from one step to the next, there were subprinciples and techniques. These include being concerned about, and hence defining invariants (i.e., well-formedness); relating "a more concrete step" to its "more abstract predecessor step", and hence defining abstraction functions; and being concerned about correctness of a stepwise refinement or extension, and hence in establishing correctness criteria. As a matter of principle, a metaprinciple, this book does not show actual proofs of correctness.

The major lesson of this chapter can be summarised: When designing systems with a great many concepts, do so in stages of refinement and extension. That is, introduce a few concepts at a time. Sketch invariants, abstraction functions, and correctness criteria. Basically try to redefine the semantic functions in every step. It is a good way to see whether one has confidence in the present step's design decisions.

27.11.2 Principles and Techniques

We summarise:

Principles. And yet another principle of *component development* is that of possibly helping to *discover* new, initially, i.e., during requirements development, unforeseen requirements. ∎

Principles. One possibility of *component development* is that of the *stepwise unfolding* of externally observable properties. That is, of the extension of an architecture that handles some, but not all requirements, to increasingly cater for additional requirements. ∎

Principles. Another principle of component development is that of *stepwise refinement* or *stepwise extension:* Either making more concrete a data type while redefining operations over any such data type (refinement), or "adding" additional operations (extension). And doing this in up to several steps. ▪

Techniques. The corresponding techniques of *component development* include development of invariants (well-formedness), abstraction (and injection) functions, adequacy and sufficiency relations, more concretised operation (action) definitions, and statements and possible proofs of correctness. ▪

27.12 Bibliographical Notes

The file system outlined in this chapter is based on Stoy and Strachey's utterly elegant operating system OS6 [347]. Work on the current model was prompted by Abrial's approach as documented in item (4) of [3].

27.13 Exercises

27.13.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

27.13.2 The Exercises

Exercises 27.1–27.4 assume that you have selected a topic and are carrying through a series of solutions around this selected topic, i.e., solutions to exercises in previous chapters, and now also this chapter. Exercises 27.5–27.13 are specific to this chapter. You may wish to try your mind and hand on the latter exercises first.

Exercise 27.1 *Informal: Specific Topic System, Initial Components Γ_{i_0}.* For the fixed topic, selected by you, and based on your solution to Exercise 26.1, suggest — for each of three process components $\Gamma_{1_0}, \Gamma_{2_0}, \Gamma_{3_0}$ — a main state, say Σ_{i_0} (like the file systems of this chapter: FS0 was a state), and some actions, i.e., α_{i_j} for some $j = 1, 2, \ldots, n_i$ (i.e., operations and functions) on these. Sketch some state structure (Fig. 27.1 was such a sketch), and narrate some actions (etc.). (Together Σ_{i_0} and $\{\alpha_{1_1}, \alpha_{1_2}, \ldots, \alpha_{1_{n_1}}\}$ form Γ_{i_0}.)

Exercise 27.2 *Formal: Specific Topic System, Initial Components Γ_{i_0}.* This exercise continues Exercise 27.1. Based on your informal specification, now formalise each Γ_{i_0}: The state Σ_{i_0} and the actions (operations and functions) over Σ_{i_0}.

Exercise 27.3 *Informal: Specific Topic System, Concrete Components Γ_{i_k}.* This exercise continues Exercise 27.1. Suggest, for each of your components Γ_i, a sequence of one or two steps of development of your chosen states and operations: Draw sketch diagrams (like Figs. 27.2 and 27.4–27.8), and narrate a selection of invariant, abstraction, adequacy, sufficiency, action and correctness specifications.

Exercise 27.4 *Formal: Specific Topic System, Concrete Components Γ_{i_k}.* This exercise continues Exercise 27.3. Based on your informal specification, now formalise each Γ_{i_k}: The state Σ_{i_k} and the actions (operations and functions) over Σ_{i_k}.

Exercise 27.5 *Narration of Abstract Function: abs_FS2.* Reference is made to Sect. 27.6.5. Provide a concise narration of an abstraction function from values of type FS2 to values of type FS1.

Exercise 27.6 *Informal: Step 2 Adequacy, Sufficiency, Operations, and Correctness.* Reference is made to Sect. 27.6.6. Narrate the adequacy and sufficiency relations, the create2, erase2, put2, get2, and delete2, actions, and the correctness statement for step 2 with respect to step 1.

Exercise 27.7 *Formal: Step 2 Adequacy, Sufficiency, Operations and Correctness.* Reference is made to Sect. 27.6.6 and to Exercise 27.6. Formalise answers to the questions posed in Exercise 27.6.

Exercise 27.8 *Informal Step 3 Erase, Get and Delete Actions.* Reference is made to Sect. 27.7.5. Narrate the erase2, get2 and delete2 actions, as defined over FS3.

Exercise 27.9 *Formal: Step 3 Erase, Get and Delete Actions.* Reference is made to Sect. 27.7.5 and to Exercise 27.8 (above). Formalise the erase2, get2, and delete2 actions as defined over FS3, and as narrated in Exercise 27.8.

Exercise 27.10 *Informal: Step 3 Adequacy, Sufficiency and Correctness.* Reference is made to Sect. 27.7.6. Narrate adequacy, sufficiency and correctness criteria for step 3.

Exercise 27.11 *Formal: Step 3 Adequacy, Sufficiency and Correctness.* Reference is made to Sect. 27.7.6 and Exercise 27.10. Formalise the adequacy, sufficiency and correctness criteria for step 3.

Exercise 27.12 *Formal Step 5: "The Rest".* We refer to Sect. 27.9.2. Narrate and formalise the invariants, abstraction function, action function definitions, and the adequacy, sufficiency and correctness criteria for step 5.

Exercise 27.13 *Formal Step 6: "The Rest".* We refer to Sect. 27.10.2. Narrate and formalise the invariants, abstraction function, action function definitions, and the adequacy, sufficiency, and correctness criteria for step 6.

28

Domain-Specific Architectures

- The **prerequisite** for studying this chapter is that you have a firm grip on most of the previous material as well as some nontrivial familiarity with such concepts as compiler systems, operating and real-time systems, database systems, administrative data processing systems, data communication systems, etc.
- The **aims** are to argue, with Michael Jackson, that no one "method", i.e., no one comprehensive set of principles and techniques suffices for all software development, to introduce the concept of a set of problem frames, each frame with its diversity of domain, requirements and design techniques, and to illustrate, in particular, such frames as translation, information repository, client/server, workpiece, reactive systems, and connection frames.
- The **objective** is to enable you — the software engineer — to better select most "fitting" problem frames, and hence models and development principles, techniques and tools.
- The **treatment** is from systematic to formal.

28.1 Introduction

The basic idea is: can we advise on a special set of principles, techniques and tools for the development of a class of clearly delineated software packages cum systems? The basic answer is: yes, for some such identified classes we can. Hence that is what this chapter is "all" about!

28.1.1 General

Reference is made to Michael Jackson's work on Problem Frames [189, 191].

28.1.2 Some Definitions

Characterisation. By a *software architecture* we shall understand a certain composition of two or more (software) components with one or more (software) connectors. ∎

Characterisation. By a *component* we shall understand a syntactic thing whose semantics can be thought of as an algebra, that is, a set of entities and a set of functions over these. We enlarge on this understanding and think of a component as a process, one with possibly many alternative behaviours. ∎

Characterisation. By *component functionality* we shall understand the set of separately invokable functions of a component as specified by the process definition of the component — where the invocations are achieved by means of input communications from the environment of the component. ∎

Characterisation. By a *connector protocol* we shall understand the set of traces of events, i.e., input/output behaviours as specified by the process definitions of one or more connected components. ∎

Characterisation. By an *uninterpreted software architecture* we shall understand almost the same as a software architecture except that we make no assumptions about component functionalities and connector protocols. ∎

Characterisation. By an *instantiated software architecture* we shall understand a software architecture whose component functionalities and whose connector protocols have been specified. ∎

Characterisation. By a *generic software architecture* we shall understand an uninterpreted software architecture whose respective instantiations implement requirements otherwise considered different. ∎

28.1.3 On Architectures

Chapter 26, "Software Architecture Design", illustrated one kind of software architecture. Let us assume that the components and the connections (i.e., the processes and the channels) of that architecture, for example, Fig. 26.6, are otherwise uninterpreted. That is, we do not specify the function (i.e., the process) of any components (i.e., box), or the protocol of messages on any channel (i.e., arrow).

The questions now are: Can one identify a number of different requirements which can be somehow implemented by the same generic software architecture, that is, by reusing the same composition of components and connections, but by defining different component functions and different connection protocols? Can one identify a number of generic architectures, i.e., compositions of

components and connections that are sufficiently distinct (see next), for each of which the instantiation of component functions and connection protocols leads to software architectures that meaningfully implement (some) respective requirements? This chapter shall answer the latter question in the affirmative.

The "somehow implemented" term used above begs an explanation. By somehow implemented we mean an instantiation of an, or the, uninterpreted software architecture, where the resulting software architecture implements the requirements. The "sufficiently distinct" term used above also begs an explanation. For two (possibly uninterpreted) software architectures to be sufficiently distinct we require something like a different (connection) topology of components, possibly a different number of components, and possibly a different wiring pattern of connectors.

28.1.4 Problem Frames

The affirmative answer alluded to above will be given by presenting a number of what we shall call problem frames.

Characterisation. By a *problem frame* we shall understand a set of problems whose partial or full solution by means of software can be expressed within one and the same generic software architecture. ∎

We identify the following problem frames for treatment in the next sections:

- **Translator frame:** The set of problems that amount to the descriptions of a sizeable variety of programming languages, as well as the implementation of individually prescribed requirements to interpreters and compilers for these programming languages.
- **Information repository frame:** The set of problems that amount to the descriptions of actual-life repository systems, i.e., systems that keep and maintain large amounts of information, as well as the definition and implementation of prescribed requirements to computerised such information systems, i.e., database systems.
- **Client/server frame:** The set of problems that amount to the descriptions of systems of "masters and slaves", i.e., of 'clients and servers', as well as the implementation of prescribed requirements to what we shall thus call client/server systems. That is domains in, respectively software for, which a number of agents (i.e., clients) can have some functions (i.e., some services) handled by shared agents (i.e., a servers).
- **Workpiece frame:** The set of problems that amount to the descriptions of typically administrative or design-based activities, as well as the implementation of prescribed requirements to such document- or workpiece-based activities: from public administration (e.g., internal revenue service [tax and excise]), via private service industries (e.g., insurance companies), to, for example, CAD/CAM design development.

- **Workflow frame:** The set of problems that amount to the descriptions of, and implementation of prescribed requirements to systems, which can best be characterised by flows and interaction of people, (other) resources, materials, information and control — such as, for example, in production (i.e., manufacturing), logistics and transport, and healthcare (i.e., hospitals).
  We illustrated the concept of workflow frame in Vol. 2, Chap. 12, Sect. 12.5.
- **Reactive systems frame:** The set of problems that amount to the descriptions of, and the implementations of prescribed requirements to the monitoring and control of mechanical, electro-mechanical, and chemical processes — such as in automobiles, aircraft, power plants, cement factories and oil refineries.
- **Connection frame:** The set of problems that amount to the descriptions of, and the implementations of prescribed requirements to connected systems — with emphasis on the connections, i.e., the possibly noisy data communication or message channel protocols — such as sensors and actuators connected, on one side, to, for example, people or mechanical processes and, on the other side, to computers, possibly via telecommunication (whether wireless or not).

Please observe the general pattern of the above sentences: "The set of problems that amount to description of ..., and implementation of prescribed requirements to ...". Those sentences reveal that a proper delineation of a problem frame consists of three parts: a generic characterisation of the domain, generic characterisation of the requirements and, finally, a generic characterisation of the domain-specific software architecture.

28.1.5 Chapter Structure

The structure of the rest of the chapter thus follows as a sequence of sections, one for each of the problem frames listed above, followed by a discussion of composite problem frames. Each section has subsections corresponding to the domain, the requirements and the software architecture design.

28.2 Translator Architectures

The aim of this section is to motivate otherwise well-known architectures for the software of interpreters and compilers.

Characterisation. By a *translator frame* we understand the set of problems that amounts to description of a sizeable variety of programming languages, and the implementation of prescribed requirements to interpreters and compilers for such programming languages. ∎

28.2.1 Translator Domain

Basis for a translation domain: We have the basis for a translation problem frame if the domain can be expressed in terms of two formal languages: their formal syntax (L_1, L_2) and semantics (D_1, D_2, M_1, M_2). ∎

type
  L_1, L_2, D_1, D_2
value
  M_1: $\mathrm{L}_1 \xrightarrow{\sim} \mathrm{D}_1$,
  M_2: $\mathrm{L}_2 \xrightarrow{\sim} \mathrm{D}_2$

Usually, for formal languages, the meaning functions M_1, M_2 can be expressed as the composition, each of two functions: a static semantics and a dynamic semantics function (Mss_i, Mds_i):

value
  Mss_i: $\mathrm{L}_i \to$ **Bool**
  Mds_i: $\mathrm{L}_i \to \mathrm{D}_i$
  $\mathrm{M}_i(\mathrm{p}_i) \equiv$ **if** $\mathrm{Mss}_i(\mathrm{p}_i)$ **then** $\mathrm{Mds}_i(\mathrm{p}_i)$ **else** $\bot$ **end**

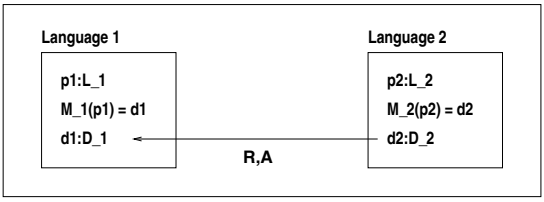


Fig. 28.1. Translation domain

Translation domain: We have a translation frame if furthermore we have a set of abstraction or retrieve functions between semantics domains of the two languages (Fig. 28.1). ∎

To express the retrieve relation we postulate a notion of 'state', subsets S_1 of D_1, respectively S_2 of D_2.[1] Then:

- Retrieve: R: $S_2 \to S_1$;
- Abstraction: A: $D_2 \times D_1 \to$ **Bool**.

[1] At present it is not important to know other than that about states!

28.2.2 Translator Requirements

The translation requirements define what is, at a minimum, required from a translator T, namely that the translation preserves the semantics, that is: That the meaning M_1 of abstract programs p_1 in L_1 is preserved, somehow (i.e., A), by the meaning M_2 of concrete, translated, T, programs p_2 in L_2. Instead of the meaning, we can assume comparable, R, abstract and concrete states, and then "run" the meanings on those states.

Translation requirements: Translator T requirements are expressed in terms of the commutation of a diagram.

- Commutation: $R(s_1, s_2) \supset A(M_1(p_1)s_1, M_2(T(p_1)s_2))$.

If states are comparable then so are the interpretations (M) of the abstract and the translated (T) programs. ∎

Interactive diagnostics and editing requirements belong to a workpiece-like frame.

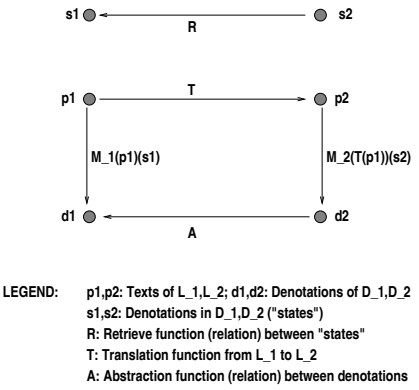


Fig. 28.2. Translation requirements

28.2.3 Translator Design

Translator (compiler) design: Following one compiler design paradigm we may structure the program organisation as follows:

- *Lexical scanner:* In a lexical scanner input language (L_1) texts have their individual characters assembled, i.e., "translated" into tokens.
- *Syntax parser:* In a syntax parser linear sequences of tokens are parsed, that is: "translated" into a parse, i.e., a tree-structured text in a language PS_1.

- *Static semantics analyser:* The static semantics analyser implements Mss_1.
- *"Optimiser":* An "optimiser" may reorganise the abstract PS_1 text in a way that may make the subsequent code generator able to generate so-called "efficient" texts.
- *Code (text) generator:* The code generator "implements" parts of Mds_1 such that the diagram commutes. The text produced is usually a linear sequence of abstract tokens or concrete characters, including blanks, line shift, etc.

Translator Functions

type TK_1, PS_1
value
  LS: $L_1 \xrightarrow{\sim} TK_1$ /* Lexical scanner */
  PS: $TK_1 \xrightarrow{\sim} PS_1$ /* Parser */
  SA: $PS_1 \to$ **Bool** /* Static Analysis */
  OPT: $PS_1 \to PS_1$ /* Optimiser */
  CG: $PS_1 \to L_1$ /* Code Generator */
axiom
  $\forall\ p_1{:}L_1, \exists\ s_1{:}S_1, s_2{:}S_2 \bullet$
    $R(s_1,s_2) \Rightarrow$ **let** $ps = PS(LS(p_1))$ **in**
      $SA(ps) \Rightarrow A(M_2(CG(ps)(s_2)),M_1(p_1)(s_1))$ **end**

■

A Multipass Compiler

The sum total is a multipass compiler (translator), usually shown as in Fig. 28.3.

We shall, in line with Chapter 26's rendition of architectures in the form of boxes (as processes) and arrows (as channels), instead show Figure 28.3 in that former form (Fig. 28.4).

The double arrows on the left of the lexical scanner box shall mean: The output arrow (to the left) shall designate that the lexical scanner wishes to input a next character. The input arrow (to the left) shall designate that the lexical scanner receives such character input. The double arrows between boxes shall mean: The output arrow (to the left) shall designate that the box process is ready to receive more input. The input arrow (to the left) shall designate that the box process receives such input. The output arrow (to the right) shall designate that the box process is ready to deliver output. And the input arrow (to the right) shall designate that the box process is requested to deliver such output. The single arrow on the right of the code generator box shall mean: The code generator box is ready to deliver, and delivers output.

By a multipass program we shall mean a program that reads what might be considered "the same input" over and over again! By a multipass compiler

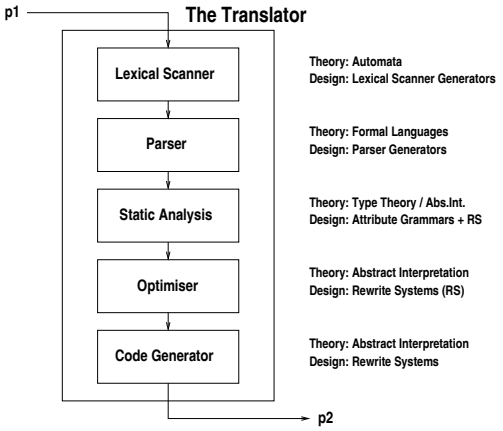


Fig. 28.3. Translation design

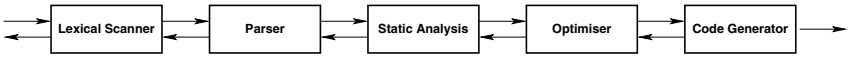


Fig. 28.4. Simple compiler architecture

we shall mean a compiler that first reads an external input, namely the program character list, and "builds" an internal representation, in the form of a sequence of tokens. It then reads that token sequence, parses it, and "builds" an internal representation, in the form of a parse tree. The multipass program then reads that parse tree, annotates it with results of the static analysis, and thus "builds" a new internal representation, in the form of an annotated parse tree. It then reads that annotated parse tree, and rearranges subtrees (in order to reflect "optimisation"), and thus "builds" a new internal representation, still in the form of an annotated parse tree; and then finally reads that annotated, possibly "optimised" parse tree, and, from it, decides what code to emit.

The multipass "nature" of the compiler can be reinterpreted, for example, into being a single-pass compiler: Instead of understanding the component processes as parallel processes, one may view them as coroutines.

28.2.4 Process Graph for Translator Development

In this section we "step aside" from the main purpose of this chapter to briefly consider how one might structure the overall development of compilers for large, complicated and, in this case, parallel programming languages. We refer to Fig. 28.5:

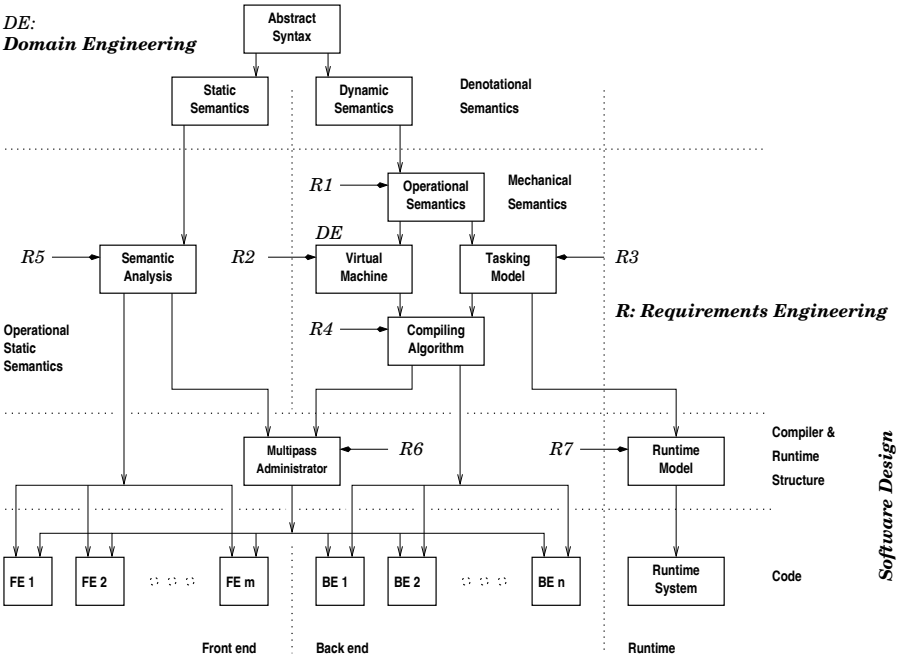


Fig. 28.5. A compiler development graph

We consider, in turn, the three phases of development: first, domain and requirements engineering, then some material in preparation for software design, and, finally, software design.

Domain Engineering

The domain engineering in the simplified case of only considering the intrinsics aspects of the programming language itself, not the way in which it is to be used by programmers, amounts to the description of the following:

- **Syntax:** Precise descriptions of both the `BNF` and the abstract, in our case, `RSL`, syntaxes, and appropriate relations between them.
- **Static semantics:** Precise descriptions of the syntactic well-formedness of all language texts.
- **Dynamic semantics:** Precise descriptions of the semantic meaning of all language texts. For a parallel programming language the dynamic semantics imply a description of an abstract tasking model (i.e., a model of how concurrent tasks may synchronise and exchange information).

Software Design

In order to formulate requirements (in the case that we are exemplifying, these are named R_1, R_2, R_3, R_4, R_5, R_6 and R_7, and they are explained further

below), i.e., in order to make the formulations (R_i) precise, one must first prescribe what the compiler and run-time systems must do:

- **"Front end":** The "front end" checks all such syntactic constraints that are statically decidable, that is, that uses of names are preceded by their definition, that operand expressions are of the right type, etc.
- **"Back end":** The "back end" generates code for a target computer.
- **Run-time system:** The run-time system, for example, either (i) handles possible serialisation of parallel program tasks on a serial computer, or (ii) synchronisation and communication between parallel program tasks on parallel computers, or (iii) combinations thereof.

To enable precise formulations of the requirements ($R_1, R_2, R_3, R_4, R_5, R_6$ and R_7), we find it useful to refine the *static* and *dynamic semantics* in respective stages of development:

- **Front end:** The development of requirements for the compiler "front end" usually consists of one or more steps of
  - ⋆ **Static analysis refinement:** From the usually abstractly described *static semantics* is developed increasingly more concrete prescriptions for *static analysis*. More on this below.
- **Back end:** The development of requirements for the compiler "back end" usually consists of one or more steps of refinements of the *dynamic semantics*. We have shown, in Volume 2 of this series of textbooks on software engineering, in Chapters 16–18, how to develop back-end prescriptions.
  - ⋆ **Operational semantics:** See Vol. 2, Chaps. 16–18's respective sections on macro-expansion semantics.
  - ⋆ **Virtual machine (design):** See Vol. 2, Chap. 16's Sect. 16.7 on the design of a machine language.
  - ⋆ **Tasking model:** See Vol. 2, Chap. 19.
  - ⋆ **Compiling algorithm:** See Vol. 2, Chap. 16, Sects. 16.8–16.10 on compiling algorithms. This prescription states what code the compiler must generate for each and every phrase segment of the language.
- **Run-time model:** From the dynamic semantics' description of an abstract *tasking model* is developed a prescription for a concrete run-time system and what it shall handle: which calls it may receive from "running", compiled code (i.e., tasks), and how it must respond to such invocations.

Requirements

In order to further explain the software design we first need to mention (some of) the requirements:

- R_1 : Compiled programs shall be subject to execution on a monoprocessor (hardware platform).
- R_2 : One and the same compiled program shall be subject to execution on different monoprocessor architectures (hardware platforms).

- R_3 : Compiled programs shall not be dependent on any computer operating system other than a run-time system specially set up for this programming language (software platforms).
- R_4 : Compiled programs shall "fit" within a 128 kilobyte storage addressing space.
- R_5 : The compiler front end shall, itself, fit within a 128-kilobyte storage addressing space.
- R_6 : The compiler back end shall, likewise, fit within a 128-kilobyte storage addressing space.
- R_7 : The compiler shall compile indefinitely large programs.

Software Design — Revisited

In preparation for some design decisions let us review some compiling possibilities that are anchored in how a compiler might prescribe the traversals of program texts.

Parse Tree Traversals: We refer to Fig. 28.6. It shows a parse tree, and three left-to-right traversals: pre-, in-, and post-order traversals.

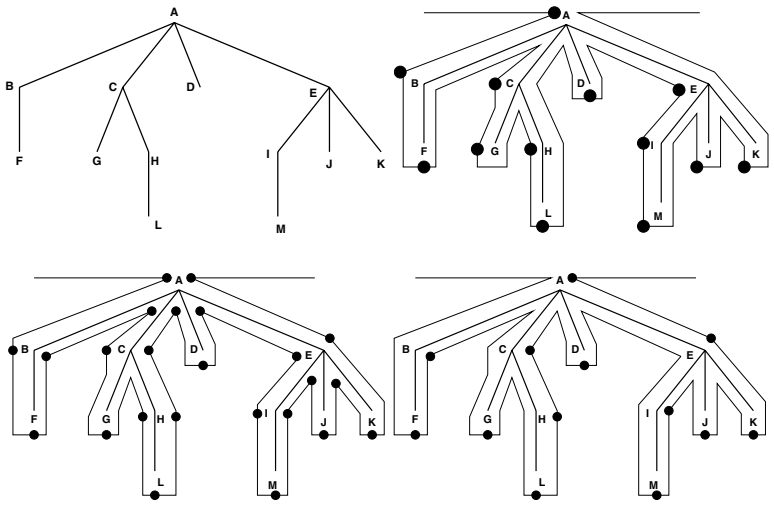


Fig. 28.6. Left-to-right parse tree traversals

- **Linear traversals:** To be able to satisfy compiler requirements R_5, R_6 and R_7, we decide to keep, in "foreground" storage, only a fragment of a program, namely that part of the parse tree which "lies" on the path between the root of the phrase tree and the node currently being handled by any part of the front or the back ends.

- **Left-to-right and right-to-left traversals:** It then turns out that one may wish to make use of the possibility of traversing the parse tree in forward (left-to-right) and backward (right-to-left) directions.[2] A reason motivating right-to-left traversals may be the collection of forward jump program points (labels) before their "use". So one should read the concrete prescriptions of the static analysis and the compiling algorithm specifications with a view towards deciding which traversal directions to deploy, pass by pass.
- **Pre-, in-, and post-order traversals:** Processing of nodes of the parse tree can proceed by handling these in either only a pre-order, or only an in-order, or only a post-order fashion, again, depending on which pass one is considering.

Multipass Compiler: There is thus a total of six possible orders of traversal. Each traversal corresponds to a pass.

- **Prescription analysis:** The front and the back end prescriptions, i.e., the (last step of development of the) *static analysis* and the *compiling algorithm*, need therefore be inspected:
  - ⋆ These documents need to be read line by line. And for each line, or even fragment of a line, it need be decided whether what that line (fragment) prescribes can be done in a first pass, in a second pass, etc.
  - ⋆ That is, in which earliest pass the compiler may honour that prescription.
  - ⋆ And for each pass it must further be decided which is a most suitable form of traversal: left-to-right or right-to-left, and pre-order, in-order, or post-order.
  - ⋆ The decisions outlined in the above three items interact with one another.
- The result of this analysis (of both the *static analysis* and the *compiling algorithm*) is the smallest number of logical, linear, left-to-right or right-to-left, and pre-order, in-order, or post-order passes.
- For each pass, say pass i, it is then examined, i.e., estimated, whether the code of that pass component will fit in a 128-kilobyte addressing storage space. If not, then pass i may have to be decomposed into two or more passes: $i_1, \ldots, i_{i_n}$.
- The result of this analysis (of each pass i prescription, as it is so marked in either the *static analysis* and the *compiling algorithm* prescriptions), is the smallest number of physical, linear, left-to-right or right-to-left, and pre-order, in-order, or post-order passes.

Based on the above we can now explain the software design of the compiler.

[2] On special hardware, especially storage, notably disk, architectures one may be able, with some access time advantage, to read in from backing store what was written out, in reverse order.

- **Main architectural decision:** The compiler software architecture now consists of a number of front end passes, and a number of back end passes — all "held together" by a multipass administrator.
- **Multipass administrator:** The basic need for the multipass administrator is motivated in the requirement that the compiler fit in a 128-kilobyte addressing storage space: The multipass administrator component sequences the deployment of each of the front and back end pass components, and administrates the "fetching" and "storing" of original input text, of all intermediate texts, and of the final output text.
- **Front end:** Each of the front end passes (i.e., of front end components) is now prescribed by appropriately annotated lines and line fragments of the *static analysis*.
- **Pass F_1:** Ideally each front end pass component can be coded (i.e., programmed) solely based on the *static analysis* and the specification of the *multipass administrator*.

  ⋮
- **Pass F_f:** These programming items are potentially carried out independently of the programming of any of the other pass components.
- **Back end:** Each of the back end passes (i.e., of back end components) is now prescribed by appropriately annotated lines and line fragments of the *compiling algorithm*.
- **Pass B_1:** Ideally each back end pass component can be coded (i.e., programmed) solely based on the *compiling algorithm* and the specification of the *multipass administrator*.

  ⋮
- **Pass B_b:** These programming items are potentially carried out independently of the programming of any of the other pass components.
- **Run-time system:** And, independently of the programming of the compiler, one can program the run-time system solely based on the *run-time model* prescription.

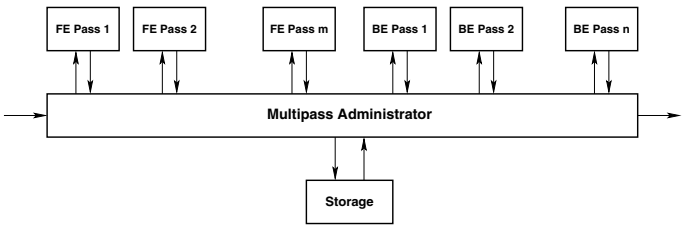


Fig. 28.7. A multipass compiler architecture

This ends our long sketch of a proper procedure for developing compilers for a large class of programming languages such as `Chill` [134,135], `Ada` [40,60], `Java` and `C#`.

28.3 Information Repository Architectures

We wish to capture an essence of such systems whose main implementation feature is that of a shared database. The database, it is argued, represents formalised representation of information. And that information resides in the actual world, i.e., in the, or in a domain.

Example 28.1 *Information Bases:* Examples of information could be:
(1) The information of a company's personnel: their hiring status (position, salary, etc.), their organisational placement (department, boss, etc.), their working hours and time sheets, their family and health status, etc.
(2) The information of a company's marketing and sales: sales catalogues, sales on order, but not delivered, past sales delivered, returns of faulty or otherwise unsatisfactory merchandise, refunds, etc.
(3) The information about the weather: the past week's, yesterday's and today's actual weather, satellite images of the weather "around us", etc.
(4) A hospital's status with respect to immediately past, current and scheduled future patients, the medical records of these patients, and staff information: names of people, qualifications, working hours, and allocation and schedules, etc.
(5) The information concerning a geographical area's natural resources and their state, land plot by plot: its coordinates, surface and other minerals, water content, current use, etc. ∎

In "converting" the information a number of abstractions take place, and a number of representation decisions are made. Usually it is a good idea, early on in this conversion process, to consider the operations that one is applying to information in the domain, and hence might wish to require for the software implementation.

Example 28.2 *Information Abstraction, Representation and Operations:* Examples of abstractions, representations and operations could be: (1) For the company personnel domain we abstract away from the finer details of health; and otherwise we represent the information in the form of records (i.e., structures, or Cartesians) of scalar data and character strings; operations are basically those of extracting fields of the record (components of the Cartesians), changing certain, for example character string field values, with others, and adding to working days and earned salary fields. ∎

Abstraction, representation and operation issues are to be captured in the very next sections.

28.3.1 Information Repository Domain

We start by showing a picture of the informal domain, i.e., the cloud (Fig. 28.8).

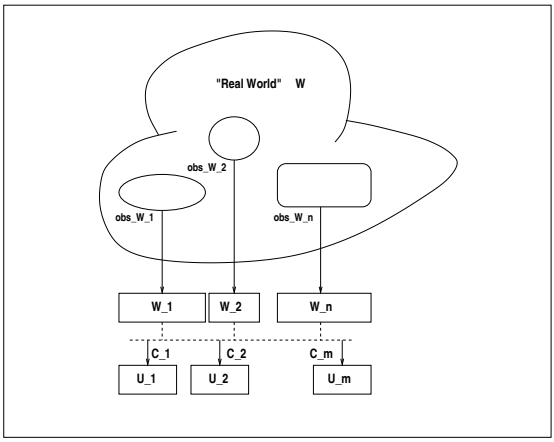


Fig. 28.8. Information repository system domain

To that we add our informal view of some of this information, the oval, the circle and the rounded box, all "inside" the cloud. Next we add the extraction, in the form of data — the first row of rectangles immediately below the cloud — of some of this information. Finally we perform calculations on these data and arrive at model (prediction) data, the second row of rectangles below the cloud.

A basis for an information repository domain: We have the basis for a(n information) repository system frame if there is given a model of the "real world", W, in terms of the $i = 1, \ldots, m$ types, W_i, of some of its atomic or composite individuals, the signature of its observer functions (relations) Q_i $(j = 1, \ldots, m)$, and predicates P_j $(k = 1, \ldots, n)$ constraining some of its (atomic or composite) individuals. ∎

Information Types and Function Spaces: Repository Frame

type for $(i, j) = 1, 2, ..., (m, n)$:
W, W_i, Index
$\Sigma = \mathrm{T} \xrightarrow{\sim} \mathrm{W}$
$\mathrm{Q}_i = \Sigma \to \mathrm{T} \to \mathrm{i{:}Index} \xrightarrow{\sim} \mathrm{W}_i$
$\mathrm{P}_j = \mathrm{W}_{k_1} \times \mathrm{W}_{k_2} \ldots \mathrm{W}_{k_\ell} \to \mathbf{Bool}$

The Q_i (observer) functions — given observation time and an index, say a name of the information to be observed — act as "projections" from the "real

world" into W_i. Typical observer functions, in a noncomputerised "world", amount to ordinary human *data collection:* the gathering of population census in paper files, or the gathering of experimental data from a series of scientific experiments, likewise in a paper file. The predicates P_j correspond to the *data vetting* normally done in the above informal example situations, i.e., the human checking and cross-checking that collected data satisfy various constraints.

Information repository domain: We have an information repository system frame if — in addition to the above basis — we further have a number of data types U_d and functions C_f which perform a number of calculations over the projected (and resulting) data W_{i_i}, U_{j_j}. ∎

Calculations: Repository Frame

type
  U_1, U_2, ..., U_d
value
  for $f = 1, 2, ..., g$:
    C_f: $\mathrm{W}_{i_1} \times \mathrm{W}_{i_2} \times ... \times \mathrm{W}_{i_k} \times \mathrm{U}_{j_1} \times \mathrm{U}_{j_2} \times ... \times \mathrm{U}_{j_\ell} \xrightarrow{\sim} \mathrm{U}\mu$

The full force of domain engineering, of course, shall apply: One has to properly model, i.e., describe, either only informally, or both informally and formally, the domain of all of a specifically chosen universe of discourse. This includes the intrinsics, the support technologies, the management and organisation, and the rules and regulations, etc. The above formula really only hints at such a description, and can thus be considered metalinguistic.

28.3.2 Information Repository Requirements

Information repository system requirements: Information repository system requirements are expressed basically as follows

- **Data collection and vetting:** Requirements prescribe observer (data collection) functions Q_i, and data vetting as expressed by the predicates P_j.
- **Data storage:** Requirements prescribe (including migration [i.e. conversion] of paper) data files.
- **Data queries:** Requirements prescribe calculations, C_j for some j, with provisions for temporary storage and display.
- **Data "creation":** Requirements prescribe remaining calculations, C_j for other j, with provisions for longer-range storage and display.
- **"Completions":** Whereas the domain functions Q_i, C_f and predicates P_j may have been incomplete, their requirements definition must now be

"complete(d):" instantiated, determined, possibly extended and possibly fitted.

■

Figure 28.9 rather informally attempts to "diagram" which data a specifically required repository system shall maintain, and the functions that shall apply to those data.

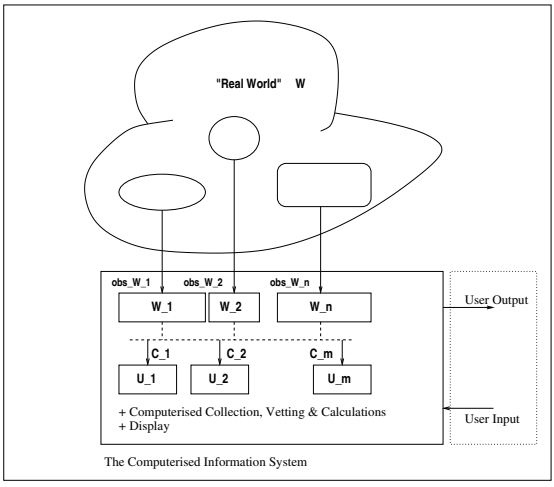


Fig. 28.9. Information system requirements

The full force of requirements engineering, of course, shall apply: One has to properly model, i.e., describe, either only informally, or both informally and formally, the requirements: the business process reengineering, the domain requirements (projection, determination, instantiation, extension and fitting), the interface requirements — which in the case of information repositories is a considerable fraction of the required software, and the machine requirements.

The above formula really only hints at such a prescription, and can thus be considered metalinguistic.

28.3.3 Information Repository Design

The Role of a Database Management System

Figure 28.10 shows the simplicity of the generic repository architecture. Complexity enters when we consider the ways in which the information is represented, for example, in the form of a `SQL` database, and the data vetting and query functions, P_j, Q_i, and modelling functions, C_f, are to be represented.

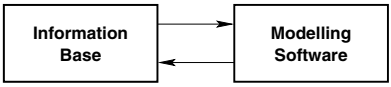


Fig. 28.10. Repository "architecture"

The information base component is usually a database management system while the modelling software typically is a workpiece-based subsystem. The "architecture" of the modelling software shall implement some of the information repository requirements while other information repository requirements are to be implemented by the database management system. Thus the modelling software shall implement the data collection and vetting, the data queries, the data "creation" and the data "completion" functionalities. The database management system shall implement the remaining information repository requirement, the data storage functionality.

A Relational Database Management System Architecture

We use the abbreviation RDMS to stand for relation[al] database management system.

Motivation

We remind the reader that in this section we are concerned only with the information base aspect of the repository "architecture". We shall motivate the seemingly "big jump" from the W's, Qi's and Wi's of Sects. 28.3.1 and 28.3.2 to the RDMS of this section. Each Qi and each Wi of Sects. 28.3.1 and 28.3.2 shall basically correspond to what is called a relation. Each Qi and each Wi is usually made up from component values. (This was not indicated in Sects. 28.3.1 and 28.3.2.) We now make the simplifying assumption that each value in each Qi and Wi is modelled as a Cartesian and that the element values of these Cartesians are atomic. The Cartesians become relation tuples of first normal form, i.e., of atomic values. This motivates the "big jump".

Summary of RDMS Architecture

Characterisation. By a *relational database management architecture* we mean a concept that accommodates a number of relations, for a number of relation schemas, and initialising, querying and updating relations. ∎

Characterisation. By a *relation* we shall understand a set of Cartesian values, also called tuples or rows, such that all tuples of a relation have the same arity. ∎

Characterisation. By a *relation tuple* we shall understand a Cartesian value. The number of element values of a tuple is called its arity. ∎

Characterisation. By *relation tuple arity* we shall understand the number of element values of the tuple. ▪

Characterisation. By a *relation tuple attribute* we shall understand the type of a relation tuple element value. ▪

Characterisation. By a *relation tuple attribute name* we shall understand a user-chosen name for the type of a relation tuple element value. ▪

Characterisation. By a *relation tuple element value* we shall understand the value at a specific position in the tuple. ▪

For any relation and for any two tuples of that relation we assume that elements from identical positions of the two tuples are of the same type.

Characterisation. By a *relation schema declaration* we understand the naming of a relation and a set of type definitions for each of its attributes. ▪

Characterisation. By a *relation database initialisation* we understand the creation by the user and the storage by the database management system of a number of relation schemas and the initial population of some or all of the relations named by the schemas and such that the relation tuple element values conform to the relation schema declaration. ▪

Characterisation. By a *relation query* we understand the computation of a relation or a tuple result where this result is computed — in some way or another — from relations in the database management system. ▪

Characterisation. By a *relation update* we understand the deletion of a relation or of tuples in a relation, or the insertion of new tuples in a relation based — in some way or another — on some criteria. ▪

What this "some way or another" is is a function, a property, of the specific database management system, more specifically, of its query and update language. We will return to this issue below.

Semantic Types of an RDMS

Formal Presentation: RDMS: Semantic Types, I

```
type
  VAL = BoolVAL | StringVAL | IntVAL | RealVAL
  BoolVAL == mkBool(b:Bool)
  StringVAL == mkString(s:Text)
  IntVAL == mkInt(i:Int)
  RealVAL == mkReal(r:Real)
```

```
  ValTyp == boolean | string | integer | float
  TUP = {| vl:VAL* • len vl≥1 |}
  TupTyp = {| vtl:ValTyp* • len vtl≥1 |}
  REL′ = TUPLE-set
  REL = {| r:REL′ • wf_REL(r) |}
  Rn
  RDB′ = Rn ~m~> (TupTyp×REL)
  RDB = {| rdb:RDB′ • wf_RDB(rdb) |}

value
  wf_REL: REL′ → Bool
  wf_REL(r) ≡ ∀ t,t′:TUP•{t,t′}⊆r⇒dom t=dom t′

  wf_RDB: RDB′ → Bool
  wf_RDB(rdb) ≡
    ∀(tt,rel):(TupTyp×REL) • (tt,rel)∈ rng rdb⇒wfR(tt,rel)

  wf_R: TupTyp×REL → Bool
  wf_R(tt,rel) ≡
    ∀ t:TUPE • t ∈ rel ⇒
      len t = len tt ∧ ∀ i:Nat • i ∈ inds t ⇒ tt(i)=xtr_type(t(i))

  xtr_type: VAL → ValTyp
  xtr_typ(v) ≡
    case v of
      mkBool(_) → boolean, mkString(_) → string,
      mkInt(_) → integer,    mkReal(_) → float
    end
```

Narration cum[3] Annotations

- An atomic value is either a Boolean, or a character string, or an integer, or a real value.
- These values are modelled as typed (i.e., tagged) values of appropriate kinds. (The mkBool, mkString, mkInt, mkReal constructor functions are the tags.)
- To each type of value there is a corresponding value type (ValTyp).
- Tuples are lists of values.
- The corresponding tuple type is a list of value types.
- A relation is a well-formed set of tuples.
- A relation[al] database is a well-formed set of uniquely relation-named pairs of tuple types (i.e., schema declaration) and relations.

[3] The term 'Narration' is used for those readers who skip the above formulas and the term 'Annotation' is used for those who do read the above formulas.

- For a relation to be well-formed means that all its tuples have exactly the same set of attribute names.
- For a relation[al] database to be well-formed means that every tuple of a relation conforms to its schema declaration.
- For a tuple to conform to its schema declaration means that for every attribute the tuple element value is of the declared type.
- The extract type function, xtr_typ, ensures conformance.

Syntactic Types

The motivation for the query language of SQL-like RDMSs is found in the usual set comprehension expression:

{ (a,b,...,c) | a:A,b:B,...,c:C • p(a,b,...,c) }

Either the individual a's, b's, ..., c's of the above set comprehension are element values of tuples or subsequences of these are tuple values of relations. That is, the individual A's, B's, ..., C's of the above set comprehension are attribute names, or subsequences of these are relation names. The predicates p are just that. The SQL-like query language of the RDMS being illustrated in this section now follows the above schematised set comprehension.

Formal Presentation: RDMS: Syntactic Types, II

```
type
   Rid, Tid
   Query′ = Targ* (Rid m→ Range) × Wff
   Query = {| q:Query′ • wfQ(q) |}
   Targ == mkRn(ri:Rid) | mkRnIdx(ri:Rid,i:Nat)
   Range == mkRnm(rn:Rn) | mkInfR(lr:Range,o:RelOp,rr:Range)
   RelOp == UNION | INTERSECT | COMPL
   Wff = QPre | IPre | NPre | APre
   QPre == mkQ(q:Quan,ti:Tid,rn:Rnm,cond:Wff)
   Quan == ALL | EXISTS
   IPre == mkI(lp:Wff,ao:BOp,rp:Wff)
   BOp == AND | OR
   NPre == mkN(pr:Wff)
   APre == mkA(lt:Term,ar:ARel,rt:Term)
   ARel == LESSEQ | LESS | EQUAL | NOTEQ | LARG | LARGEQ
   Term = Val | Elem
   Val == mkV(v:VAL)
   Elem == mkE(vt:RTid,i:Nat)
   RTid == mkR(ri:Rid) | mkT(ti:Tid)
```

Narration cum Annotations

- The token types Rid and Tid model various kinds of (free) identifiers. Rid's stand for relations (defined in range expressions) and Tid's stand for tuples (defined in quantified expressions, see below).
- A query consists of three parts:
  - ⋆ a target list, corresponding to the (a,b,...,c) expression of the set comprehension,
  - ⋆ a binding of variable identifiers to relations and
  - ⋆ the (well-formed formula, Wff) corresponding to the p(a,b,...,c) predicate of the set comprehension.
- The target list consists of either relation names or of tuple element index qualified relation names.
- A range expression either names a relation or is an infix expression denoting the union, the intersection or the complement of of the value of two range expressions.
- A well-formed formula is either a quantified, or an infix, or a negated, or an atomic well-formed formula.
  - ⋆ A quantified well-formed formula indicates the quantifier ($\forall, \exists$), identifies, by t:Tid, the name of an arbitrary, the quantified tuple in a named relation (rn:Rnm), and states the subsidiary well-formed formula (in which t is expected to occur free and range over the tuples of the named relation, rn:Rnm).
  - ⋆ An infix well-formed formula expresses the conjunction (AND) or the disjunction (OR) of two subsidiary well-formed formulas.
  - ⋆ A negated well-formed formula expresses the negation of a (the subsidiary) well-formed formula.
  - ⋆ An atomic well-formed formula expresses an arithmetic relation (less than, less than or equal, equal, not equal, larger than or equal, or larger than) between two term values.
- A term expression is either a simple value or stands for an indexed tuple element value.

Formal Presentation: RDMS: Syntactic Well-formedness Predicates, III

```
value
  attributes: Rnm → RDB → Nat-set
  attributes(rn)(rdb) ≡ let (tt,_) = rdb(rn) in inds tt end

  pre_E_Query: Query → RDB → Bool
  pre_E_Query(tal,rm,wff) ≡
    wfTargl(tal)(dict(rm,rdb))
   ∧ wfRanges(rm)(rdb)
   ∧ wfWff(wff)(rdb)(dict(rm,rdb))

type
  Δ = (Rid|Tid) ~m~> Nat-set
```

```
value
  dict: (Rid ~m~> Range) × RDB → Δ
  dict(rm,rdb) ≡
    [rn↦attrs(rm(rn),rdb)|rn:Rnm•rn ∈ dom rm]

  attrs: Range × RDB → Nat-set
  attrs(range,rdb) ≡
    case range of
      mkRnm(rn)→attributes(rnm)(rdb),
      mkInfR(lr,_,_)→attrs(lr,rdb)
    end

  wfTargl: Targ* → Δ → Bool
  wfTargl(tal)(δ) ≡ ∀ t:Targ • t ∈ elems tal ⇒ wfTarg(t)

  wfTarg: Targ → Δ → Bool
  wfTarg(t)(δ) ≡
    case t of
      mkRn(ri) → ri ∈ dom δ,
      mkRnAn(ri,i) → ri ∈ dom δ ∧ i ∈ δ(ri)
    end

  wfRanges: (Rid ~m~> Range) → RDB → Bool
  wfRanges(rm)(rdb) ≡
    ∀ range:Range • range ∈ rng rm ⇒ wfRange(rng)(rdb)

  wfRange: Range → RDB → Bool
  wfRange(range)(rdb) ≡
    case range of
      mkRnm(rn) →
          rn ∈ dom rdb,
      mkInfR(lr,_,rr) →
          wfRanges(lr)(rdb)∧wfRanges(rr)(rdb)
        ∧ attrs(lr)(rdb)=attrs(rr)(rdb)
    end

  wfWff: Wff → RDB → Δ → Bool

  wfWff(wff)(rdb)(δ) ≡
    case wff of
      mkQ(_,ti,rn,pr) →
        rn ∈ dom rdb
        ∧ wfWff(pr)(δ†[ti↦attrs(rn)(rdb)]),
      mkI(lp,_,rp) →
```

```
            wfWff(lp)(rdb)(δ) ∧ wfWff(rp)(rdb)(δ),
        mkN(pr) →
            wfWff(pr)(rdb)(δ),
        mkA(lt,ar,rt) →
            wfTerm(lt)(δ)∧wfTerm(rt)(δ)
    end

wfTerm: Term → Δ → Bool
wfTerm(trm)(δ) ≡
    case trm of
        mkV(_) → true,
        mkE(mkR(ri),i) → ri ∈ dom δ ∧ i ∈ δ(ri),
        mkE(mkT(ti),i) → ti ∈ dom δ ∧ i ∈ δ(ti)
    end
```

Narration cum Annotations

- The function attributes yields a set of attributes of named relations.
- For a query to be evaluated its precondition for evaluation must hold:
  - ⋆ the target list must be well-formed,
  - ⋆ the range expressions must be well-formed and
  - ⋆ the predicate (wff) must be well-formed.

  The well-formedness of these syntactic quantities depends on a context.
  - ⋆ For target list well-formedness the context is a dictionary, Δ, which maps identifiers of tuples into index sets.
  - ⋆ For the range expression well-formedness the context is the relational database.
  - ⋆ And for the predicate well-formedness the context is both the dictionary mentioned above and the database.
- The function dict creates from the range expressions and the database a dictionary. It does so using the auxiliary function attrs. For every relation name, rn, of the range expression map, rm, attrs extracts the attribute names for that relation.
- The function attrs should be reasonably self-explanatory.
- A target list is well-formed if all of its target expressions are well-formed in the same dictionary context.
  - ⋆ A target expression is either a simple relation identifier which must then be defined in the dictionary,
  - ⋆ or it is a pair of a relation identifier and an attribute name where the formed must be defined in the dictionary and the latter must be in the definition set of attribute names for that relation identifier.
- The range expression map (from relation identifiers to range expressions) is well-formed if all of the range expressions are well-formed.

  - ⋆ A range expression is either just the name of a relation which must then be defined in the database,
  - ⋆ or it is an infix range expression both of whose range expressions must be well-formed.
- The well-formedness of a predicate expression wff depends on the kind of expression it is.
  - ⋆ If wff is a quantified expression then its relation name, rn, must be defined in the database and the contained wff, pr, must be well-formed in a context which keeps the database but updates the dictionary to map the quantified tuple variable to the set of attribute names of the relation rn.
  - ⋆ If wff is an infix predicate expression then both predicate expression operands must be well-formed.
  - ⋆ If wff is a negated predicate expression then that predicate expression operand must be well-formed.
  - ⋆ Finally, if wff is an atomic expression of two terms (and an arithmetic relation operator) then the two terms must be well-formed.
- The well-formedness of a term depends on the kind of expression it is.
  - ⋆ A simple term value is always well-formed.
  - ⋆ A term reference of a relation or a tuple identifier and a tuple element index is well-formed if
    - · the relation or a tuple identifier is defined in the dictionary,
    - · and the index is in the definition set of that identifier (in the dictionary).

Formal Presentation: RDMS: Evaluation Functions, IV

```
type
   V_Rs = Rid ~m~> REL
   VRs = (Rid ~m~> TUP)-set

value
   G: V_Rs → VRs
   G(vrs) ≡
      if vrs=[] then {[]}
      else { [v↦t] ∪ m | v:Rid,t:TUP •
                  v ∈ dom vrs∧t ∈ vrs(v)∧m ∈ G(vrs\{v})} end

   C: Targ* × VRs → TUP*
   C(tal)(vrs) ≡
      ⟨ case tal(i) of
            mkRn(rn) → vrs(rn),
            mkRnAn(rn,i) → ⟨(vrs(rn))(i)⟩ end | i in [1..len tal] ⟩

   Conc: TUP* → TUP
```

```
  Conc(tupl) ≡ if tupl=⟨⟩ then ⟨⟩ else hd tupl ⌢ Conc(tl tupl) end
```

Some technicalities:

- The functions G, C and Conc are auxiliary.
- G: For each combination of (a1,b1j),...,(an,bnj) in [a1↦{b11,...,b1m1},a2↦{b21,...,b2m2},...,an↦{bn1,... bnmn}] G delivers the map [a1↦b1j,...,an↦bnj] in the set G(rm) of such maps.
- C: For a pair of a list ⟨a′,a″,...,a‴⟩ and a map m:[a1↦b1j,...,an↦bnj] C delivers a tuple ⟨ m(a′),m(a″),...,m(a‴)⟩.
- Conc take a list of tuples and produces a tuple.

```
value
  E_Query: Query → RDB → REL
  E_Query(tal,rm,wff)(rdb) ≡
    let v_rs = [v↦E_Range(rm(v))(rdb)|v:Vid•v ∈ dom rm] in
    {Conc(C(tal,m))|m:VRS•m ∈ G(v_rs)∧E_Pred(wff)(m)(rdb)}
    end

  E_Pred: Wff → VRs → RDB → Bool
  E_Pred(wff)(vrs)(rdb) ≡
    case wff of
      mkQ(q,ti,rn,pr) →
        let (_,rel) = rdb(rn) in
        case q of
          ALL → ∀ tup:TUP • tup ∈ rel
                        ∧ E_Pred(pr)(vrs † [ti↦tup])(rdb),
          EXISTS → ∃ tup:TUP • tup ∈ rel
                        ∧ E_Pred(pr)(vrs † [ti↦tup])(rdb)
        end end
      mkI(lp,bo,rp) →
        let lb = E_Pred(lp)(vrs)(rdb), rb = E_Pred(rp)(vrs)(rdb) in
        case bo of AND → lb∧rb, → OR → lb∨rb end,
      mkN(pr) → ∼E_Pred(pr)(vrs)(rdb),
      mkA(lt,ao,rt) →
        let lv = E_Term(lt)vrs, rv = E_Term(rt)vrs in
        case ao of
          LESSEQ → lv≤rv, LESS → lv<rv, EQUAL → lv=rv,
          NOTEQ → lv≠rv, LARG → lv>rv, LARGEQ → lb≥rv
    end end end end

  E_Term: Term → VRs
  E_Term(t)vrs ≡
    case t of
      mkV(v) → v,
      mkE(vt,i) → (vrs(vt))(i),
```

```
        mkT(ti) → (vrs(ti))(i),
    end
```

Narration cum Annotations

- The evaluation of a query, E_Query, results in a relation.
  - ⋆ Query evaluation takes place in the context of the state of the database,
  - ⋆ and makes use of an auxiliary evaluation function E_Pred,
  - ⋆ and the auxiliary functions G, C and Conc.
  - ⋆ G is applied to the range definitions and yields a set of mappings from range identifiers to tuples.
  - ⋆ For each such mapping, m, E_Pred is applied to the query wff and the database. If true, then the mapping m is made into a tuple of tuples by means of the target list and using function C.
  - ⋆ Finally the tuple of tuples is "straightened out" into a simple tuple using function Conc.
  - ⋆ And this is done for all mappings m, hence generating a set of simple tuples, i.e., a relation.
- Evaluation, E_Pred, of a predicate proceeds according to the form of the predicate.
  - ⋆ The predicate wff:mkQ(ALL,ti,rn,pr) holds if the predicate pr holds for ti being bound to every tuple, tup, in the rn named relation in the database rdb.
  - ⋆ The predicate, wff:mkQ(EXISTS,ti,rn,pr) holds if the predicate pr holds for ti being bound to some tuple, tup, in the rn named relation.
  - ⋆ The predicate wff:mkI(lp,AND,rp) holds if both the predicates lp and rp holds.
  - ⋆ The predicate wff:mkI(lp,OR,rp) holds if either of the predicates lp or rp hold.
  - ⋆ The predicate wff:mkN(pr) holds if pr does not hold.
  - ⋆ The atomic expression wff:mkA(lt,ao,rt) holds if the values of the terms lt and rt stand in the relation designated by the arithmetic relation operator ao.
- Evaluation, E_Term, of a term, t, proceeds according to the form of the term t.
  - ⋆ The term t:mkV(v) evaluates to that value v.
  - ⋆ The term t:mkE(vt,i) evaluates to the i indexed tuple value designated by vt.
  - ⋆ The term t:mkT(ti) evaluates to the i indexed tuple value designated by ti.

Other Database Management System Architectures and Discussion

There are, or rather were, other database management system models than the relational database model detailed above.[4] The first reasonably "clean" model was the hierarchical database model "pioneered" by IBM in the form of the Information Management System (IMS) product that first appeared in the late 1960s [38, 80, 81]. The second reasonably "clean" model was the network database model "pioneered" by the Conference of Data and Systems Language (CODASYL) Database Task Group (DBTG) report [79–81, 220]. Chris Date was the first to clearly express the hierarchical and the network database models [80, 81]. These and the relational database models were formalised in [33, 38, 39]. The beauty of the relational database model is due to Ted Codd [61].

28.4 Client/Server Architectures

The client/server concept is not one, but several related, concepts. In this section we shall present a number of models of those concepts. First, we will present two rather simple formal models not involving `RSL/CSP` input/output. Then we will present basically the same kind of models but now with `RSL/CSP` input/output. That is, this section will, necessarily, require formalisation, at least in the form of simple `RSL/CSP` "programs".

28.4.1 Client/Server Domain/Requirements Models

First, we present models of what might be termed a single-client, single-server concept. This is followed by a multiple-client, single-server concept and, finally, a multiple-client, multiple-server concept. Then we present a series of more detailed models of what might be meant by servers: certain kinds of application systems (or, if you like that jargon, subsystems[5]). That is, we present application systems as collections of components or modules, without, respectively with, end-user event notification, and, if the latter, either directly and immediately upon event occurrence, or synchronously, i.e., still "immediately", but via a shared "event manager" component, or asynchronously, still via a shared "event manager" component, but not as polled by this component.

[4] We write "were" as these other models seem out of fashion today, January 2006.

[5] It seems that some software engineers cum programmers cum computing scientists can get all "hung up", i.e., excited, about, otherwise imprecisely stated, and most probably also confused, delineations of systems and subsystems, parts and whole, and what have you!

Single-Client, Single-Server Domain/Requirements Architecture

The single-client, single-server architecture — superficially speaking — reminds us of the repository architecture.

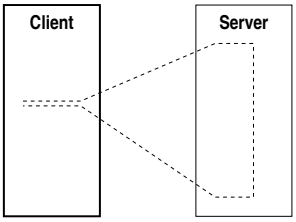


Fig. 28.11. A single-client, single-server system

The idea, colloquially speaking, is that the client needs something to help perform one of a limited variety of services. That something we shall call a server.

Formal Presentation: A Single-Client, Single-Server "System", I

```
type
   Γ, VAL
   Service == s1 | s2 | ... | sn
value
   γ:Γ

   system: Γ → Unit
   system(γ) ≡ client(γ)

   client: Γ → Unit
   client(γ) ≡
      let s = s1 ⫿ s2 ⫿ ... ⫿ sn in
      client(update(s,server(request(s)(γ)))(γ)) end

   update: Service × VAL → Γ → Γ
```

There are further unexplained notions of client state: γ:Γ, and values VAL. A definite number of different services can be offered: s_i. The single-client/single-server system is modelled as a client. The client (internally) nondeterministically selects a service: s. That service is requested of the server function, and the client state is updated with that request and the state resulting from the

server function applying the request on the client state γ, whereupon the client "starts all over" again!

Formal Presentation: A Single-Client, Single-Server "System", II

```
type
  Req = R1 | R2 | ... | Rn
  ..., Ri == mkRi(...), ...
value
  request: Service → Γ → Req
  request(s)(γ) ≡
    case s of
      s1→make_r1(γ),s2→make_r2(γ),...,sn→make_rn(γ)
    end
  server: R → VAL
  server(r) ≡
    case r of
      mkR1(...)→srv1(r),mkR2(...)→srv2(r),...,mkRn(...)→srvn(r)
    end
  make_r1: Γ → R1, make_r2: Γ → R2, ..., make_rn: Γ → Rn
  srv1: R1 → VAL, srv2: R2 → VAL, ..., srvn: Rn → VAL
```

The request function simply takes a Service token, s_i, and makes a proper request — usually one that provides arguments for the selected service. We do not detail how that provision, $make\_r_i$, is achieved. The server function, depending on the service request, mkR_i, then performs the actual service function, srv_i, which is not detailed.

Multiple-Client, Multiple-Server Domain/Requirements Model

We leave it to the reader to decipher and thus explain the next model (see Fig. 28.12).

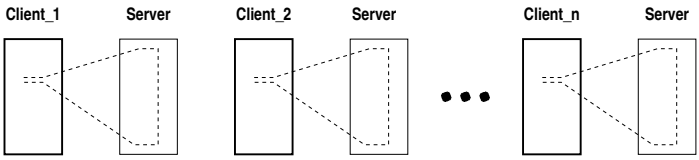


Fig. 28.12. A multiple-client, multiple-server system

Formal Presentation: Multiple-Client, Multiple-Server Model

```
type
   Γ, VAL, CIdx
   Ψ = CIdx ~m~> Γ
   Service == s1 | s2 | ... | sn
   R = R1 | R2 | ... | Rn
   ..., Ri == mkRi(...), ...
value
   ψ:Ψ

   system: Γ → Unit
   system(γ) ≡ || { client(i)(ψ(i)) | i:CIdx }

   client: CIdx → Γ → Unit
   client(i)(γ) ≡
      let s = s1 [] s2 [] ... [] sn in
      client(i)(update(i,s,server(request(i,s)(γ)))(γ))
      end

   update: CIdx × Service × VAL → Γ → Γ

   request: CIdx × Service → Γ → REQ
   request(i,s)(γ) ≡
      case s of
         s1→mkr_1(i)(γ),s2→mkr_2(i)(γ),...,sn→mkr_n(i)(γ)
      end

   server: R → VAL
   server(r) ≡
      case r of
         mkR1(...)→srv_1(r),mkR2(...)→srv_2(r),...,mkRn(...)→srv_n(r)
      end

   ..., mkr_i: CIdx → Γ → Ri, ..., srv_i: Ri → VAL, ...
```

28.4.2 Some Meta-RSL/CSP Constructs

Before presenting the various models now to be expressed using the RSL/CSP specification language constructs, let us first present some notational shorthand. These concepts are introduced and used in order, it is hoped, to better highlight the interaction between *clients* and *servers*.

Motivation and Justification

Between two process behaviours, P, Q, one often observes any of the following interactions:

- (0) input, compute, output: P expects some input from Q; once the input, m, has been obtained (from Q), P computes over m; and outputs a result v to Q.
- (1) output, input: P outputs a value v to Q; P then proceeds to await input from Q; once the input, m, has been obtained (from Q), P proceeds (with whatever it wishes to do [with v]).
- (2) P only outputs a value v to Q, and otherwise proceeds (with whatever it wishes to do).
- (3) P only inputs a value v from Q, and otherwise proceeds (with whatever it wishes to do [with v]).

Three Process Interaction Macros

Cases (0) and (1) are inverses of each other. The four cases thus justify our introducing three macros, in a somewhat reverse order:

output: Channel_name × ARG → **Unit**
input: Channel_name → RES
output_input: Channel_name × ARG → RES

The four cases above, (0)–(3), "translate" into the following uses of the macros:

(0) **let** val = **input**(ch) **in** **output**(ch,compute(val)) **end**
(1) **output**(ch,v) ; **let** val = **input**(ch) **in** ... val ... **end**
(2) **output**(ch,v)
(3) **let** val = **input**(ch) **in** ... val ... **end**

Next we will more properly define the macros, then we will illustrate their use in different client/server models.

'Meaning' of the Three Process Interaction Macros

We explain the intended meaning of what the macros designate: **output**(ch,v) prescribes the output of any value v on(to) channel ch. **input**(ch) prescribes the input of any value on (i.e., from) channel ch. **output_input**(ch,v) prescribes the sequential actions first of the output of any value v on(to) channel ch, then, immediately thereafter, of the input of any value on (i.e., from) that same channel ch. Here ch is thought of as a channel over which P and Q may synchronise and exchange messages, i.e., engage in rendezvous.

```
type
  ARG, RES
  MSG = ARG | RES
  Channel_name == c1 | c2 | ... | cn
channel
  c1,c2,...,cn:MSG
macros:
  output: Channel_name × ARG → Unit
  output(ch_n,a) ≡ ch_n ! a

  input: Channel_name → RES
  input(ch_n) ≡ ch_n ?

  output_input: Channel_name × ARG → RES
  output_input(ch_n,a) ≡ output(ch_n,a) ; input(ch_n)
```

Pragmatics of Macros

By a macro we mean a function which, when applied to an argument value of the kind designated by its formal parameters, yields the text prescribed by the body of the macro as evaluated in the calling environment, except for the binding of the formal parameter identifiers to actual arguments. The latter is in contrast to λ-expressions, whose bodies are evaluated in the defining environment.

Thus wherever a macro, with arguments, appears, substitute the formal parameter identifiers of its body with the actual arguments and yield the resulting text. Use that text instead of the macro "call", and then evaluate this text as any other RSL text is evaluated.

28.4.3 Single-Client, Single-Server Model

A single-client, single-server configuration of processes is basically just a decomposition of one problem (or process) into two. The decomposition might be done for several reasons (one or all): to express separation of concerns, to simplify the overall configuration by simplifying each of its two postulated components, and by emphasising a "clean" interface between them, or to prepare for more complicated concepts — such as those shown in subsequent sections.

We present two versions of the single-client, single-server model. We present one in which the client offers interaction, of one of n kinds, over one channel with the single-server; and we present one in which it offers basically the same n kinds of interaction content, but not forms, over n distinct channels.

Example 28.3 *An Abstract Single-Client, Single-Server Model, I:* Each of the components (client and server) has its own state. The server may, nondeterministically internally, invoke different services from the one server it here interacts with.

```
type
  Γ, Σ, VAL
  R = R1 | R2 | ... | Rn
  ..., Ri == mkRi(...), ...
  Service == s1 | s2 | ... | sn
channel
  cs:M
value
  γ_0:Γ, σ_0:Σ

  system: Unit → Unit
  system() ≡ client(γ_0) || server(σ_0)

  client: Γ → in,out cs  Unit
  server: Σ → in,out cs  Unit

  client(γ) ≡
    let s = s1 ⌈] s2⌈] ... ⌈] sn in
    let r = request(s)(γ) in
    let v = output_input(cs,r) in
    client(c_update(r,v)(γ))
    end end end

  c_update: R × V → Γ → Γ

  request: Service → Γ → R
  request(s)(γ) ≡
    case s of
      s1 → mR1(...),
      s2 → mR1(...),
      ...,
      sn → mR1(...)
    end
```

The request r is of any of the n forms mkRi(...). This is reflected in the case server distinction.

```
value
  server(σ) ≡
    let r = input(cs) in
```

```
    let v =
      case r of
        mkR1(...) → serve_1(r)(σ),
        mkR2(...) → serve_2(r)(σ),
        ...
        mkRn(...) → serve_n(r)(σ)
      end in
    output(cs,v);
    server(s_update(r,v)(σ))
    end end

  ..., serve_i: Ri → Σ → VAL, ..., s_update: R × VAL → Σ → Σ
```

■

Example 28.4 *An Abstract Single-Client, Single-Server Model, II:* The client's internally nondeterministic choice of which service to request leads to a communication offer over a client/server channel specific to the request.

```
type
  Γ, Σ, VAL
  R = R1 | R2 | ... | Rn
  Service == s1 | s2 | ... | sn

channel
  cs1:R1,cs2:R2,...,csn:Rn

value
  γ_0:Γ, σ_0:Σ

  system: Unit → Unit
  system() ≡ client(γ_0) || server(σ_0)

  client: Γ → out,in cs1,c2,...,csn  Unit
  server: Σ → in,out cs1,c2,...,csn  Unit
```

As the channel distinction is sufficient, there is no need to also secure distinctness of type of communicated requests. That is, R need not be a union of discriminated types.

One expression form

```
value
  client(γ) ≡
    let s = s1 ⌈⌉ s2 ⌈⌉ ... ⌈⌉ sn in
```

```
    let v = case s of
        s1 →
            let r1 = request_1(s1)(γ) in
            output_input(cs1,mkR1(m)) end,
        s2 →
            let r2 = request_2(s2)(γ) in
            output_input(cs1,mkR2(m)) end,
        ...
        sn →
            let rn = request_n(sn)(γ) in
            output_input(cs1,mkRn(m)) end
        end in
    client(c_update(s,v)(γ))
    end end
```

Another expression form

```
value
  client(γ) ≡
    let v =
            let r1 = request_1(s1)(γ) in
            output_input(cs1,mkR1(m)) end
        ⌈⌉
            let r2 = request_2(s2)(γ) in
            output_input(cs1,mkR2(m)) end
        ⌈⌉
        ...
        ⌈⌉
            let rn = request_n(sn)(γ) in
            output_input(cs1,mkRn(m)) end
    in client(c_update(s,v)(γ)) end

  ..., request_i: Service → Γ → Ri, ...,
  c_update: Service×V → Γ → Γ
```

The server is prescribed in terms of n externally nondeterministic clauses:

```
  server(σ) ≡
    let r1 = cs1? in
    let v1 = serve_1(r1)(σ) in
    cs1!v1 ; server(s_update(r1,v1)(σ)) end end
  ⌈⌉
    let r2 = cs2? in
    let v2 = serve_2(r2)(σ) in
```

```
      cs2 ! v2 ; server(s_update(r2,v2)(σ)) end end
  ⌷
      ...
  ⌷
      let rn = csn ? in
      let vn = serve_n(rn)(σ) in
      csn ! vn ; server(s_update(rn,vn)(σ)) end end

  ..., serve_i: Ri → Σ → VAL, ..., s_update: R> VAL → Σ → Σ
```

In this second version the internal nondeterminism of the client is "matched" by an external nondeterminism of the server. ▪

We leave it to the reader to further study the last two examples.

28.4.4 Multiple-Client, Single-Server Model

Again we challenge the reader to decipher and explain the formulas.

Example 28.5 *An Abstract Multiple-Client, Single-Server Model:*

Program Specification: Multiple-Client, Single-Server Model

```
type
  Γ, Σ, VAL, CIdx
  R = R1 | R2 | ... | Rn
  ..., Ri == mkRi(...), ...
  Service == s1 | s2 | ... | sn
channel
  { cs[ c ]:M | c:CIdx }
value
  γ_0:Γ, σ_0:Σ

  system: Unit → Unit
  system() ≡ || { client(c)(γ_0) | c:CIdx } || server(σ_0)

  client: CIdx → Γ → in,out cs  Unit
  server: Σ → in,out cs  Unit

  client(c)(γ) ≡
      let s = s1 ⌈| s2 ⌈| ... ⌈| sn in
      let r = request(s)(γ) in
      let v =output_input(cs[ c ],r) in
      client(c)(c_update(r,v)(γ)) end end end
```

```
server(σ) ≡
  [] { let r = input(cs[c]) in
       let v =
         case r of
           mkR1(...) → serve_1(c,r)(σ),
           mkR2(...) → serve_2(c,r)(σ),
           ...
           mkRn(...) → serve_n(c,r)(σ)
         end in
       output(cs[c],v); server(s_update(c,r,v)(σ))
       end end | c:CIdx }

request: Service → Γ → R,
c_update: R → Γ → Γ
...,
serve_i: CIdx×Ri → Σ → VAL,
...
s_update: CIdx×R×VAL → Σ → Σ
```

■

28.4.5 Client/Server Event Manager Model

Typical, and at least "classical", application systems, A, offer many (i.e., n) oftentimes diverse services A_i to users. Users (u) invoke these services through an interface, I, to A. The services, A_i, are here modelled as individual processes. A is modelled as the parallel composition of all the A_i in parallel with I. I interprets a user request and passes that request on to the appropriate application package, A_i.

An application package, A_i, may (i) handle a user request "all by itself", without a need for invoking other application packages A_j; or may (ii) request another application package A_j to perform some service to A_i, which then returns the result of the originally requested service via I to the requesting user. (iii) Or A_i may, after some computation, pass on a thus obtained intermediate service result to another application package A_k, which will then "complete" the service request and return the result of the originally requested service via I to the requesting user. (iv) Or the originally requested service may be handled by a (dynamically well-formed) composition of two or more of subservices (ii) followed by a subservice (iii) — which again may lead to a composition of the form (iv). We shall model the above in Example 28.6.

In addition to the above service/subservice concept, we additionally lead up to a concept of *application event notification management*. By an *application event notification* we here mean one of the following — explained relative

to the application system roughly outlined above — during computation by any subservice (A_i), A_i may wish to *notify* the requesting user u of normal, or abnormal request handling; or A_i may wish to *notify* some other stakeholder of normal, or abnormal request handling with respect to user u; or A_i may wish to *"immediately" notify* some other subservice; or A_i may wish to *"time delayed" notify* some other subservice. *Notifications* are thought of as messages sent to users, respectively "other stakeholders" — for example in the form of e-mails, SMS messages, letters, voice-recorder telephone messages, or other. *Notifications* have no immediate effect: It is up to the notified to decide on that. Such effects may then occur immediately, i.e., require parallel action, or be delayed (say queued).

The next example leads up to later assignments, which illustrate a variety of "solutions" to the above sketched event notification management problem. Example 28.6 itself illustrates the case of no event notification "management". Exercise 28.7 addresses the case of direct A_i event notification management. Exercise 28.8 addresses the case of event notification management via an event manager service E_{synch} such that A_i requests this service via (i.e., from) E_{synch} — where E_{synch} is expected to perform the event notification service synchronously (i.e., "immediately" upon request). Exercise 28.9 addresses the case of deferred event notification management by an event manager service E_{asynch} (invoked by I when I also invokes A_i) — where E_{asynch} itself discovers the need for also "raising" an, or the, event. E_{asynch} may handle this event notification service asynchronously, i.e., "out of step" with A_i's handling of the original service request.[6]

We present a general model of an archetypical application system as consisting of several, separately invokable application functions. We do not highlight any interaction between any one specific application function (A_i, i.e., application package) and an invoking user. There might be such user/A_i interactions, but we ignore them.

Example 28.6 *An Application System:* The application system A consists of n separately invokable application packages A_i (for i in the index Aidx set of n indices); an interface/interpreter I which interfaces between A, i.e., the n separately invokable application packages A_i and the users; and a storage S shared between all A_i. The individual A_i may have local, "scratch" storage, as designated by the various v, v', v'', etc., below, but it is the shared storage that all A_i can access. You may think of S as a shared database.

The model otherwise revolves around an index set Uidx of users; channels, cui[i], between users and the common interface I; an index set Aidx of application packages; channels, cia[i], between I and A_is; channels, caa[i,j], between

[6] These assignments need not be answered formally, but convincing narratives need be given of how you would structure your event management processes.

any pair[7] of distinct A_is — such that $i > j$; and channels, cas[i], between any A_i and the shared storage S.

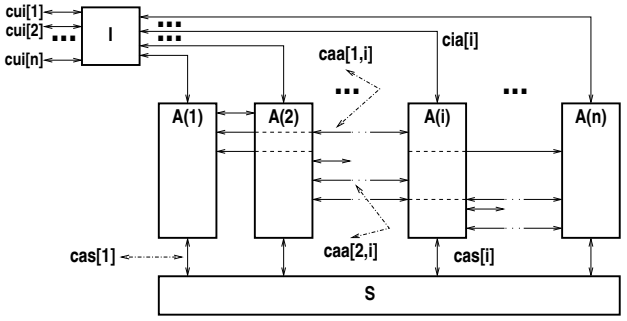


Fig. 28.13. A schematic diagram: the I, A(i) and S process and channel complex

Figure 28.13 is intended to diagram a generic application system. The figure is matched by Figs. 28.14 and 28.15 at the end of Example 28.6.

Program Specification: An Application System

```
type
    Σ, AIdx, UIdx, R, V, L
    M == stop
        | mkReadCoUpdRet(m:V)
        | mkReadCoCallCoUpdRet(m:V)
        | mkReadCoCallCoCallCoUpdRet(m:V)
        | mkReadCoUpdPassOn(m:V)
        | ...
    RW = Read | Write
    Read == mkRead(ℓ:L)
    Write == mkWrite(ℓ:L,v:V)
    LV = L×V
channel
    { cui[u]:(R|V) | i:Uidx }, { cia[i]:(M|V) | i:Aidx }
    { caa[i,j]:M | i,j:Aidx • i>j }, { cas[i]:(RW|V) | i:Aidx }

value
    σ_0:Σ
```

[7] One does not here need more than one channel between any pair of distinct A_is.

```
System: Unit → in,out { cui[u]:(R|V) | i:Uidx }  Unit
System() ≡ A() ‖ S(σ_0)

A: Unit → Unit,
A() ≡ (I() ‖ { Ai(i) | i:AIdx }) ; A()
```

The definition A()≡(I()‖{Ai(i)|i:AIdx});A() reflects that once one user request has been completed all Ai(i) and the I() are to have completed, and must hence be restarted. Figure 28.13 schematically shows the complex of channels and processes. Figures 28.14 and 28.15 follow up on Figure 28.13. I() potentially communicates with all users and all application packages. **let** r = **input**(cui[u]) **in** designate receipt of a request, r, from user u. **let** (j,m) = analyse(r) **in** designate analysis of the request r, resulting in identification of the application package A_j and the message m being passed to A_j. **output**(cia[j],m) designates invocation of A_j. The nondeterministic external choice □{**let** v=**input**(cia[k]) **in output**(cui[u],v) **end**|k:AIdx} designates receipt of the final service result from some application package A_k — and the communication of this final result to the initially requesting user. (Although the application package first invoked was A_i, the final result may emerge from another application package, hence A_k.) Finally, the initialiser "resets" all application packages to **stop** via the message stop.

Program Specification: An Application System (Continued)

```
value
  I: Unit →
      in,out {cui[u]:(R|V)|u:Uidx}
      out,in {cia[i]:(M|V)|i:Aidx}  Unit
  I() ≡
     □ { let r = input(cui[u]) in
         let (j,m) = analyse(r) in
         output(cia[j],m);
         □{let v=input(cia[k]) in output(cui[u],v) end|k:AIdx};
         ‖{output(cia[i],stop)|i:AIdx•i≠j}
         end end | u:UIdx }
```

When I() has stopped all application packages it terminates itself. These stopping and termination actions are mere technicalities. The definition A()≡(I()‖{Ai(i)|i:AIdx});A() then prescribes that A() is reinvoked: The system is made ready for a next user request.

The shared storage behaviour expresses willingness to engage in updates (writes) or reads with any application package. The storage behaviour S() is

composed in parallel with the application behaviour A(). Hence S() recurses after any read/write — in the latter case with a potentialy updated state σ.

Program Specification: An Application System (Continued)

```
value
  S: Σ →
       in,out {cas[i]|i:AIdx} Unit
  S(σ) ≡
    [] { let m = input(cas[i]) in
         case m of
           mk_Write(ref,val) →
             S(update_stg(ref,val)(σ)),
           mk_Read(ref) →
             output(cas[i],read_stg(ref)(σ)) ; S(σ)
         end end | i:AIdx }

  update_stg: L × V → Σ → Σ
  read_stg: L → Σ → V
```

The A(i) behaviour is prescribed "generically". That is, the four possibilities are just examples. There could be other possibilities. We leave definition of such other possibilities to the reader. The details of all auxiliary functions are left unspecified, except for their signatures. Any one A(i) may allow any subset of the below functionalities. The main function definition, A(i), selects any one of the four functionalities based on input from the interface interpreter or from other application packages (A(j), A(k)). We leave it to the reader to otherwise decipher the four alternatives.

Program Specification: An Application System (Continued)

```
  Ai: i:AIdx → in,out {caa[j,i]|j:AIdx•j>i}, cas[i], out cai[i]  Unit
  Ai(i) ≡
    let m = input(cia[i]) in
    if m ≠ stop
    then case m of
      mkReadCoUpdRet(v) →
        rd_comp_upd_return(i,v),
      mkReadCoCallCoUpdRet(v) →
        rd_comp_call_comp_upd_return(i,v),
      mkReadCoCallCoCallCoUpdRet(v) →
        rd_comp_call_comp_call_comp_upd_return(i,v),
      mkReadCoUpdPassOn(v) →
        rd_comp_upd_pass_on(i,v),
```

```
        ...
        end ; Ai(i)
    else skip end end
```

Each of the four alternative functionalities is now described. rd_comp_upd_return reads from storage; computes; updates storage, while returning the result to the user via the interpreter.

Program Specification: An Application System (Continued)

```
value
  rd_comp_upd_return:
    i:AIdx × V → out,in cas[i], out cia[i] Unit
  rd_comp_upd_return(i,v) ≡
    let v′ = output_input(cas[i],mk_Read(rcur_get_loc(i,v))) in
    let v″ = rcur_compute(i,v) in
    ( output(cas[i],mk_Write(rcur_get_loc_val(i,v″))) ||
      output(cia[i],v″) ) end end

  rcur_get_loc: AIdx × V → Read
  rcur_compute: AIdx × V → V
  rcur_compute: AIdx × V × AIdx → V
  rcur_get_loc_val: AIdx × V → Write
```

rd_comp_call_comp_upd_return reads from storage; computes a partial (own) result; invokes (i.e., calls) another application package; and receives a partial result from that other package. It then further computes the final result; and updates storage, while returning the result to the user via the interpreter.

Program Specification: An Application System (Continued)

```
value
  rd_comp_call_comp_upd_return:
    i:AIdx × V → out,in cas[i], out cia[i],
                 out,in {caa[j,i]|j:AIdx•j>i} Unit
  rd_comp_call_comp_upd_return(i,v) ≡
    let v′ = output_input(cas[i],rcccur_get_loc(i,v)) in
    let (j,v″) = rcccur_compute_1(i,v′) in
    let v‴ = output_input(caa[i,j],rcccur_call_fct(i,v″)) in
    let v⁗ = rcccur_compute_2(j,v‴) in
    ( output(cas[i],rcccur_get_loc_val(i,v⁗)) ||
      output(cia[i],v⁗) ) end end end end
```

```
rcccur_get_loc: AIdx × V → Read
rcccur_compute_1: AIdx × V → AIdx × V
rcccur_call_fct: AIdx × V → V
rcccur_compute_2: AIdx × V → V
rcccur_get_loc_val: AIdx × V → Write
```

rd_comp_call_comp_call_comp_upd_return reads from storage; computes the first part of a partial (own) result; invokes (i.e., calls) another application package, and receives the partial result from that other package. It then computes the second part of the partial (own) result; invokes (i.e., calls) another application package, and receives a partial result from that other package. It then further computes the final result; and updates storage, while returning the result to the user via the interpreter.

Program Specification: An Application System (Continued)

```
value
  rd_comp_call_comp_call_comp_upd_return:
    i:AIdx × V → out,in cas[i], out cia[i],
                    out,in {caa[j,i]|j:AIdx•j>i}  Unit
  rd_comp_call_comp_call_comp_upd_return(i,v) ≡
    let v′ = output_input(cas[i],rcccur_get_loc(i,v)) in
    let (j,v″) = rcccur_compute_1(i,v′) in
    let v‴ = output_input(caa[i,j],rcccur_call_fct_1(i,v″)) in
    let (k,v⁗) = rcccur_compute_2(j,v‴) in
    let v⁗′ = output_input(caa[i,k],rcccur_call_fct_2(i,v⁗)) in
    let v⁗″ = rcccur_compute_3(k,v⁗′) in
    ( output(cas[i],rcccur_get_loc_val(i,v⁗″)) ||
      output(cia[i],v⁗″) ) end end end end end end

  rcccur_get_val: AIdx × V → Read
  rcccur_compute_1: AIdx × V → AIdx × V
  rcccur_call_fct_1: AIdx × V → V
  rcccur_compute_2: AIdx × V → V
  rcccur_call_fct_2: AIdx × V → V
  rcccur_compute_3: AIdx × V → V
  rcccur_get_loc_val: AIdx × V → Write
```

rd_comp_upd_pass_on reads from storage; computes an intermediate (own) result; and updates storage, while passing the intermediate result onto another

application package. This "other" application package takes over further control: No return will take place to A(i) — which terminates.

Program Specification: An Application System (Continued)

```
value
  rd_comp_upd_pass_on:
    i:AIdx × V → in,out cas[i], out cia[i],
                        in,out {caa[j,i]|j:AIdx•j>i}  Unit
  rd_comp_upd_pass_on(i,v) ≡
    let v′ = output_input(cas[i],rcupo_get_loc(i,v)) in
    let v″ = rcupo_compute(i,v,v′) in
    ( output(cas[i],rcupo_get_loc_val(i,v″)) ||
      output(caa[i,j],rcupo_pass_on_val(i,v″)) ) end end

  rcupo_get_loc: AIdx × V → Read
  rcupo_compute: AIdx × V × V → V
  rcupo_get_loc_val: AIdx × V → Write
  rcupo_pass_on_val: AIdx × V → V
```

The *flow problem* of A is the following: (i) When I invokes A(i) and A(i) does not invoke any other A(j) but returns its result to I, then the *flow* is without problems. (ii) If A(i), above, invokes A(k), and A(k) returns its result to A(i), etc., then the *flow* is without problems. (iii) If A(i), above, passes flow onto A(k), that is, when A(k) is not expected to returns its result to A(i), etc., then there might be a *flow problem*. In any case, if A(i), above, passes flow onto A(k), then A(i) is expected to terminate. (iv) A(k), above may return its final result to I, in which case there is no *flow problem*. (v) A(k), above, may invoke some A(j), which then returns its result to A(k), which proceeds as in case (iv), in which case there is no *flow problem*. (vi) Or A(k), above, may pass flow onto some A(j), and we have a *flow problem* similar to that of case (iii).

Determination of *flow problems* cannot be done statically: cannot be done solely by a simple, linear static inspection of the specification texts of the A(i)s. An iterative (recursive) *flow analysis* — reminiscent of "program execution" — has to be performed. We shall not cover, here, this important aspect of flow analysis — other than referring to the seminal work of Patrick Cousot et al. [63–71].

Figure 28.14 illustrates: I invokes A(2); A(2) first invokes A(i), with A(i) invoking A(n); A(n) returns to A(i), which returns to A(2); then A(2) invokes A(1), and A(1) returns to A(2); finally A(2) returns the result to I.

Figure 28.15 illustrates: I invokes A(2); A(2) invokes A(i), which returns to A(2); A(2) passes on to A(n); A(n) invokes A(i), which returns to A(n); finally A(n) returns the result to I. ∎

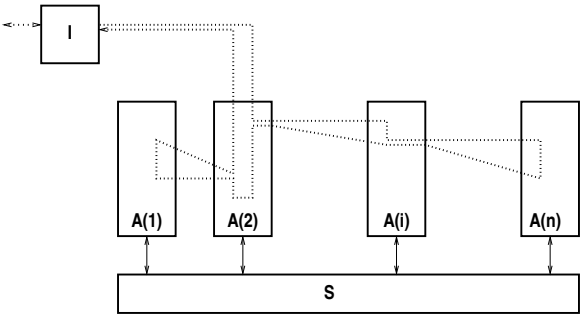


Fig. 28.14. Simple invocation of services

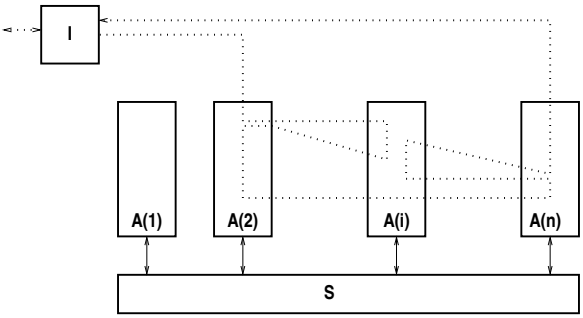


Fig. 28.15. Invocation and passing-on of services

28.4.6 Discussion

The client/server problem frame is strongly related to the connection problem frame (Sect. 28.7). For that reason we refer here to the concept of coordination languages. Programming in these one can express client/server protocols. Coordination languages have evolved since the mid-1990s. Several symposia have now been held. Proceedings have been published by Springer in their Lecture Notes in Computer Science series (Vols. 1061, 1282, 1594, 1906, 2315, 2949 and 3454). We recommend papers by Jean-Marie Jacquet et al. [48, 49, 194].

28.5 Workpiece Architectures

Examples of problems which belong to the workpiece frame are: (i) administrative forms (budgets and accounts, requisitions, invoices, applications, etc.),

their creation and handling; (ii) CAE/CAD (drawings and drawing, designs and design); (iii) graphical user interfaces (GUI); and (iv) software development (*"documentation is everything!"*).

28.5.1 Workpiece Domain

Workpiece domain: We have a workpiece frame when the universe of discourse can be characterised in terms of: *users*, *templates*, *designs* and *aggregations*. Users manipulate tangible, manifest documents such as templates, designs and aggregates. Templates are one kind of document — think of them as forms with preformatted rules and regulations that are ready to be "filled in". Designs are partially or fully "completed" templates. Aggregates are based on templates and are composed from zero, one or more designs. ∎

Figure 28.16 shows one template, top left, and one design based on that template, top right. It then shows n designs, of which three are instantiated, centre. Then, at bottom, the figure shows a simple aggregation from just these n designs.

We refer to templates, designs and aggregates as units. Our descriptions above are deliberately sketchy. This allows us to consider a large class of applications as belonging to the workpiece frame. Figure 28.17 shows a schematic summary of this rough-sketch explanation.
Schematically:

type
  Te, De, Ag, Inf
value
  gTe: Inf $\xrightarrow{\sim}$ Te
  bTe: Inf $\xrightarrow{\sim}$ Te-**set**
  uTe: Te × Inf $\xrightarrow{\sim}$ Te
  gDe: Te × Inf $\xrightarrow{\sim}$ De
  bDe: Te × Inf $\xrightarrow{\sim}$ De-**set**
  uDe: De × Inf $\xrightarrow{\sim}$ De
  gAg: Te × Inf × De* $\xrightarrow{\sim}$ Ag
  bAg: Te × Inf $\xrightarrow{\sim}$ Ag-**set**
  uAg: Ag × Inf × De* $\xrightarrow{\sim}$ Ag

Legend: Te: template, De: design, Ag: aggregate, Inf: information, g: generate, b: browse, u: update.

Explanation: gTe: generate template based on (some) information. bTe: browse for (suitable) template with (some) information. uTe: Update (given) template with (some) information. gDe: generate design given template and (some) information. uDe: update given design with (some) information. gAg: generate aggregate given (its) template and list of (input) designs. uAg: update (given) aggregate with information and list of (input) designs. And so on.

28.5.2 Workpiece Requirements

Workpiece requirements: Workpiece requirements typically state: Implement the workpiece "system" on the computer. That is, instead of manifest

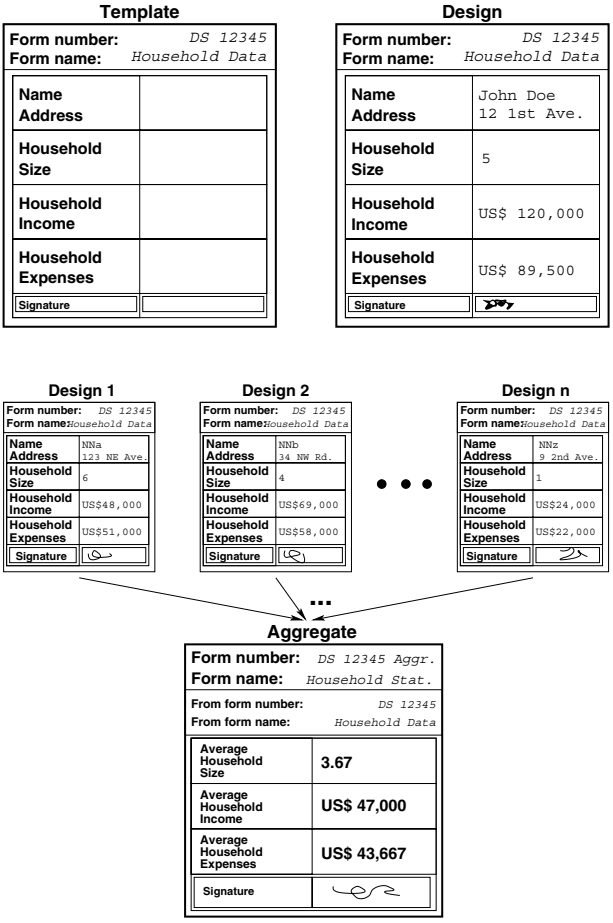


Fig. 28.16. One template and design + n aggregated designs

(tangible) units and the physical communication of these among users, support templates, designs and aggregations — their "storage" and manipulation (creation, update, etc.) — by computing (Fig. 28.18). ∎

Workpiece requirements are usually more detailed in their description of templates, designs and aggregations as well as the operations on these. Typically the units all have their own, unique identifications and separate or shared storage. Sometimes time-stamped versions of units are kept for reference, etc.

type
T, Ti, Di, Ai, Inf
Temp = Ti × Te
Dsgn = Di × Ti × De

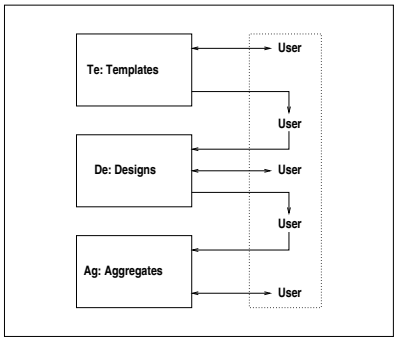


Fig. 28.17. Workpiece domain

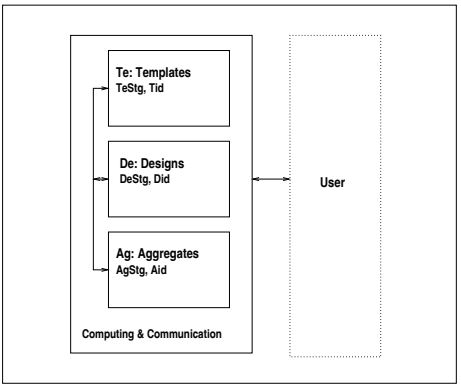


Fig. 28.18. Workpiece requirements

Aggr = Ai × Ti × Ag
Σ = TeStg × DeStg × AgStg
TeStg = Ti $\overrightarrow{m}$ (T $\overrightarrow{m}$ Te)
DeStg = Ti $\overrightarrow{m}$ Di $\overrightarrow{m}$ (T $\overrightarrow{m}$ De)
AgStg = Ti $\overrightarrow{m}$ Ai $\overrightarrow{m}$ (T $\overrightarrow{m}$ Ag)

value

GTe: Inf → T $\xrightarrow{\sim}$ Σ $\xrightarrow{\sim}$ Σ × Ti
BTe: Inf → Σ → Ti-**set**
UTe: Ti × Inf → T $\xrightarrow{\sim}$ Σ $\xrightarrow{\sim}$ Σ
GDe: Ti × Inf → T $\xrightarrow{\sim}$ Σ $\xrightarrow{\sim}$ Σ × Di
BDe: Ti × Inf → Σ $\xrightarrow{\sim}$ Di-**set**
UDe: Di × Inf → T $\xrightarrow{\sim}$ Σ $\xrightarrow{\sim}$ Σ
GAg: Ti × Inf × Di* → T $\xrightarrow{\sim}$ Σ $\xrightarrow{\sim}$ Σ × Ai
BAg: Ti × Inf → Σ $\xrightarrow{\sim}$ Ai-**set**
UAg: Ai × Di* × Inf → T $\xrightarrow{\sim}$ Σ $\xrightarrow{\sim}$ Σ

Legend: T stands for (the domain of) time. Ti, Di and Ai are template, design and aggregate identifiers. Temp, Dsgn and Aggr are "more" concrete forms of Te, De and Ag. Σ is the "grand" state. It consists of the template, the design and the aggregate storages. G, B, U stand for generate, browse and update.

Explanation: Designs and aggregates "carry" their template identifications. Storage of templates, designs and aggregates are time-stamped — thus "old" versions are (never) discarded. Generation updates storage and yields appropriate (template, design or aggregate) identifiers. Browsing may yield zero, one or more template, design or aggregate identifiers but does not change storage. Updates apply to the latest time-stamped and identified template, design or aggregate and change storage.

28.5.3 Workpiece Systems Design

The *template, design, aggregate* and *information* (i.e., the Te, De, Ag, Inf) entity types are, by definition of being workpieces, usually highly structured, i.e., composite types, and are different from application area to area. Typically they have very definite syntaxes and their *generate, browse,* and *update* functions (i.e., G.., B.., U..) are accordingly syntax-directed. This means that all the semiotics principles and techniques of Vol. 2, Chaps. 6–9 apply. The handling of their storage is classically an information repository, i.e., a database problem.

The *generate, browse* and *update* functions, usually, present challenges in addition to those of a syntactic and semantics nature: Typically issues of, for example, constraint satisfaction problems may be implied. We refer to the delightful book *Constraint Programming* by Krzysztof Apt, [15], for a seminal introduction to the subject.

28.6 Reactive System Architectures

What Michael Jackson calls control frame we call reactive systems frame. Examples of problems which belong to the reactive systems frame are: automated process monitoring and control, computer-assisted automobile monitoring and control, technologically monitored and humanly "controlled" air traffic, and train dispatch.

28.6.1 Reactive Systems Domain

Basis for a reactive system domain: We have the basis for a reactive systems frame if entities of the domain can be expressed in terms of time, T, a set of states, S, a set of external inputs, J, a set O of external outputs, the

dynamics of the input and state as relations Γ and Σ over time, and usually nondeterministic next-state and output function, F.

When F is (implicitly described, i.e.) approximated by a set of differential equations, where time is a crucial factor, then we shall here model it as G. ∎

Mathematical Characterisation: Reactive Systems Domain, I

type
T, S, J, O
Σ = T $\xrightarrow{\sim}$ S
Γ = T $\xrightarrow{\sim}$ J
value
F: J×S $\xrightarrow{\sim}$ (S×0)-**infset**
G: (T $\xrightarrow{\sim}$ (J×S)) $\xrightarrow{\sim}$ (T $\xrightarrow{\sim}$ (S×O)-**infset**)

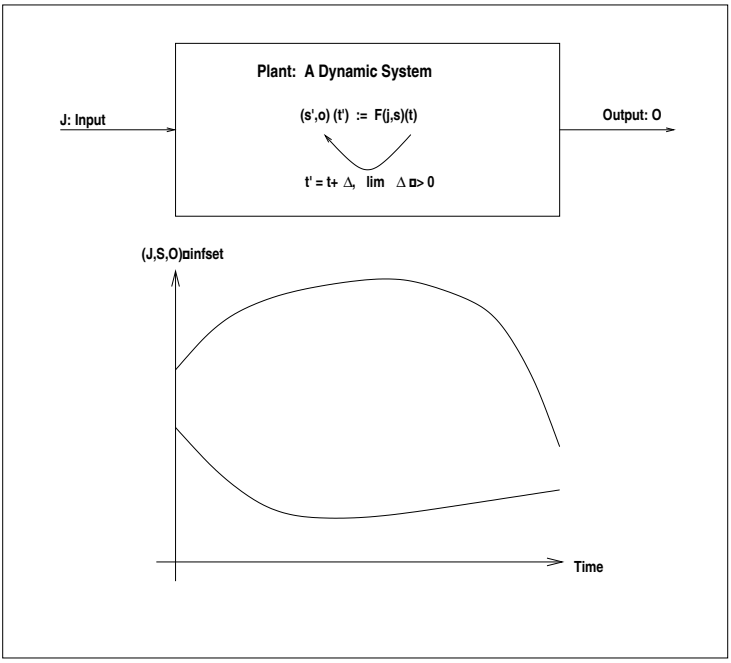


Fig. 28.19. Reactive systems domain

In Fig. 28.19 the uncontrolled operating regime, i.e., the set of all possible, including erroneous, (i.e., undesirable) behaviours,[8] is "symbolised" by the

[8] We call $T \to J \times S \times O$ a space of behaviours. The sequence:

space between the two curves. The X axis depicts time, and the Y axis a mapping onto one dimension of the three-dimensional point set space $(J \times S \times O)$-**set**.

Further towards a control domain: Observer predicates express such control theoretic concepts as coherency, stability, robustness and controllability properties of the domain. ∎

These notions are classical concepts of control theory. Some books are: [18, 29, 89, 109, 225]. A seminal article is: [373]. Various articles are: [53, 54, 56, 57, 106, 163, 166, 167, 217, 286, 326, 376].

Mathematical Characterisation: Reactive Systems Domain, II

```
type
   Sys_base = T → J×S×O
   Sys = {| sb:Sys_base • coherent(sb) |}
value
   coherent: Sys_base → Bool
   stable: Sys → Bool
   robust: Sys → Bool
   observable: Sys → Bool
   controllable: Sys → Bool
```

28.6.2 Reactive Systems Control Requirements

Reactive systems control requirements: We finally have a reactive systems control frame if, in addition to its basis, we have value constraints for the observable variables, i.e., if only some behaviours among all possible are permitted. ∎

Mathematical Characterisation: Reactive Systems Control Requirements

```
value  P: (T → (J×S×O)-infset) → Bool
```

The idea is that P is expressed in terms of, amongst others, the coherency, stability, robustness and controllability properties of the Domain. Additionally, optimality criteria form part of P. In Fig. 28.20 the constraints are symbolically shown as the "narrower" space of shaded behaviours. Major require-

$$< (t, j, s, o), (t', j', s', o'), ..., (t'', j'', s'', o''), ... >$$

such that the sequence of time stamps $t, t', ..., t''$ is dense and ascending resembles a behaviour.

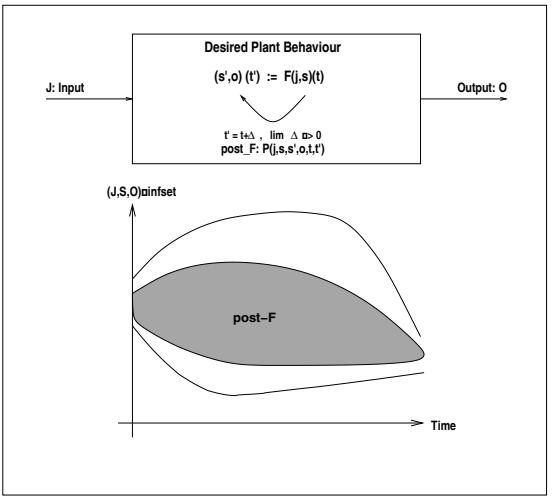


Fig. 28.20. Reactive systems monitoring and control requirements

ments definition concerns are: determination of the image of the state to be projected (into the computer), regularity of update and initial input.

28.6.3 Reactive Systems Control Design

Control engineering is about controlling some of the observable variables based on past values of all (required) observable values. The iteration over time, together with the nondeterministic nature of the input j:J, implies a feedback loop. Some observable variables are sampled (sensed at regular (incl. computer) time periods).

Control design: We have solved the control requirements if we have: a sampling (sensor) function π which observes the state, a control function, C, and a feedback activator function ϕ, such that s′ in $C(\pi(s)_{(t)}) = (c1, \ldots, cm)_{(t+\delta)}$, and $F(j \oplus \phi(c1, \ldots, cm)_{(t+\delta)}, s'')(t+\delta') = s'_{(t+\Delta)}$, lie within the control regime (s″ is the state at time $t + \delta'$). ∎

We refer to the classical feedback control set up for the monitoring and control of reactive systems in Fig. 28.21.

Mathematical Characterisation: Control Design

type c1,...,cm:V
value
π: $\Sigma \to$ T $\to$ (V $\times$ T)
C: (V $\times$ T) $\to$ (V $\times$ T)
ϕ: (V $\times$ T) $\to$ J $\times$ T

$\oplus$: J $\times$ J $\to$ J
F′: J $\to$ S $\xrightarrow{\sim}$ T $\xrightarrow{\sim}$ (S $\times$ O)

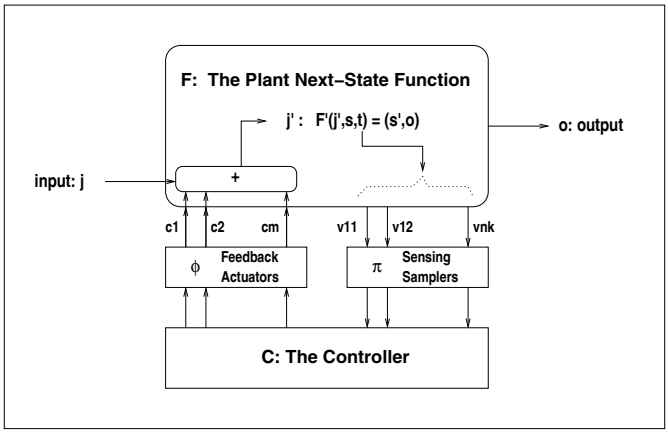


Fig. 28.21. A reactive systems control design

28.6.4 Discussion of Reactive Systems Design

We shall not, other than the above characterisations, cover the development principles, techniques and tools for reactive systems. The field of software development methods for reactive systems is currently, perhaps, the field where the use of formal techniques is often mandated by customers. Formal verification, including model checking, is here at the forefront. Some such principles, techniques and tools for reactive systems (domain, requirements and design) modelling are covered in some detail in Vol. 2, Chaps. 12–15. Example reports and papers which illustrate specific applications are found in [289,290,335,336,339,383,384]. Seminal textbooks on embedded, real-time systems development are [58,216,228–230].

28.7 Connection Frame

Examples of problems which belong to the connection frame are check and credit card slip transaction clearing, and programmable (cabled) sensor and actuator connections between humans or processes and computers. Data communication with its protocols belongs to the communications frame.

The connection problem frame is related to the client-server problem frame (Sect. 28.4).

28.7.1 Connection Domain

Connection domain: When one amongst one or more (so-called) senders wishes to send a message (a signal, a package, whatever) to one amongst one or more (so-called) receivers, then we have the basis for a connection frame (see Fig. 28.22). ▪

We assume all senders and all receivers to be uniquely identified. Senders may "store" messages sent and mark these acknowledged when such has been obtained. Receivers may "store" messages received and mark these acknowledged when such has been done.[9]

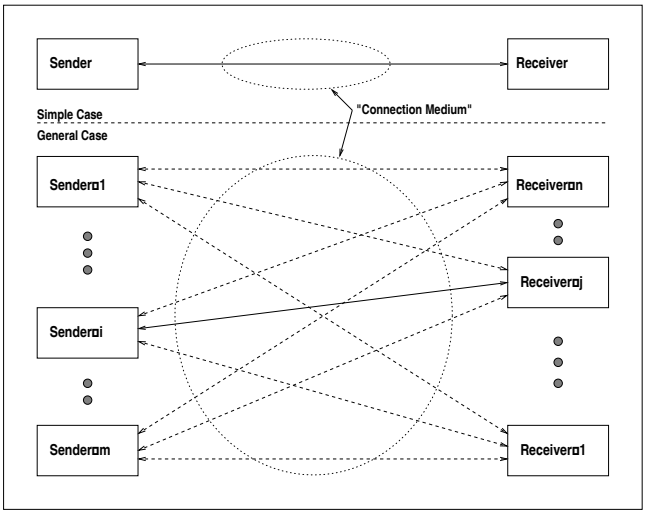


Fig. 28.22. Connection domain

Types, Signatures and Protocol: Connection Domain

type
T, Si, Ri, M
Sn = Ri $\overrightarrow{m}$ (T×M)-**set** ...
Rc = Si $\overrightarrow{m}$ (T×M)-**set** ...
Σ = (Si $\overrightarrow{m}$ Sn) × (Ri $\overrightarrow{m}$ Rc) ...
value
SndAck: Si × M × Ri $\tilde{\to}$ T $\tilde{\to}$ Σ $\tilde{\to}$ Σ × T
snd: M $\tilde{\to}$ Sn $\tilde{\to}$ T $\tilde{\to}$ (Sn × T)
rcv: M $\tilde{\to}$ Rc $\tilde{\to}$ T $\tilde{\to}$ (Rc × T)

[9] In a "real world" the same physical entity may take on both sender and receiver roles, and there is nothing in our description that prevents this.

```
ack: M ~→ Sn ~→ T ~→ (Sn × T)
max: T* → T

SndAck(s,m,r)(t)(ss,rs) ≡
      let (sn,rc) = (ss(s),rs(r)) in
   1. let (sn′,t′) = snd(m)(sn)(t) in
      let t″:T • t″ ≥ max{t,t′} in
   2. let (rc′,t‴) = rcv(m)(rc)(t″) in
   3. let (sn″,t⁗) = ack(m)(sn′)(t‴) in
      let t⁗⁗:T • t⁗⁗ ≥ max{t,t′,t″,t‴,t⁗} in
   4. (ss † [s↦sn″],rs † [r↦rc′],t⁗⁗)
   end end end end end end
```

Legend: T stands for time. Si, Ri: Sender and receiver identifiers. M: Messages and acknowledgements. The state Σ "summarises" all senders and receivers.

Explanation: At any time any sender has sent messages to and possibly received acknowledgements from identified receivers. Similarly for receivers.

(1.) Sender s processes the message to be sent at time t, thereby changing its (own) state at that time.

(2.) Receiver r receives the message at time t''', thereby changing its state at that time.

(3.) Sender s receives an acknowledgement from receiver r at time t'''', thereby changing the state at that time.

(4.) The sum-total result is a new system state at time t''''.

Connection via the medium takes time, as illustrated above.

28.7.2 Connection Requirements

Connection requirements: When one amongst one or more (so-called) senders wishes to send a message (a signal, a package, whatever) to one amongst one or more (so-called) receivers *via a shared connector* and within a certain time period then we have a connection frame. ∎

There may be one or more potentially shareable connectors. For each type of message and pair of senders and receivers there may be exactly one connector. Issues like message transmission failures additionally render the problem a data communications [hence multi] frame problem.

Formal Characterisation: Connection Requirements

```
type
   MTyp, Ci, Ξ, R
   Ψ = ClrIns × Conns × Σ
   ClrIns = (Si × MTyp × Ri) ~m→ Ci
```

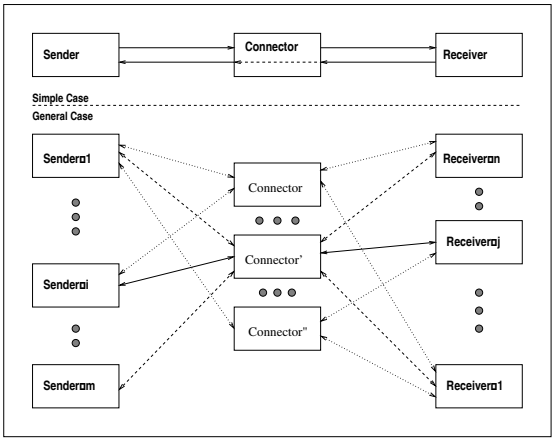


Fig. 28.23. Connection requirements

```
  Conns = Ci ~m~> Ξ
value
  SndAck: Si × (MTyp × M) × Ri × T ~> Ψ ~> Ψ
  sc: Si × M × Ri × T ~> Sn ~> (R × T × Sn)
  cr: R × Si × M × T ~> Ξ ~> R × T × Ξ
  ak: R × M × Si ~> Rc ~> T ~> R × T
  rc: R × T ~> Rc ~> R × T × Rc
  cs: R × T ~> Ξ ~> R × T × Ξ
  cl: R × T ~> Sn ~> Sn
  TimeLimit: T×T×T×T×T×T → Bool

  SndAck(s,(mt,m),r,t)(ψ) ≡
    let (cins,cnns,σ) = ψ in
    let c=cins(si,mt,ri), (ss,rs)=σ in
    let (p,t′,σs)=sc(s,m,r,t)(ss(s)) in
    let (p′,t″,ξ)=cr(p,s,m,t′)(cnns(c)) in
    let (p″,t‴,σr)=rc(ak(p′,m,s),t″)(rs(r)) in
    let (p‴,t⁗,ξ′) = cs(p″,t‴,s)(ξ) in
    let σ′s = cls(p‴,t⁗)(σs) in
    (cins,cnns † [c ↦ ξ′],
     (ss † [s ↦ σ′s],rs † [r ↦ σr]))
    end end end end end end end
  post TimeLimit(t,t′,t″,t‴,t⁗,t⁗′)
```

Legend: MTyp designates message types. Ci designates connector (clearer) identifiers. Ξ designates connector states. p:R designates forwarding and reply reports.

Explanation: The requirements are thought of as stipulating strict TimeLimits on the "processing" of message forwarding and acknowledgement returns.

28.7.3 Connection Systems Design

We shall not, other than the above characterisations, cover the development principles, techniques and tools for connection frame problems. Vol. 1, Chap. 21, Concurrent Specification Programming, and Vol. 2, Chap. 13, Message and Live Sequence Charts, presented principles, techniques and tools for specifying connection frame domains, requirements and design.

The work by the Carnegie Mellon University group around David Garlan (G.D. Abowd, R. Allen, M. Shaw, and C. Shekaran, and others) has contributed rather significantly to the clarification of many software architecture issues, notably those that relate to components and their connections: [1, 2, 7–9, 113–116, 333]. The concept of coordination languages for expressing connections has evolved since the mid-1990s. Several symposia have now been held. Proceedings have been published by Springer in their Lecture Notes in Computer Science series (Vols. 1061, 1282, 1594, 1906, 2315, 2949 and 3454). We recommend papers by Jean-Marie Jacquet et al. [48, 49, 194]. Finally we recommend *Principles of Protocol Design* by Robin Sharp [332] as a good textbook related to connection frames.

28.8 Discussion

28.8.1 General

We have treated, in some isolation from one another, a number of decreasingly distinct problem frames. It is, however, rarely the case that any one problem belongs to exactly one of these frames. Most compiler developments contain strong elements of the translator and of the workpiece frames. Most embedded, real-time systems developments contain strong elements of the reactive and of the connection frames. Most information systems developments contain strong elements of the repository and the workpiece frames. And so on.

28.8.2 Principles, Techniques and Tools

We summarise:

Principles. A main principle of *domain-specific architectures* is that of being prepared for there being a domain-specific architecture (i.e., a design) that most aptly fits the requirements hence that of analysing requirements into a suitable such architecture, if applicable, and otherwise putting a new one together — most likely by an appropriate composition of base domain-specific architectures such as those hinted at in this chapter. ▪

Techniques. *Domain-Specific Architecture:* This chapter has hinted at a number of techniques. For each frame we hinted at a separate set of techniques. This, of course, is reflected in our inability to be fully specific, and hence in our pointing the reader to specialist textbooks and monographs covering these special frames. We refer to Michael Jackson's book on *Problem Frames — Analysing and Structuring Software Development Problems* [191], for a more substantial treatment with many more detailed principles and techniques. ▪

28.9 Exercises

28.9.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

28.9.2 The Exercises

Please read subsections titled "On Event Notification Management" (I) and (II) below (each between pairs of three bullets: • • •) before trying to tackle Exercises 28.7–28.9.

Exercise 28.1 *Your Selected Domain's Problem Frame.* For the fixed topic, selected by you, analyse your requirements and suggest which problem frames appear applicable to the various parts of your requirements.

Exercise 28.2 *Translation Frames.* Which of the 15 running domain (requirements and software design) examples outlined in Sect. 1.7.1 may give rise to a translation frame?

Exercise 28.3 *Information Repository Frames.* Which of the 15 running domain (requirements and software design) examples outlined in Sect. 1.7.1 may give rise to an information repository frame?

Exercise 28.4 *Client/Server Frames.* Which of the 15 running domain (requirements and software design) examples outlined in Sect. 1.7.1 may give rise to a client/server frame?

Exercise 28.5 *Workpiece Frames.* Which of the 15 running domain (requirements and software design) examples outlined in Sect. 1.7.1 may give rise to a workpiece frame?

Exercise 28.6 *Reactive Systems Frames.* Which of the 15 running domain (requirements and software design) examples outlined in Sect. 1.7.1 may give rise to a reactive systems frame?

• • •

On Event Notification Management, I

The next three exercises relate to Sect. 28.4.5.

We refer specifically to the text preceding Example 28.6. We hinted at a notion of *event notification management.* We shall now elaborate further on this idea. We assume that some user, u, is invoking some service A_i (through I). Four kinds of event notification are then relevant:

1_n: immediate notification of the user who requested the A_i (through I)
2_n: immediate notification of some other user (or set of users), u'
3_n: immediate notification of some other services, A_k, of user u's use of A_i
4_n: time-delayed notification of some other services, A_k, of user u's use of service A_i

After the next three exercises we bring in some more explanatory material related to those exercises.

• • •

Exercise 28.7 *Immediate Event Management.* By direct A_i event notification management we shall understand that the software of A_i, wherever appropriate, specifies detection of the need for event notification — as per the outlines explained in the paragraphs "On Event Notification Management, I–II". You are to suggest how user requests for A_i service (i.e., directly from I) are to handle this form of event notification management. Depending on the outcome of A_i's analysis of the service request, A_i may decide to perform or not perform the service request.

Hint: Start by deciding on information that needs to be part of initial user requests in order to effect the four possible event notifications.

Exercise 28.8 *Indirect, Synchronous Event Management.* We now seek a separate event manager service E_{synch} such that A_i requests this service to analyse whether a notification is required and then handles it. E_{synch} is expected to perform the analysis and handling synchronously (i.e., "immediately" upon request). Depending on the outcome of E_{synch}'s analysis of the service request E_{synch} may decide to let A_i perform or not perform the service request.

You are to suggest how user invocations are to handle this form of event notification management via E_{synch}.

Hint: Start by deciding on information that needs to be part of initial user requests in order to effect the four possible event notifications — and thus on the messages communicated between I and E_{synch}, and between E_{synch} and A_i.

Exercise 28.9 *Indirect, Asynchronous Event Management.* We now seek another form of separate event manager service E_{asynch}. It is invoked by I, at the same time that I invokes A_i. As for E_{synch}, we expect E_{asynch} itself to discover the need for event notification. E_{asynch} may handle this event notification service asynchronously, i.e., "out of step" with A_i's handling of the original service request. Thus, irrespective of the outcome of E_{asynch}'s analysis of the service request, A_i shall also perform the service request.

You are to suggest how user invocations of I are to handle this form of event notification management via E_{asynch}.

Hint: Start by deciding on information that needs to be part of initial user requests in order to effect the four possible event notifications — and thus on the messages communicated between I and E_{asynch}.

• • •

On Event Notification Management, II

The three event notification management "systems" can, perhaps, be better understood by also providing the following conceptual diagrams (see Fig. 28.24).

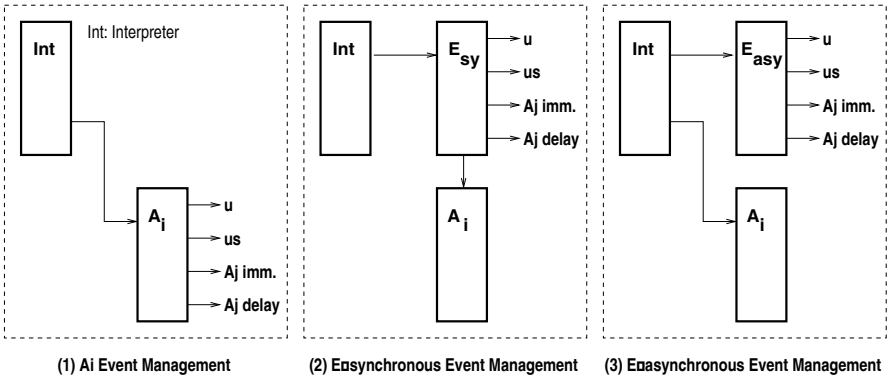


Fig. 28.24. Three event notification management systems

In (1) A_i analyses original user requests and decides as to whether one or another or none of the four possible event notifications should take place. If such an event notification shall take place, then it shall take place immediately.

In (2) E_{synch} analyses original user requests and decides as to whether one or another or none of the four possible event notifications should take place. If such an event notification shall take place, then it shall take place immediately, and before possible invocation of A_i.

In (3) E_{asynch} analyses original user requests and decides as to whether one or another or none of the four possible event notifications should take place. If such an event notification shall take place, then it can take place at any time, irrespective of invocation of A_i.

29

Etcetera: Coding and All That!

- The **prerequisite** for studying this chapter is that you have ended your study of this part and that you are wondering: *How, now, do I do the final steps, from formal software design specifications to practical programming?*
- The **aims** are to indicate translations from formal specifications to current programming, to, more generally, put formal specifications in perspective, and to link, as it were, the world of formal techniques to the world of informal programming.
- The **objective** is secure that you, as a professional software engineer, indeed will know how to span the spectrum, from formal specifications to practical programming.
- The **treatment** is discursive.

A Disclaimer

This chapter is cursory. This volume is not about programming, and certainly not about programming "in the small", designing algorithms and data structures for efficient computations. It is assumed that the reader will have learned such programming prior to studying this volume. The purpose of this chapter is to link the principles, techniques and tools of formal specification to those of current programming languages.

29.1 From Formal Specification to Programming

Domain descriptions and requirements prescriptions, yes, even software architecture design specifications are — usually — not programs in the sense of their texts immediately (say, through interpretation or compilation) serving as a basis for computations. That is, we usually require a stage of software design in which the, as in the case, mostly, of these volumes, `RSL` specifications are further "converted" into code, i.e., programs, in some programming language. This chapter considers some issues in that conversion process.

29.1.1 From Specifications to Programs

One issue is that of abstraction. Usually the abstractions are not immediately and, in most cases, not efficiently representable. That is, it is not immediately obvious how certain abstractions could be coded in a chosen programming language. And it is likewise not immediately obvious which such representation (i.e., coding) is desirably efficient.

The "immediately obvious" hedge used above is intended to indicate that one cannot generally expect to automate the transformation from abstractions to concretisations. Much can be done in this direction, and much has been and is being done in this exciting area of compilation cum optimisation [12, 308–310, 314, 315].

29.1.2 From Abstract Types to Data Structures

When focusing only on transformations from more abstract to more concrete types (where by the latter we mean concrete, i.e., defined types), these three volumes have shown many examples. The choice of concrete types for sorts (i.e., "really abstract" types in RSL) is determined by the total set of operations that are defined over the sorts.

We refer, for example, to Vol. 1, Chap. 16, Sects. 16.4.5–6. There we show a sequence of what you may think of as transformation steps from abstract models of graphs to rather concrete, pointer-based data structures. This is an example of concretisation "in the small". With data type transformations (reifications) follow corresponding, reasonably obvious, "translations" of those expressions which involve the concretised data types.

And we refer, as another example, to the present volume's Chap. 28. There we show a sequence of what you may think of as transformation steps from abstract models of "entire" file systems to pointer-based data structures. This is an example of concretisation "in the large". The "final step" from RSL's type definitions to the type definitions of the chosen programming language is obvious. A general approach to implementation of sorts and abstract types is given in [118]. Other than these few references we shall not cover the transformation of types into data structure definitions.

29.1.3 From Applicative to Imperative Programs

We refer to Vol. 1, Chap. 20, Sect. 20.5: Translations: Applicative to Imperative, and in particular Sect. 20.5.3: Applicative to Imperative Schemata. The essence of the above reference is that prescriptions are given for the translation of a number of applicative constructs into imperative ones. Those not explicitly covered are usually either "ordinary" operator/operand expressions which need not be translated, or of such a simple kind that the reader is expected to "invent" the translation. The seminal RSL reference [118] gives further hints as to translations from the applicative to the imperative.

29.1.4 Translations into Concurrent Programs

We refer to Vol. 1, Chap. 21, Sect. 21.5: Translation Schemas. The essence of the above reference is that prescriptions are given for the translation of a number of applicative and imperative constructs into concurrent ones.

29.1.5 From RSL to SML, Java, C# and Other Languages

Above we pointed to earlier sections of these volumes in which we have shown how to translate one kind of RSL construct into another, in the directions from applicative to imperative, and from applicative or imperative to concurrent RSL/CSP constructs. Those translations were thought of as being done "manually", by the specifier cum programmer.

Some of these translations, including those of the transformation or reification of abstract data structures into concrete ones, can, however, be mechanised. Several model-oriented specification languages (such as B [4], RSL, VDM-SL [104] and Z [377]) can provide tools that help translate concrete applicative (and imperative) constructs into the imperative constructs of such programming languages as C, C++, C# [161], Java [17] and Standard ML (SML [142]).

Data structure concretisations to convert abstract, formal specification language constructs into those of, for example, C, C++, Java [11, 328], C# or SML [247] are also of research interest. "The last word has yet to be uttered."

29.2 The Beauty of Programming

Art, Discipline, Craft, Science, Logic and Practice

Edsger W. Dijkstra is credited with the utterance: The beauty of programming [103]. Programming is also an art [204–206], a discipline [86], a craft [296], a science [130], a logic [158], and a practice [159].

The references just given cover a number of exciting, deep approaches to the discovery and justification of algorithms and data structures. This discovery and justification is, in these volumes, seen as an indispensable activity adjoint to software engineering. What we have covered in these volumes is not a replacement for discovery and justification. Usually the discovery and justification is the work of one person, one mind. And software engineering is about principles, techniques and tools to be adhered to, followed and used by groups of engineers. Software development management must secure the smooth, harmonious interplay between "programming in the small", algorithm discovery, and software engineering "in the large".

29.3 Programming Practices

Over the years many programming principles and techniques have been proposed. For a time they attracted the attention of many programmers. For years to come new such "schools" will rise to fame, briefly, for longer periods or more permanently. The art [204–206], discipline [86], craft [296], science [130], logic [158], and practice [159] references, as well as others, notably refinement calculus [249], and calculus of reactive systems [19], can be expected to last for quite some time.

29.3.1 Structured Programming

Michael A. Jackson's Structured Programming (JSP) [187] is well worth getting acquainted with. In short, it addresses the development of software for those problems whose information (cum data) structures can be expressed simply in terms of certain kinds of (sequential data structure) diagrams: JSP diagrams.

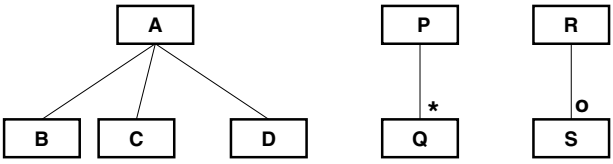


Fig. 29.1. JSP diagrams

Figure 29.1 is "equivalent" in meaning to:

type
A = B × C × ... × D
P = Q*
R == nil | mkS(s:S)

where B, C, ..., D, Q and S are some further specified data types. We invite the reader to study [187].

29.3.2 Extreme Programming

The basic idea of extreme programming (XP) is this: If some programming practice appears, or is believed to be beneficial, and since two programmers working closely together seem to be of that (beneficial) nature, then let all programming occur by two programmers working together all the time, at one terminal, big screen, etc., etc.! We refer to works by Kent Beck [22, 23].

If you think or believe that this form of programming, XP, is good, then practice it. There is nothing in the principles and techniques of these three volumes that prevents them from being used in the *extreme!*

29.3.3 Object-Oriented cum UML Programming

Chapter 10 in Vol. 2 introduced the concept of modularisation, and hence of objects. It also covered, to some extent, the so-called Unified Modeling Language (UML) concept of class diagrams [44,193,264,303].

We shall refrain from risking a definition of what object-oriented (OO) programming is — since various "schools" who proclaim to be OO differ in their definitions. When, in these volumes, we "encapsulate" a data structure as basically constituting all of a state in a process, then we can be said to be OO. Process inputs/outputs are then the messages that invoke OO methods, that is, which invoke functions and receive return values. And then we "mimic" UML.

When, in these volumes, we "encapsulate" a data structure and its operations in a class (or scheme or object), then we can be said to be OO. And then we still mimic UML. When, in these volumes, we combine RSL with Petri nets [196, 273, 293–295] (as in Vol. 2, Chap. 12), then we also mimic UML. When, in these volumes, we combine RSL with message sequence charts [182–184] (as in Vol. 2, Chap. 13), then we also mimic UML. When, in these volumes, we combine RSL with statecharts [73, 149, 203] (as in Vol. 2, Chap. 14), then we finally mimic UML. So, the principles, techniques and tools offered in Vols. 1–3 offer a formal alternative to, but are also complementary with, UML.

29.3.4 Chief Programmer Programming

Characterisation. By *chief programmer programming* we shall understand a process in which, in essence, one person basically decides "everything". This person decides how the domain is to be modelled, decides how the requirements are to be modelled and decides the software architecture and major components design. ▪

The chief programmer is thus like the lead architect on the design of a building. That is: By chief programmers we expect to have such persons as the likes of Andrea Palladio (the Rotunda Villa near Vicenza, Northern Italy), Antonio Gaudí (Sagrada Familia, Barcelona, Spain), Frank Lloyd Wright (the Guggenheim Museum, New York City, NY, USA), Victor Horta (Horta Museum, Brussels, Belgium), Mies van der Rohe (Villa Tugendhat, Brno, The Czech Republic), Le Corbusier (Unité d'Habitation, Marseilles, France), Oscar Niemeyer (Brasilia, Brazil), Jørn Utzon (the Sydney Opera House, Australia), Richard Meier (the J. Paul Getty Museum, Los Angeles, USA), Paul Gehry (the Guggenheim Museum, Bilbao, Basque Region, Spain), etc.

Over the years, successful software development projects have been associated with chief programmers: Peter Naur (the Gier Algol 60 compiler, [256]) and Linus Torvalds (the Linux operating system). Modesty ought to prevent us from mentioning [41]!

We expect our chief programmers to otherwise follow the principles, apply the techniques and use the tools of this series of textbooks on software engineering. By the efforts of such chief programmers we hope to achieve elegant, beautiful, pleasing, adequate and correctly functioning software systems.

29.4 Confidence-Building Software Development

Characterisation. (I) By *verification* we shall loosely understand a process whereby a pair of formal specifications (including one formal specification and a concrete program) are being formally related to one another — with the aim of stating that one of the specifications is a correct implementation of the other. ▪

Characterisation. (I) By *model checking* we shall loosely understand a process whereby one specification, "the model", serves as a reference for the animation, i.e., for a kind of program execution, of the other (the actual) specification — the claim being made that the actual specifications behave as the model. ▪

Characterisation. (I) By the *testing* of a program we shall loosely mean a set of executions of that program for different, given input data (i.e., initial states) and such that the execution outcome, "the results", are being held up against, i.e., compared to, a correspondingly given set of expected outcomes — the aim being to state that the concrete specification is a correct implementation of the abstract (at least with respect to those inputs). ▪

The last of these three approaches to building confidence in the program being coded will be treated in some detail below (Sect. 29.5.3). The two others (verification and model checking) will only be briefly mentioned.

Explorative, Experimental and Incremental Software Development

The real aim of Sect. 29.4 is to advocate a special way of exploring domains, of experimenting with requirements, and of incrementally developing, verifying, model checking and testing the software. This is done hand in hand with developing domain descriptions, requirements prescriptions, software architecture design, component design and unit (i.e., module) coding.

29.4.1 When to Verify, Model Check and Test

In a number of steps we reason our way towards this special way of incrementally designed software systems.

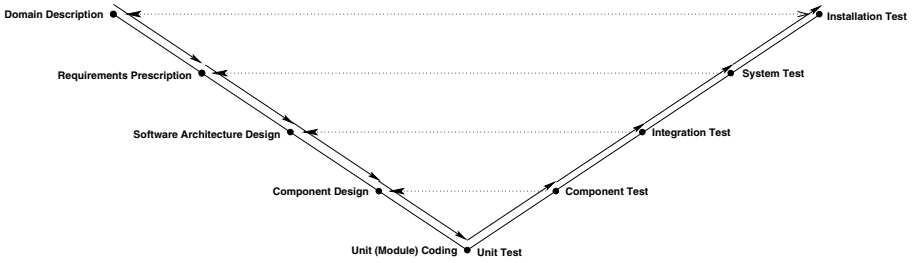


Fig. 29.2. The ∨ diagram

The ∨ Diagram

Figure 29.2 shows a so-called ∨ diagram — one that is used quite often in the literature. It is used in order to explain, to bring to the fore, the problem of introducing misunderstandings into the domain model, misunderstandings into the requirements prescription, and errors into the software design, at all levels!

Order of Tests

Let us focus just on testing. Conventional wisdom says that one must first do tests on modules before one can perform tests on components (aggregations of modules). And hence, so says conventional wisdom, one can only do integration tests once two or more components have been tested, and only perform system tests once all components have been tested. Once the product is in the field, it is being installation-tested, or field-tested, through the daily use of the ultimate customers.

Discovery of Errors, Misinterpretations and Misunderstandings

With this conventional wisdom we experience, therefore, that errors that crept into the software, during module design and coding, can usually be found during module test, i.e., immediately, or almost immediately. But one also finds that errors that crept into component design are first discovered — in the reverse order of design — after module tests, and so forth. The real misinterpretations of customer requirements and misunderstandings of the domain appear first to be tested, and are perhaps found, during, at the earliest, installation test, and, at its worst, a long time after the product has been in the field, i.e., in use.

Cost of Corrective Maintenance

The cost of corrective maintenance is small, comparatively speaking, for module design errors, and larger for component design errors due to components

being "larger": Being aggregations of modules. The costs of correcting architecture design errors, of requirements prescription misinterpretations and of domain description misunderstandings, grow proportionally to the time elapsed between when these misunderstandings, misinterpretations and errors were introduced and their possible discovery.

Impossibility of Certain Corrective Maintenance

Hence correcting a domain misunderstanding can be so costly as to simply void the product. It must be withdrawn since very many and fundamental development decisions may have to be "corrected", so many that it is plainly impossible. By 2005 it was judged that one of every three almost fully, or fully developed software systems had to be entirely scrapped for this reason.

• • •

We are now ready to argue for our suggestion for a special way of incrementally designing software systems.

29.4.2 Demo → Skeleton → Prototype → System

What can we conclude from this? We conclude that we must try reverse the order of tests, model checks and verification, ideally to follow the hints of the | diagram, Fig. 29.3 (pronounced i diagram). Thus we propose that design be done as: demo → skeleton → prototype → system.

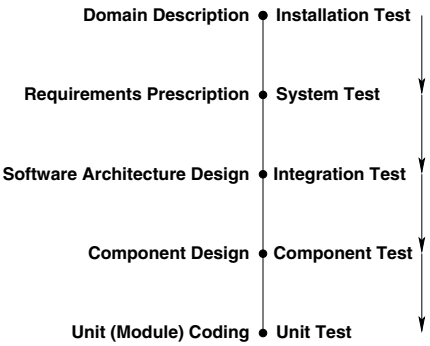


Fig. 29.3. The | diagram

In the "I" diagram we wish to express that one should do the tests, model checks and verifications, not in the reverse order of "what is being tested", etc., but in the same order as that "which is being tested", etc., is being first introduced. Hence we advise the following:

Installation verifications, model checks and tests should be done by asserting properties of the domain description and seeing to it (i.e., verifying, model checking, testing) that they hold of the domain description.

System verifications, model checks and tests should be done by asserting properties of the requirements prescription and seeing to it (i.e., verifying, model checking, testing) that they hold of the requirements prescription.

Integration verifications, model checks and tests should be done by asserting properties of the software architecture design and seeing to it (i.e., verifying, model checking, testing) that they hold of that design.

Component verifications, model checks and tests should be done by asserting properties of the component design and seeing to it (i.e., verifying, model checking, testing) that they hold of that design.

Unit (module) verifications, model checks and tests should be done by asserting properties of the unit (module) design and seeing to it (i.e., verifying, model checking, testing) that they hold of that design.

To achieve the above implies that what is being subjected to verification, model checking and testing, i.e., the description (the prescription or the specification) can indeed be so analysed. This means, for verification and model checking, that the specification be formal. For testing it means that the descriptions, the prescriptions, and the specifications can be executed. Now, normally domain descriptions, requirements prescriptions and abstract software design specifications cannot be readily, i.e., immediately, per se, subject to computer interpretation. They must first be concretised, but the concretisation need not be targeted at efficient interpretations. Figure 29.4, the *demo/skeleton/prototype development* (D/S/P) diagram, attempts to show what is at stake.

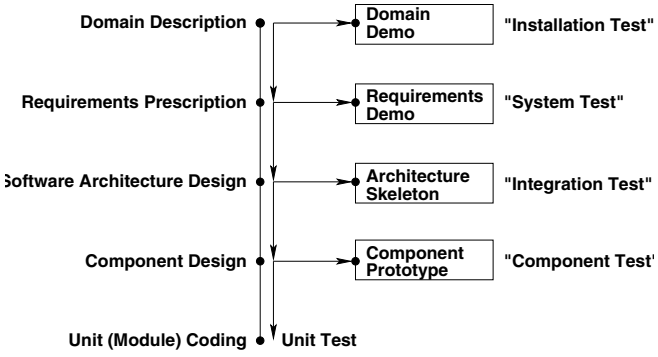


Fig. 29.4. The D/S/P diagram

Domain demo: From the domain description is constructed a demo which (i) visualises and animates the domain phenomena and concepts, and which (ii) allows the user to browse domain description documents.

Requirements demo: From the requirements prescription is constructed a demo which (i) visualises and animates the requirements phenomena and concepts, and which (ii) allows the user to browse requirements prescription documents.

Architecture skeleton: From the software architecture design specification is constructed a skeleton which (i) visualises and animates the set of components (i.e., the external components and their interfaces), and which allows (ii) the user to browse software architecture design specification documents.

Component prototype: From the component design specification is constructed a prototype which (i) visualises and animates the individual (i.e., internal) component concepts, and which (ii) allows the user to browse component design specification documents.

Some technical terms have been used.

Demo: By a demo we shall understand a software package that can visualise and animate important phenomena and concepts, and which allows the user to freely browse a set of documents for which the demo was built.

Visualise: By visualisation we understand the showing of still pictures, of diagrams, of photos, etc., of phenomena and concepts.

Animate: By animation we understand the temporal, i.e., the timewise, flow of states and events of, and interactions with, the system for which the demo skeleton or prototype was built.

Browse: By browsing we mean the free selection of document texts based on contents listings, highlighted (or not highlighted) texts and associations of such. Browsing requires sophisticated linking between and searching of texts.

Skeleton: By a skeleton we understand a demo software package that allows the user to see, control and monitor the interfaces between components, including simulating invocations of functions "across" interfaces.

Prototype: By a prototype we understand a skeleton software package that allows the user to see, control and monitor the intrafaces within components, including simulating results of actions internal to a component.

Please note that there is an implied sequence:

Domain Demo→Requirements Demo→Architecture Skeleton→Component Prototype

Figure 29.5 attempts to show the completion of this development into the final software system. The V:MC:T dashed wedges shall indicate that verification, model checking and testing is targeted at either or both of the specifications linked by the wedges.

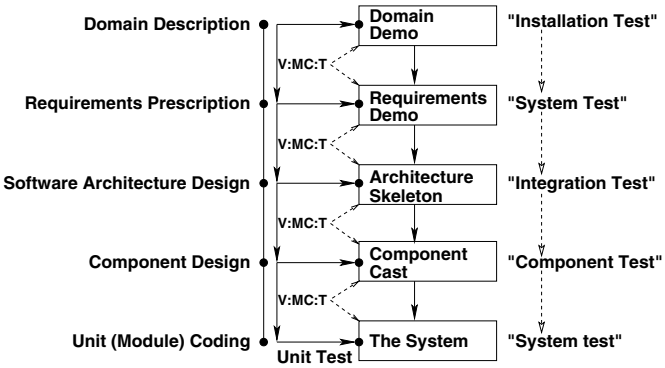


Fig. 29.5. The D diagram

The basic idea is to develop this sequence of software packages in such a way as to reuse, in a subsequent phase or stage, most of what has been designed and verified/model checked/tested in the previous phase or stage. These verifications, model checks and tests inherit those of previous stages, and with the subsequent phase or stage software demo, or skeleton and/or prototype design being refinements or extensions of previous phase or stage designs.

The ideal is that eventually it becomes just a matter of inserting all modules, one by one, in respective component slots, one by one, ending up, hocus pocus, with the system. Well, it is not always that simple, but quite a lot of the development can be done this way without sacrificing the efficiency of the final system.

29.5 Verification, Model Checking and Testing

The concept of validation has been treated extensively, in connection with domain validation and requirements validation: Sects. 14.3 and 22.3.

Validation Contra Verification

From Barry Boehm [42] we quote:

- **Validation:** Validation is concerned with getting the right product.
- **Verification:** Verification is concerned with getting the product right.

We examine in detail some issues of verification, model checking and — especially — testing. Validation is performed on domain descriptions and requirements prescription. Verification, model checking and testing is performed wrt. requirements prescriptions.

29.5.1 Verification

Characterisation. (II) By *verification* (of one or a pair of specifications) we understand a proof of a theorem, where the theorem states that a stated property (contained in the theorem) of the specification holds. ▪

The Bases for Verification

The bases for verification are (the domain description, the requirements prescription and) the software design documents. Verification effectively means to base proofs of otherwise stated lemmas on (one or more of) these documents. The proof is a purely syntactic process: From what is stated in the document(s) and from the lemma to be proved, a strategy is chosen. The chosen strategy will prove that the lemma follows from the document text(s) — proceeding by applying proof rules (i.e., syntactic quantities) to the syntax of the document text(s), eventually leading to a conclusion which states the lemma.

Characterisation. (III) *Verification* is thus a syntactic process. That is, the specification(s) and the theorem are treated syntactically, the proof consists of a sequence of steps where each step is seen as a syntactic quantity, and where the transitions from one proof step to the next proof step are justified by syntactically substituting an axiom or an inference rule into the text of one proof step to obtain the text of the subsequent proof step — finally to achieve the text of the theorem. ▪

We shall not in these volumes show any examples of verifications. This may come as a surprise to many readers. After all, is the whole purpose of formal specifications not that of being able to prove? Yes and no! Yes to *being able* to; no to *whole purpose*!

In these volumes we have emphasised the specification, i.e., the abstraction and the modelling parts of software development, from domain descriptions via requirements prescriptions to, and including, as now, software design. And we have emphasised the engineering, the intuitive relations between these phases. We have emphasised the need for decomposing the development of phase specifications into stages and steps of the development of subsidiary specifications. We presently consider that most important. And we have covered the formalisation aspects of abstraction and modelling very extensively because we find that important. If the software engineer does not understand how to abstract and how to model — and what to abstract and model — thoroughly, as we believe we are teaching that software engineer, then there is no reason to also teach verification.

Now we have done that: taught the software engineer abstraction and modelling. We have presented that material, i.e., principles and techniques of abstraction and modelling, in a way that can be used whether the specification

language is RSL [117, 118], or B [4] (or eventB [5]), or VDM-SL [35, 36, 104], or Z [162, 341, 343, 377], and whether it is combined with Petri nets [196, 273, 293–295], or message sequence charts [182–184], or live sequence charts [73, 149, 203], or statecharts [144, 145, 147, 148, 150], or the duration calculus [381, 382], or not. Now is the time for that software engineer to learn verification. For that we would like to refer to a text on verification which is as independent of the specific specification notations as we believe our treatment of abstraction and modelling has been.

But such a text is not yet available. And we do not see such a text becoming readily available. Alas, to learn verification, i.e., to teach verification, one is still bound, very tightly, to the specific proof systems and the specific proof assistants and/or theorem-provers at hand.

In a sense we rely on the reader to also follow specific courses on program development, courses that contain a significant verification component. In another sense we expect the readers, when actually in industrial practice, and using formal techniques, to then choose and learn by themselves the proof rules, principles, techniques and tools of a specific formal specification language. It is not a small effort. But, we trust, it is an effort that pays off well.

Some specification approaches, notably B [4] and eventB [5], strongly advocate the tight alternation between specifications and proofs. Valid claims are made that one really cannot find, or choose, a most appropriate form of specification without also thinking hard on verification. There is much to that. So much that we really regret not covering verification to any serious depth. We take solace, however, in the fact that what the present volumes have brought in is all still fully relevant.

RAISE [118], as a method, and RSL [117], as its language, comes with proof rules, i.e., a proof system, and tools for assisting proofs and (via translation to PVS [268, 269, 330, 331]) for automated proofs. To learn these, study [118] carefully, and search the Internet (www.iist.unu.edu/raise) for freely downloadable texts and tools. We refer to some articles for examples of principles and techniques of verification and proofs about RSL specifications: [151–154, 219].

There are, today, January 2006, many theorem-proving tools and proof assistant tools: PVS (Predicate Verification System) [268, 269, 330, 331][1], Isabelle/HOL (Higher-Order Logic) [261, 271][2], Raise Tool Set [118][3], and many, many others. For reasonably up-to-date information on proof tools we refer to the seminal Formal Methods Home Page: http://www.afm.sbu.ac.uk/, as maintained by Jonathan P. Bowen.

[1] http://pvs.csl.sri.com/
[2] http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html
[3] Click on open-source software: http://www.iist.unu.edu/

29.5.2 Model Checking

Characterisation. (II) By *model checking* we understand a semantic process: The specification about which one wishes to prove the theorem is modelled, is respecified, reprogrammed, in some model checking language, and that model is symbolically executed. That is, a possibly simplifying model of the specification is created. Some statements are expressed as to the desired state behaviour of that model, i.e. the theorem to be "checked". And the model checker explores the (possibly simplified) model as to its ability to enter desired states (or state sequences). ▪

The Basis for Model Checking

Thus the basis for model checking is a specification (a domain description, a requirements prescription or a software design). Model checking effectively means to base executions of a computer on a usually simplified model of one of these specifications. In addition to the specification and the model (i.e., another specification) there is then a statement (the conjecture) — to be checked — namely that the model specifies certain behaviours. Finally, there is the claim that the model behaviour properly reflects the behaviour as implied by the basis specification.

Model checking is a field of research and application in rapid development. We refer to a number of home pages: The NASA/JPL Laboratory for Reliable Software, Pasadena, California: SPIN;[4] Carnegie Mellon University's Model Checking @ CMU;[5] and the *Formal Systems (Europe) Ltd.* company,[6] which markets tools, FDR2, for, amongst other things, the model checking of CSP specifications.

29.5.3 Testing

In Sects. 14.2.2 and 22.2.2 (on domain engineering, respectively requirements engineering), we briefly alluded to testing, and we promised a somewhat deeper treatment. The present section will now deliver that coverage, but it will not cover testing to the depth needed by the practicing software engineer. For that we refer to Chap. 13 of Hans van Vliet's delightful book on *Software Engineering* [369].

Characterisation. (II) By specification (or software or program) *testing* we understand a systematic effort to refute a claim of correctness of one (i.e., a concrete) specification (for example, a program) with respect to another (the abstract) specification. ▪

[4] http://spinroot.com/spin/whatispin.html
[5] http://www-2.cs.cmu.edu/~modelcheck/
[6] http://www.fsel.com

The Bases for Testing

The bases for testing are (the domain description, or the requirements prescription or) the software design documents. Testing effectively means to base executions of a computer on (one of) these documents. In addition to the documents some data are provided, data which have been selected in such a way as to steer the (in the case of most abstract specifications, symbolic) executions through specific parts of the documents: to test that these parts (describe, prescribe and) specify what is expected, or to test that the documents specify that certain (erroneous) actions cannot take place.

Test Objectives

To deal effectively with testing we need clarify the terms *error, fault* and *failure*. This was done in Sect. 19.6.4. We refer the reader to that section. The next definitions are variants of those given in Sect. 19.6.4.

Characterisation. A software programming *error* is an action that produces an incorrect result. ▪

Characterisation. A software *fault* is the consequence of a human error in the development of that software. ▪

Characterisation. A software fault may result in a (computing) *failure*. ▪

A major software test objective is to find faults in software, and to evaluate, i.e., ascertain, the error that caused the fault. To do so the tester must provoke failures.

Test Strategy

So the test strategy is to find a suitable mix of text cases whose deployment, in testing, will provoke failures. Historically many, sometimes often rather ad hoc, test strategies have been implemented in order to achieve the testing objectives.

Test Coverage

Test coverage is a notion associated with the text of that which is being tested. There are several different coverage criteria:

- *Control flow coverage:* If all paths through a program have been executed in some test involving a test set S (of one or more test data), then we say that the control flow coverage is 100%.

- *Data flow coverage:* is associated with data variables of a program. One may distinguish between variable initialisations, variable assignments and uses of variables, either in conditional tests or in general expression evaluation (in connection with assignments and function arguments. To each of these different "occurrences" of data variables one can associate coverage numbers.
- *Fault seeding:* The deliberate insertion of faults with the aim of checking that tests do indeed catch these faults may convince some developers that these tests then also catch a reasonable proportion of all other faults.
- *Mutation testing:* is a variant of fault seeding: the systematic insertion of as many variants of program subtexts.

Test Cases and Test Suites

Characterisation. A *test case* is a pair: (i) an assignment to initial states, i.e., to those data variables whose values are normally set by proving external input to the program, and (ii) a set of values that the program is expected to output during execution. ▪

For a compiler to be tested then means to provide correct as well as erroneous programs in the language of the compiler.

Characterisation. A set of tests (i.e., initial assignments and execution results) is called a *test suite*. ▪

Test Adequacy Criteria — Test Requirements

By a test adequacy criterion is understood to be a requirement that is put on testing: How much coverage? How many faults to be detected by testing relative to those otherwise found (per N customers) in the field per M months?

Classical Testing Approaches

We shall give only a very brief survey. For more thorough treatments we thus refer the reader to more detailed treatments of testing. Chapter 13 of Hans van Vliet's delightful book on software engineering is, in our mind, a best restart [369]. The *Further Reading* section (Sect. 13.11) of [369] provides authoritative references to further literature.

Black Box/Functional/Specification-Based Testing

In a black-box testing the program (i.e., specification) is not known: only what it is supposed to do. Test cases are derived from specifications.

White Box/Structural/Implementation-Based Testing

In a white-box test we may consider, for example, various coverage criteria. So we know the program structure. Test cases are derived from the program text.

Manual Testing

Several forms of manual testing are possible:

- *Reading:* Always read your specifications (programs) and have others read them too.
- *Walkthroughs* (or Inspection) is reading by a group whose members have been assigned definite roles: moderator, inspector, author, etc. An inspection then starts out with a kind of test cases, called checklists.
- *Scenario-based evaluation* is walkthroughs (inspections) based on "use cases", that is, scenarios in the form of systematic application-oriented behaviours.

Formal Specification-Based Testing Approaches

Many of the above test techniques can be automated. Specification-based testing is sometimes referred to as abstract interpretation. The field of abstract interpretation is "old" [63–71]. The variety of testing that can be explained in terms of abstract interpretation of specifications is growing, day by day. More and more automated abstract interpretation comes to border on the techniques of theorem proving and model checking.

29.5.4 Discussion

We have covered testing ever so lightly. We have failed, in a sense, to present testing as systematically as we would have liked it. We think much research need be done to "clean" up and properly relate the very many testing approaches just hinted at above and to relate them to modern theories. We refer to work on abstract interpretation [63–67, 70, 71], spearheaded by Patrick Cousot, and to work on partial evaluation [201], spearheaded by Neil D. Jones.

29.6 Discussion

We have covered *"Coding and ∀ that!"* ever so summarily: Some, but far from all facets have been covered. The selection is personal. The hope is that the reader, in other university courses of more specific character — functional, imperative, parallel, distributed, OO, etc., programming — will receive what is missing here.

29.7 Exercises

29.7.1 A Preamble

We refer to Sect. 1.7.1 for the list of 15 running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. 1.7.2 concerning the use of the term "selected topic".

29.7.2 The Exercises

Exercise 29.1 *Domain Demo.* For the fixed topic, selected by you, suggest the structure of a domain demo which, in as systematic a way as you can think of, demonstrates, i.e., visualises and possibly animates, the domain being developed.

Exercise 29.2 *Requirements Demo.* For the fixed topic, selected by you, and following your solution to Exercise 29.1, suggest the structure of a requirements demo which, in as systematic a way as you can think of, demonstrates, i.e., visualises and possibly animates, the requirements being developed — and in such a way that you reuse the domain demo.

Exercise 29.3 *Architecture Skeleton.* For the fixed topic, selected by you, and following your solution to Exercise 29.2, suggest the structure of a software architecture skeleton which, in as systematic a way as you can think of, demonstrates, i.e., visualises and possibly animates, the software being designed — and in such a way that you reuse the requirements demo.

Exercise 29.4 *Component Prototype.* For the fixed topic, selected by you, and following your solution to Exercise 29.3, suggest the contents of some components, i.e., suggest a component prototype, which, in as systematic a way as you can think of, demonstrates, i.e., visualises and possibly animates, the components being developed — and in such a way that you reuse the architecture skeleton.

Exercise 29.5 *Test Suites.* For the fixed topic, selected by you, and following your solutions to Exercises 29.1–29.4, suggest test case suites for testing the domain demo, the requirements demo, the software architecture skeleton and the component prototype.

30

The Computing Systems Design Process Model

- The **prerequisite** for studying this chapter is that you have now completed the study of the last of the software development phases and are ready to summarise.
- The **aims** are to review the major stages and steps of software design, and to review the totality of documents needed to present, in a proper, professional manner, the fruits of the software design process.
- The **objective** is to further, and almost finally, ensure that you will become a professional software engineer, one who carefully develops and carefully documents also the software itself.
- The **treatment** is systematic and discursive.

This chapter should be seen in the context of the similar chapters on the *Domain Engineering Process Model,* and the *Requirements Engineering Process Model,* Chaps. 16 and 24, respectively.

30.1 Introduction

The computing systems design process model must take into account such things as the (efficient) algorithm and data structure discovery problem, the sharing of modules between components, the design of the interface "glue" that "links" components and that links modules, and so on. The discussion of this chapter does not reflect these serious concerns, but focuses on the overall software engineering issues rather than on the above-listed programming concerns.

30.2 Review of Software Design

This section should be seen in the context of the similar sections on the *Review of Domain Development*, and the *Review of Requirements Development*, Sects. 16.2 and 24.2, respectively.

30.2.1 A Process Model

Figure 30.1 is intended to diagram an idealised abstraction of the stages and steps of the software design phase. Figure 30.1 is in contrast to the corre-

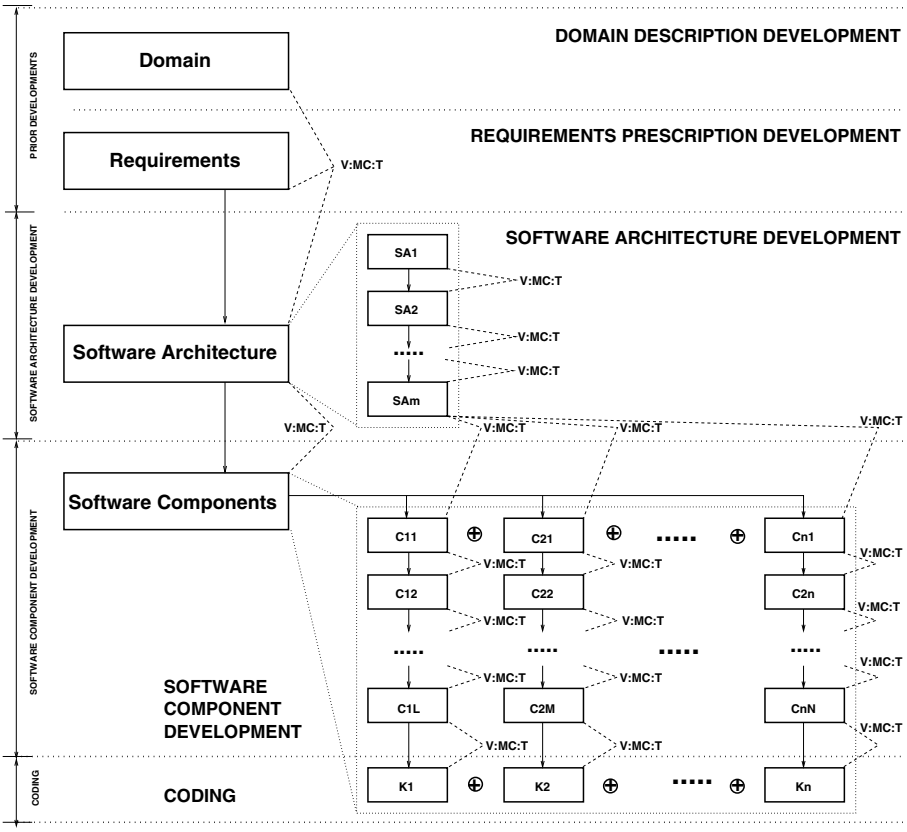


Fig. 30.1. The software design development processes

sponding domain and requirements development process model diagrams of Figs. 16.1, 24.1 and 24.2. We explain Fig. 30.1.

Requirements: The requirements box denotes the activity of requirements development as well as the resulting full requirements documentation (see Fig. 24.3).

Software architecture: The software architecture box denotes the activity of software architecture design as well as the resulting full software architecture design documentation. Either of these can be decomposed into m stages, respectively sets of design documents $\mathrm{SA}_1 \ldots \mathrm{SA}_m$.

..... The dotted frame around the $SA_1 \ldots SA_m$ boxes and the dotted lines "connecting" this frame to the software architecture box, are intended to show that the software architecture box is sequentially decomposable into these $SA_1 \ldots SA_m$ boxes.

V:MC:T: The V:MC:T wedge between the domain, the requirements and the software architecture boxes denotes the possibility of verification, model checking and testing — aimed at building confidence in the correctness of the software architecture design with respect to the requirements (prescription) and the domain (description).

Software components: From a last step, SA_m, of the software architecture design one can decide, as was shown for example in Chaps. 26 and 27, which are suitable components, C_i, and then develop these in a stepwise manner: C_{i_1}, C_{i_2}, ..., C_{i_k}.

.....: The dotted frame around the totality of C_{i_j} boxes and the dotted lines "connecting" this frame to the software components box are intended to show that the software components box thus is decomposable into these C_{i_j} boxes.

Coding: From the last step of development of components, each with their possibly many and, see below, possibly shared modules, follows a final step of coding. Some formal specification language tool sets offer the possibility of automatic translation from last module design step to programming language code. `RSL`, for example, offers automatic translation from low-level `RSL` specifications into `C++` and `SML`.

30.2.2 Discussion

A number of comments are in order. The process model, as diagrammed in Fig. 30.1, does not show any "feedback" lines, but it should. That is, between and within any of the stages — software architecture, components and coding — there ought be dashed arrows leading from any later box to any earlier box, reflecting that a stage or a step may encounter such design problems that necessitate redesign as from some earlier stage or step.

Each component step may involve the design and further stepwise refinement of modules, which is not shown, at all, in Fig. 30.1. Finally, the components stage, with its many steps, should, in a more detailed manner than shown here, reflect that certain parts of the development of one component are shared with the development of another component: During the development of one or more components, design reviews may reveal the possibility of designing shared modules.

Many other remarks could be made. The totality of these and such other remarks indicates that the software design process, as is also the case for the stages and steps of the domain and requirements development processes, cannot be fully prescribed before the work involved in these process parts is undertaken. The developer must be prepared to edit the corresponding process diagrams — and deliver these edited versions as part of the documentation.

30.3 Review of Software Design Documents

This section should be seen in the context of the similar sections on the *Review of Domain Documents*, and the *Review of Requirements Documents*, Sects. 16.3, respectively 24.3. The listing below is intended to represent an idealised abstraction of the table of contents of the full documentation of a software design phase. That listing is in contrast to the corresponding domain and requirements development tables of contents in Sects. 16.3 and 24.4.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management
      ii. Developers
      iii. Consultants
2. Software Specifications
   (a) Architecture Design ($S_{a_1} \ldots S_{a_n}$)
   (b) Component Design ($S_{c_{1_i}} \ldots S_{c_{n_j}}$)
   (c) Module Design ($S_{m_1} \ldots S_{m_m}$)
   (d) Program Coding ($S_{k_1}, \ldots, S_{k_n}$)
3. Analyses
   (a) Analysis Objectives and Strategies
   (b) Verification (S_{i_p}, $S_i \sqsupseteq_{L_i} S_{i+1}$)
      i. Theorems and Lemmas L_i
      ii. Proof Scripts $\wp_i$
      iii. Proofs Π_i
   (c) Model Checking ($S_i \sqsupseteq P_{i-1}$)
      i. Model Checkers
      ii. Propositions P_i
      iii. Model Checks $\mathcal{M}_i$
   (d) Testing ($S_i \sqsupseteq T_i$)
      i. Manual Testing
         • Manual Tests $M_{S_1} \ldots M_{S_\mu}$
      ii. Computerised Testing
         A. Unit (or Module) Tests C_u
         B. Component Tests C_c
         C. Integration Tests C_i
         D. System Tests $C_s \ldots C_{s_{i_{t_s}}}$
   (e) Evaluation of Adequacy of Analysis

Legend:
S Specification
L Theorem or Lemma
$\wp_i$ Proof Scripts
Π_i Proof Listings
P Proposition
$\mathcal{M}$ Model Check (run, report)
T Test Formulation
M Manual Check Report
C Computerised Check (run, report)
$\sqsupseteq$ "is correct with respect to (wrt.)"
$\sqsupseteq_\ell$ "is correct, modulo ℓ, wrt."

We explain the above generic table of contents listing.

1. Information: We refer to Sect. 2.4 for a generic treatment of what goes into the informative documents listed in items 1(a)–1(k).

2. Software Specifications:

2(a). Architecture Design: $S_{a_1} \ldots S_{a_n}$ also indicate several sets of thus stepwise developed software architecture design documents (corresponding to steps SA_1 to SA_n).

2(b). Component Design: By $S_{c_{i_j}}$ we mean the documentation of the ith component's jth step of refinement.

2(c). Module Design: By $S_{m_{i_j}}$ we mean the documentation of the ith module's jth step of refinement. What number (m) of modules need to be designed is determined from the n components. Whether some module S_{m_i} is shared between two or more components is not shown in Fig. 30.1.

2(d). Program Coding: By S_{k_i} we mean the documentation of the ith components collection of one or more module codes. (cf. the Discussion in Sect. 30.2.2.)

3. Analyses:

3(a). Analysis Objectives and Strategies: The developer has to document which kinds of properties are being analysed (verified, model checked and tested), and according to which strategies the analyses are to be pursued: whether verified, model checked and/or tested. Such considerations need to be documented.

3(b). Verification: By S_{i_p} we mean the verification of a property p of specification S_i. By $S_i \sqsupseteq_{L_i} S_{i+1}$ we mean the verification of lemma L_i which purportedly holds of the refinement or transformation S_{i+1} with respect to S_i.

3((b)i). Theorems and Lemmas: Many lemmas (and theorems) L_i can be stated, they usually contain texts of specifications.

3((b)ii). Proof Scripts: By $\wp_i$ we understand a plan for how to conduct a proof. Plans may prescribe that a proof of a lemma is to be decomposed into proofs of auxiliary lemmas.

3((b)iii). Proofs: By Π_i we mean the full documentation of a proof, including S_{i_p} or S_i, S_{i+1}, either of them with L_i.

3(c). Model Checking: By $S_i \sqsupseteq P_{i-1}$ we mean the model checking of a property P_{i-1}, ostensibly one where P_i contains text from S_{i-1}. By $\mathcal{M}_i$ we mean the actual "run" of the model checker with respect to $S_i \sqsupseteq P_{i-1}$.

3(d). Testing: By $S_i \sqsupseteq T_i$ we mean that a specification S_i is being subject to tests with respect to T_i.

3((d)i). Manual Testing: By M_{S_i} we mean the full documentation of a manual test (reading, walkthrough, inspection, etc.) of a specification (incl. code) S_i.

3((d)ii). Computerised Testing:

3((d)iiA). Unit (or Module) Tests: By C_μ we mean the full documentation of a unit (or module) test: the name of the unit (or module) tested, the test data, the test outcome, the configuration of the host system on which the test was performed, the name of the tester and the date.

3((d)iiB). Component Tests: By C_c we mean the full documentation of a component test: the name of the component

tested, the test data, the test outcome, the configuration of the host system on which the test was performed, the name of the tester and the date.

3((d)iiC). Integration Tests: By C_i we mean the full documentation of an integration test: the names of the set of two or more components tested (i.e., the names of all their modules), the test data, the test outcome, the configuration of the host system on which the test was performed, the name of the tester and the date.

3((d)iiD). System Tests: By C_s we mean the full documentation of a complete system test: the name of the entire system (i.e., names of all components, etc.) tested, the test data, the test outcome, the configuration of the host system on which the test was performed, the name of the tester and the date.

3(e). Evaluation of Adequacy of Analysis: A document has to be produced, one which is to be signed by all parties to a software design contract. The document assesses the adequacy of the analyses performed, which went OK, which did not, what possible remedial actions to take, etc.

30.4 Discussion

Our long odyssey is nearing its end. From Fig. 30.1 (and its annotation) and from the table of contents listing — both based on several earlier chapters — we can draw many sobering conclusions. For example, software design, like domain as well as requirements modelling, requires many stages and steps of development, and a great deal of careful documentation. The questions are: Are you ready? Is your software house ready? Do you have the necessary documentation tools?

Part VII

CLOSING

31

The Triptych Development Process Model

- The **prerequisite** for studying this chapter is that you have studied Parts IV–VI of the present volume.
- The **aim** is to summarise, ever so briefly, the combined process model and documentation of a full-scale software development project: from, and including, domain description via requirements prescription to, and including, software design.
- The **objective** is to ensure that you become a full-fledged professional software engineer.
- The **treatment** is discursive.

We shall briefly overview material from earlier chapters.

31.1 Phase Process Models

In this section we shall repeat some process models in the form of their diagrams. Figure "repeat" references will also list their first occurrence. We encourage the reader to go back and review the relevant material given in those referenced earlier chapters.

Figure 31.1 is a repeat of Fig. 1.2 Sect. 1.3.6. It shows the three main phases of software development, (i) domain development, (ii) requirements development and (iii) software design. We emphasise the feedback loops, that is, arrows leading from temporally later boxes to earlier ones. They designate that the developers may have to go back and redo some earlier work, with the consequence that the developer then has to redo all subsequent work from there on.

Figure 31.2 is a repeat of Fig. 1.3, Sect. 1.3.6. It shows the stages of the domain development. A particular ordering of these has been suggested. But it may be that the domain engineer may, depending on the problem at hand, choose another ordering, or that some stages are not necessarily meaningful for some domain development — although it is hard (for me) to believe so!

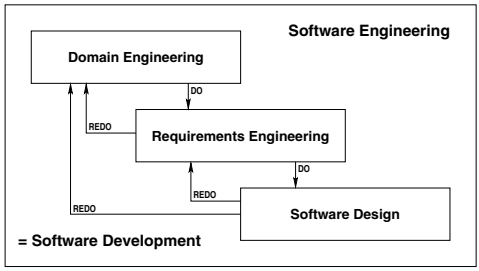


Fig. 31.1. The triptych iterative phase development: Fig. 1.2

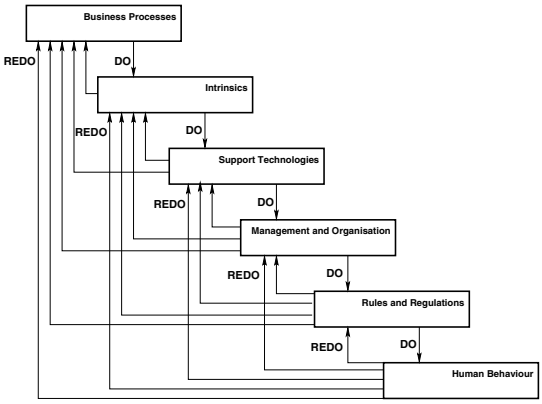


Fig. 31.2. The domain stage iterations: Fig. 1.3

Figure 31.3 is a repeat of Fig. 1.4, Sect. 1.3.6. It shows that not all stages, as hinted at in the domain development diagram of Fig. 31.2, need to be done sequentially. Some, such as the individual stages of machine requirements, can in many cases be done "concurrently", say, by different groups of requirements engineers.

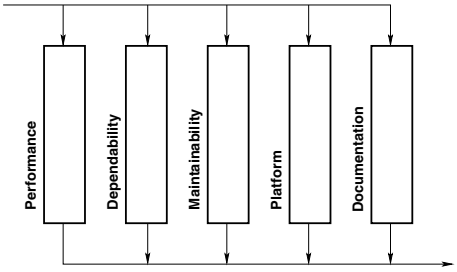


Fig. 31.3. The machine requirements stage iterations: Fig. 1.4

Figure 31.4 is a repeat of Fig. 16.1, Sect. 16.2. It shows Fig. 31.2 in some detail. Figure 31.2 emphasises the stages that goes into the construction proper of the domain model. Figure 31.4 additionally shows the stages prior to and subsequent to the domain modelling stage. Again, we have left out that there usually are feedback loops from any later box in the diagram to potentially any earlier box. Identification of which earlier boxes may have to serve as

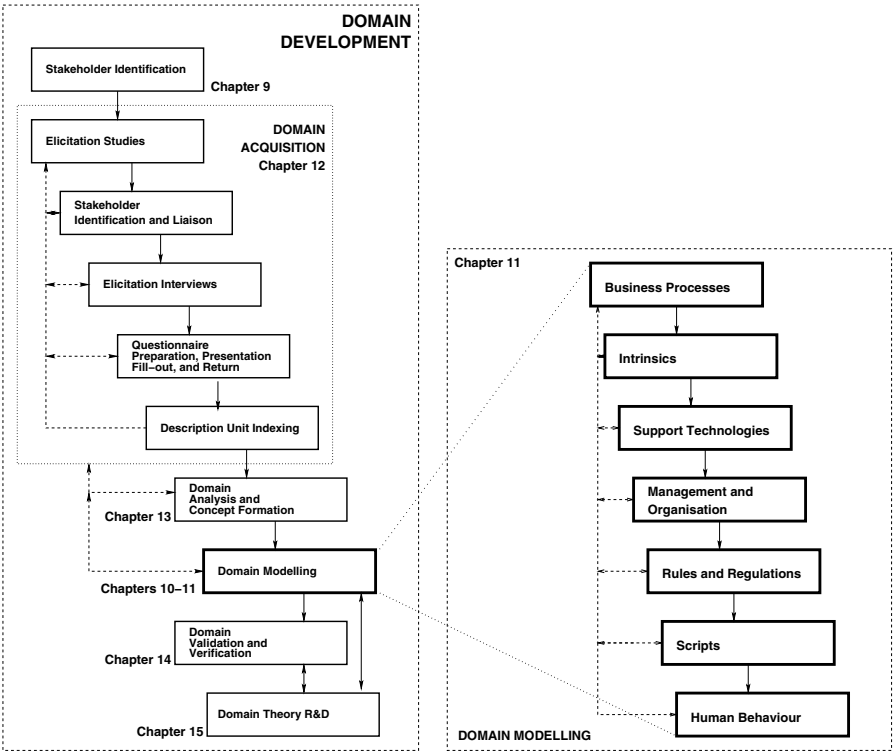
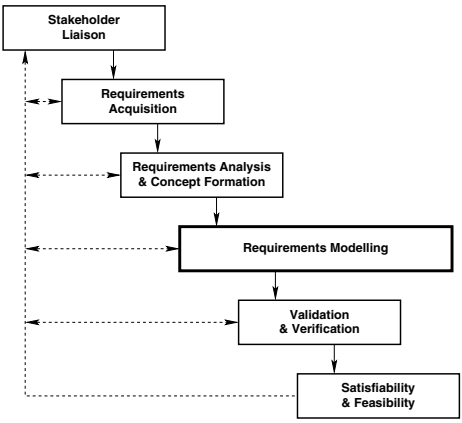


Fig. 31.4. The domain development process model: Fig. 16.1

a restarting point for rework follows from the careful documentation of all parts of development and the ability to trace, backwards, from a *text point of contention* (from where a potential feedback arrow emanates) to the earliest stage or step where the contentious issue may first have arisen.

Figure 31.5 is a repeat of Figs. 24.1 and 24.2, Sect. 24.5. We see that in many ways it is very similar to Fig. 31.4. It contains the same sequence of boxes before and after requirements (domain) modelling. The main difference between the two sets of activities, domain, respectively requirements development, is the modelling stage. They are shown, in respective diagrams, blown up into different (sequential or concurrent) sets of modelling actions.

This similarity between the two phases of development is beneficial. It affords reviews of activities and their results (i.e., documents). It can thus rely on common tools.



The requirements modelling box is shown in detail below:

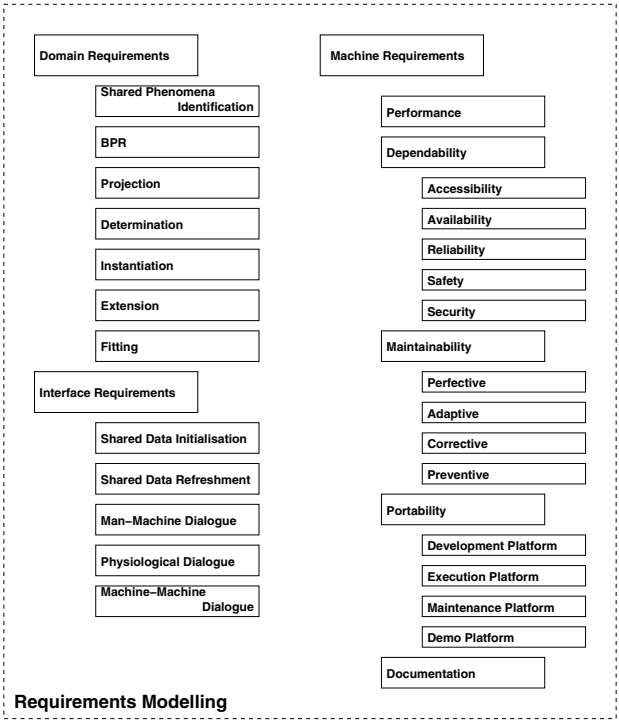


Fig. 31.5. The requirements development process model: Figs. 24.1 and 24.2

Figure 31.6 is a repeat of Fig. 30.1, Chap. 30's Sect. 30.2.1. We see that the software design process is not similar in its flows of actions, etc., to the domain and requirements development processes. The software design process emphasises stepwise refinement and/or transformation. And hence it emphasises repeated verifications, model checks and tests. The software design process is, perhaps, also the process which appeals to the individual programmer. Whereas many stages of the domain and the requirements phases significantly relied on team work, it seems that most stages and steps of the software design phase need the ingenuity of the individual programmer.

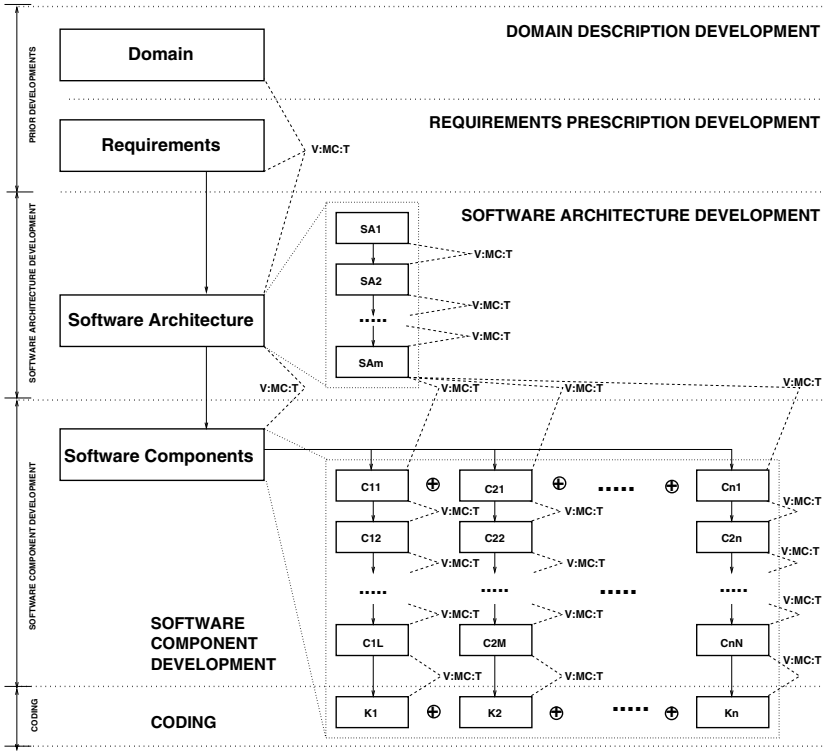


Fig. 31.6. The software design process model: Fig. 30.1

31.2 Phase Documentation Table of Contents

Hand in hand with, i.e., indistinguishable from, the phase (and their stage and step) developments, is the production of documents. We briefly review corresponding classes of development documents. Common to all three listings

(Figs. 31.2, 31.8, and 31.2) are their parts 1: Information, with all its subparts. As was outlined in Sect. 2.4, those informative parts are of an administrative, that is, pragmatic, and thus important nature. The information given in these parts helps us to understand the context in which respective description (prescription) and analysis documents are to be understood. The *domain development* document listing of Fig. 31.2 is a repeat of that of Chap. 16.

Domain Description Documents: Sect. 16.3

1. Information
   - (a) Name, Place and Date
   - (b) Partners
   - (c) Current Situation
   - (d) Needs and Ideas
   - (e) Concepts and Facilities
   - (f) Scope and Span
   - (g) Assumptions and Dependencies
   - (h) Implicit/Derivative Goals
   - (i) Synopsis
   - (j) Standards Compliance
   - (k) Contracts
   - (l) The Teams
     - i. Management
     - ii. Developers
     - iii. Client Staff
     - iv. Consultants
2. Descriptions
   - (a) Stakeholders
   - (b) The Acquisition Process
     - i. Studies
     - ii. Interviews
     - iii. Questionnaires
     - iv. Indexed Description Units
   - (c) Terminology
   - (d) Business Processes
   - (e) Facets:
     - i. Intrinsics
     - ii. Support Technologies
     - iii. Management and Organisation
     - iv. Rules and Regulations
     - v. Scripts
     - vi. Human Behaviour
   - (f) Consolidated Description
3. Analyses
   - (a) Domain Analysis and Concept Formation
     - i. Inconsistencies
     - ii. Conflicts
     - iii. Incompletenesses
     - iv. Resolutions
   - (b) Domain Validation
     - i. Stakeholder Walkthroughs
     - ii. Resolutions
   - (c) Domain Verification
     - i. Model Checkings
     - ii. Theorems and Proofs
     - iii. Test Cases and Tests
   - (d) (Towards a) Domain Theory

Fig. 31.7. Domain Description Documents

It should be emphasised that the contents listings of Figs. 31.2–31.2 only express that the entries mentioned ought be part of the "sum total" documentation. Whether they appear, sequentially, in the full phase documentation, in the order listed, or, for example (as might very well be quite reasonable), whether specification (domain description, requirements prescription and design specification) stage or step parts alternate with appropriate analysis parts, is not mandated. Contracts may specify the exact appearance, ordering, naming conventions, and headings, in all documents.

The *requirements development* document listing of Fig. 31.8 is a repeat of that of Chap. 24.

A generic requirements documentation contents listing: Sect. 24.4

1. Information
   - (a) Name, Place and Date
   - (b) Partners
   - (c) Current Situation
   - (d) Needs and Ideas (Eurekas, I)
   - (e) Concepts and Facilities (Eurekas, II)
   - (f) Scope and Span
   - (g) Assumptions and Dependencies
   - (h) Implicit/Derivative Goals
   - (i) Synopsis (Eurekas, III)
   - (j) Standards Compliance
   - (k) Contracts, with Design Brief
   - (l) The Teams
     - i. Management
     - ii. Developers
     - iii. Client Staff
     - iv. Consultants
2. Prescriptions
   - (a) Stakeholders
   - (b) The Acquisition Process
     - i. Studies
     - ii. Interviews
     - iii. Questionnaires
     - iv. Indexed Description Units
   - (c) Rough Sketches (Eurekas, IV)
   - (d) Terminology
   - (e) Facets:
     - i. BPR
       - Sanctity of Intrinsics
       - Support Technology
       - Management and Organisation
       - Rules and Regulations
       - Human Behaviour
       - Scripting
     - ii. Domain Requirements
       - Projection
       - Determination
       - Instantiation
       - Extension
       - Fitting
     - iii. Interface Requirements
       - Shared Phenomena and Concept Identification
       - Shared Data Initialisation
       - Shared Data Refreshment
       - Man-Machine Dialogue
       - Physiological Interface
       - Machine-Machine Dialogue
     - iv. Machine Requirements
       - Performance
         - ⋆ Storage
         - ⋆ Time
         - ⋆ Software Size
       - Dependability
         - ⋆ Accessibility
         - ⋆ Availability
         - ⋆ Reliability
         - ⋆ Robustness
         - ⋆ Safety
         - ⋆ Security
       - Maintenance
         - ⋆ Adaptive
         - ⋆ Corrective
         - ⋆ Perfective
         - ⋆ Preventive
       - Platform (P)
         - ⋆ Development P
         - ⋆ Demonstration P
         - ⋆ Execution P
         - ⋆ Maintenance P
       - Documentation Requirements
       - Other Requirements
     - v. Full Requirements Facets Documentation
3. Analyses
   - (a) Requirements Analysis and Concept Formation
     - i. Inconsistencies
     - ii. Conflicts
     - iii. Incompletenesses
     - iv. Resolutions
   - (b) Requirements Validation
     - i. Stakeholder Walkthroughs
     - ii. Resolutions
   - (c) Requirements Verification
     - i. Theorem Proofs
     - ii. Model Checks
     - iii. Test Cases and Tests
   - (d) Requirements Theory
   - (e) Satisfiability and Feasibility
     - i. Satisfaction: correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
     - ii. Feasibility: technical, economic, BPR

Fig. 31.8. A generic requirements documentation contents listing

The *software design* document listing of Fig. 31.2 is a repeat of that of Chap. 30.

A generic software design documentation contents listing: Sect. 30.3

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management
      ii. Developers
      iii. Consultants
2. Software Specifications
   (a) Architecture Design ($S_{a_1} \ldots S_{a_n}$)
   (b) Component Design ($S_{c_{1_i}} \ldots S_{c_{n_j}}$)
   (c) Module Design ($S_{m_1} \ldots S_{m_m}$)
   (d) Program Coding ($S_{k_1}, \ldots, S_{k_n}$)
3. Analyses
   (a) Analysis Objectives and Strategies
   (b) Verification (S_{i_p}, $S_i \sqsupseteq_{L_i} S_{i+1}$)
      i. Theorems and Lemmas L_i
      ii. Proof Scripts $\wp_i$
      iii. Proofs Π_i
   (c) Model Checking ($S_i \sqsupseteq P_{i-1}$)
      i. Model Checkers
      ii. Propositions P_i
      iii. Model Checks $\mathcal{M}_i$
   (d) Testing ($S_i \sqsupseteq T_i$)
      i. Manual Testing
         • Manual Tests $M_{S_1} \ldots M_{S_\mu}$
      ii. Computerised Testing
         A. Unit (or Module) Tests C_u
         B. Component Tests C_c
         C. Integration Tests C_i
         D. System Tests $C_s \ldots C_{s_{i_{t_s}}}$
   (e) Evaluation of Adequacy of Analysis

Legend:
S Specification
L Theorem or Lemma
$\wp_i$ Proof Scripts
Π_i Proof Listings
P Proposition
$\mathcal{M}$ Model Check (run, report)
T Test Formulation
M Manual Check Report
C Computerised Check (run, report)
$\sqsupseteq$ "is correct with respect to (wrt.)"
$\sqsupseteq_\ell$ "is correct, modulo ℓ, wrt."

Fig. 31.9. A generic software design documentation contents listing

Typically the documents appear electronically. Typically they could appear on the Web. As in a reasonable handbook (or textbook), large bodies of (pages and pages of) formulas could be conceptually placed in appendixes, i.e., could be invokable or inspectable (displayable, readable), at different zooming levels. Typically these documents would offer quite substantial search and browsing facilities. Ideally every (informal and formal) text part of one step, stage or phase would be linkable backwards and forwards toward: (i) corresponding formal (if informal) or informal (if formal) texts, (ii) previous development phases, stages and steps (where applicable), and (iii) its analysis (verification, model check and tests). Likewise, obviously, the verification, model checking and test parts would be similarly linkable.

31.3 Conclusion

We have ended our presentation of technical and scientific textbook material for Vols. 1–3. It is time to conclude.

32

Finale

- The **prerequisite** for studying this chapter is that you have ended the informal study of Vol. 3, respectively the formal study of all three volumes of this series of textbooks on software engineering. You are ready to conclude — as is the author!
- The **aims** are to briefly comment on the two ways in which the present volume can be studied: informally and formally, to place formal techniques — as covered in all three volumes of these textbooks in software engineering — in a context of conventional software engineering, and to answer some typical "Frequently Asked Questions", to indicate current and future research and tool areas for formal techniques, and to indicate application areas for formal techniques.
- The **objective** is to set you free: you should now be able to embark upon small projects, and, with colleagues, on larger to very large scale projects using the approaches covered in these three volumes on software engineering.
- The **treatment** is systematic and discursive.

32.1 Informal and Formal Software Engineering

Volumes 1 and 2 focused on principles and techniques of abstraction and formal modelling, hence on the formal aspects of software engineering. The present volume has, on one hand, focused on principles and techniques of informal software development, from domains via requirements to software design. On the other hand, the present volume has also shown how domain models, requirements models and software designs can be expressed formally — hence building, as it were, on the material of the previous volumes. Now one might well ask: *Can one do without the "formal stuff" and just do software engineering the informal way?* To this our answer, not surprisingly, in view of the form of the present volume's possibility of being studied informally as well as formally, is given in the next two sections.

32.1.1 Informal Software Engineering

The fundamental division of software development into three phases, domain engineering, requirements engineering and software design, makes good sense, also if pursued using informal tools and techniques. We will go further and claim: Informal software engineering, that is, software development using the principles and techniques of the present volume is a reasonably responsible way of pursuing one's craft. And we will go even further: If one cannot use the principles and techniques of the present volume, then formalisations will not help, they will only obfuscate.

32.1.2 Formal Software Engineering

But one will not reap the real benefits of the principles and techniques of Vols. 1 and 2 unless one also deploys formal techniques and tools (i.e., the formal specification languages). We still claim that it is outright irresponsible engineering not to use the formal techniques and tools of Vols. 1 and 2.

32.1.3 Conclusion

So what is our conclusion? It is that the individual software development group, for every specific project, finds a balance between informal and formal software engineering, one that we could characterise as *formal methods lite!* — one that is commensurate with the demands of the project. To help you find such balance we bring in the next major section of this chapter.

32.2 Myths and Commandments of Formal Methods

As of the year 2005 some, even respectable software engineers and academics, have problems with what they refer to as formal methods, which we prefer to call formal techniques.[1] One often finds that these sceptics voice various concerns. Some in the form of myths or claims. Other concerns reflect hesitancy with respect to how such formal techniques can be inserted into university curricula and into industry.

In this section we shall discuss these and related topics.

The **aim** of this section is to cover some — perhaps, let's hope — historical objections made against formal techniques and related issues. The **objective**

[1] Recall that a good method is a set of principles for selecting and applying a number of principles, techniques and tools in order to [efficiently] analyse a problem and provide (i.e., construct, synthesize) an [efficient] solution to that problem. Given that the principles cannot be formalised in that they most often relate to pragmatic issues — which also cannot be formalised — it does not seem wise to refer to a method as a formal method. So instead we prefer to speak about formal techniques and formally based tools.

of this section is to prepare you with counterarguments should you become engaged in discussions centred around these topics.

32.2.1 First Seven Myths

Anthony Hall [138] lists and dispels the following myths (claims) about formal techniques:[2]

1. *Using formal techniques can* ***guarantee that software is perfect.***

   Of course, use of formal techniques cannot guarantee perfect software. It can, when properly followed, and in most cases indeed does, lead to far more appropriate software.

2. *Formal Techniques are* ***all about program proving.***

   Well, at least in these three volumes of textbooks of software engineering it is not. In these three volumes we have emphasised abstract modelling.

3. *Formal techniques are* ***only useful for safety-critical systems.***

   Formal techniques are useful for any kind of software system, whether a translator (compiler, interpreter), a database information management system, a reactive system, a workpiece (spreadsheet, text processor) system, etc.

4. *Formal techniques* ***require highly trained mathematicians.***

   No, they do not. But they do require software engineers who are willing and able to think abstractly, and here mathematics is a wonderful carrier. To do proofs requires, not highly trained logician mathematicians, but software engineers with a sense of logic, with analytic minds and the ability to reason.

5. *Using formal techniques* ***increases the cost of development.***

   No. In numerous projects (some conducted under the auspices of the European Union's IT research programmes, in the 1980s and the 1990s) it has been demonstrated that using formal techniques did not increase cost of development, and in several cases it decreased the cost. For example, consider the Dansk Datamatik Center's (DDC) very successful development of a full `Ada` compiler [40, 41, 60]. DDC spent around 44 man years to develop a United States Department of Defense validated compiler — while another European and several US companies spent at least three–five times the manpower.

6. *Formal techniques are* ***unacceptable to users.***

[2] We list the "myths" and claims as enumerated in [138], but the subsequent indented comments represent our own views.

Who says users should read formal specifications? In the present volume we have stressed the importance of concurrently developing and maintaining informal as well as formal domain descriptions, requirements prescriptions and software design specifications.

7. *Formal techniques are* ***not used on real, large-scale software.***

   Of course they are. And, where they are not, they should be! Doing otherwise is basically outright criminal, and is cheating the customer — since formal techniques can be used.

We encourage the readers to study Anthony Hall's delightful [138].

32.2.2 Seven More Myths

Jonathan P. Bowen and Michael G. Hinchey [45] builds upon Anthony Hall's analysis [138], and add a further seven "myths" and claims:[3]

8. *Using formal techniques* ***delays the development process.***

   Like item 5 above, using formal techniques does not, in general, delay the development process. It may, and usually will demand that far more time is spent on domain and on requirements modelling, and on early stages of software design. But, also in industrial projects, use of formal techniques has shown to then decrease rather significantly the length and manpower needs of coding.

9. *Formal techniques are* ***not supported by tools.***

   Most formal techniques today come with industry-scale tool sets.

10. *Formal techniques mean* ***forsaking traditional engineering design methods.***

    No. Many traditional engineering methods still apply. Some need to be revised a little.

11. *Formal techniques* ***only apply to software.***

    No. Formal techniques are, interestingly enough, today far more widespread in hard development than in software development. It seems hardware producers are more responsible, since the costs of having to withdraw a chip from the market can easily run into US $ 300 million.

12. *Formal techniques are* ***not required.***

    Yes, they are. In particular, software for military applications, in the UK, now demands the use of formal techniques.

13. *Formal techniques are* ***not supported.***

[3] We list the "myths" and claims as enumerated in [45], but the subsequent indented comments represent our own views.

There are now many software houses, especially in Europe, which offer consultancy advice on the use of formal techniques in other software houses' formal developments.

14. ***"Formal methods" people always use formal methods.***

    Well, we really cannot speak on behalf of all "formal methods people". So, let's leave this one uncommented.

Despite the seeming "outdatedness" of some of the seven more myths, we still encourage the readers to study Bowen and Hinchey's delightful [45].

32.2.3 Ten Formal Methods Commandments

Given that the myths and claims have been disposed of in a trustworthy, believable manner, we can then go on and reiterate what has been said again and again in these volumes: When using formal techniques, please consider carefully the following sound advice from Jonathan P. Bowen and Michael G. Hinchey [46]:[4]

15. **Choose an appropriate notation.**

    Certainly.

16. **Formalise, but not overformalise.**

    What is probably meant here is: Choose an appropriate abstraction level.

17. **Estimate costs.**

    Always.

18. **Have a formal methods guru "on call".**

    See answer to item 13 above.

19. **Do not abandon thy traditional development methods.**

    See answer to item 10 above.

20. **Document sufficiently.**

    This volume in our three-volume series of textbooks on software engineering stresses this point almost to the extreme. In most other engineering practices documentation is far more extensive than what we witness today (year 2005) in software development. So follow the advice of the present volume: *Document, document, document.*

21. **Do not compromise thy quality standards.**

    In fact, tighten your quality standards.

22. **Do not be dogmatic.**

[4] We list the Ten Commandments as listed in [46], but the subsequent indented comments represent our own views.

Creating abstract models and making design decisions as to software data structures and algorithms requires exceedingly open minds. Developing software, in general, requires the effort of at least two, and usually 5–8, people in tight collaboration. Dogmatism, sticking to early development (modelling and design) decisions, simply "is out". Your colleagues will not have it.

23. **Test, test and test again.**

    Also this point has been stressed in the present volume. Besides verification and model checking, it is indeed necessary to test. In this volume we have advocated, in Sect. 29.4.2, an approach to testing which was tied very closely to an approach to software development, from demos, via skeletons, as we called them, and prototypes to actual unit code and systems. The testing went hand in hand with that development.

24. **Reuse.** There are two issues here.

    (a) **Reuse of software designs:** Reuse of modules is what is primarily referred to here. Since we have not covered, in Vol. 2, Chap. 10's treatment of 'modularity' that concept to the depth necessary for large-scale specifications and projects, we really cannot give any qualified advice in this area. It seems to this author that "reuse" is sort of a "white elephant," a desideratum that few can live up to.

    When, for example, a compiler is first developed, its development, all stages, from domain (i.e., language semantics) description via requirements prescription to software design, is being reused. We hope. That is, the company, the group that develops the first compiler, "survives" to make several subsequent generations of that "same" compiler, but now for slightly, or less slightly changed, requirements, for new language features, etc.

    That is reuse at the level we know and are familiar with. For us to think of reusing a module, or even a component, from some problem frame, i.e., from some domain-specific architecture development in an entirely different domain-specific architecture development makes less sense.

    It does, however, make sense in the meaning of that of the Object Management Group's (OMG) guidelines for, say, dictionary components. The kind of dictionaries referred to have a base part which can be reused across compiler, operating system, database, and several application system developments.

    So, really, all we can say is: The jury is still out, and the verdict can be expected in the next decade!

    (b) **Reuse of domain descriptions and requirements prescriptions:** This is an altogether different matter. The very

purpose of developing domain descriptions is that they be reused whenever requirements for software within the application area are being developed.

And even the requirements, for some applications, can be partially reused, i.e., fitted to, requirements for a "neighbouring" area of the same domain.

The cogent observations of [46] has been likewise complemented by [47].

32.3 FAQs: Frequently Asked Questions

32.3.1 General

25. *Should/can/must stakeholders understand formal specifications?*

    No, not necessarily. As we have advocated, proper developments should contain complementary informal and formal descriptions, prescriptions and specifications.

    For a number of software developments, customers may engage consultants to also check that the formal descriptions, prescriptions and specifications are up to standard.

    For a number of software products, insurance companies require that a certified company, like Lloyd's [224] (or such similar companies as Norwegian Veritas [262], Bureau Veritas [51] or TÜV [356]), regularly and irregularly, unannounced, inspect and, in various ways, check the development. Such insurance and "verification" companies are increasingly turning to formal techniques so their staff can understand and professionally evaluate the use of formal descriptions, prescriptions and specifications.

26. *What should/could be the languages of informal descriptions?*

    For domain descriptions it should be the national, i.e., natural language of the client plus the professional language of the domain. No IT jargon is basically needed — unless, of course, IT plays a nontrivial role in the already existing domain.

    For requirements prescriptions the answer is the same as for domain descriptions, except that now one is allowed to use, in appropriate areas of typically interface and machine requirements, an appropriate, generally established sublanguage of IT.

    For software designs — for which we have not dealt with informal annotations to any serious extent — it is, of course, necessary to use the language also of IT (software).

    As for domain-specific languages, make also sure that proper terminologies are established for the IT (software) sublanguages that are used.

27. *What should/could be the languages of formal descriptions?*

Whichever is most appropriate and at hand. For most developments that we know of, i.e., for most problem frames, the RAISE Specification Language, RSL, is adequate. You can then, when and as needed, augment RSL descriptions, prescriptions or specifications with Petri nets [196, 273, 293–295], message sequence charts [182–184], live sequence charts [73, 149, 203], statecharts [144,145,147,148,150] or duration calculus [381,382] descriptions, prescriptions or specifications — or several of these. These "augmentations" were covered, to some nontrivial depth, in Vol. 2, Chaps. 12–15.

Or you can use B [4], eventB, VDM-SL [35,36,104] or Z [162, 341,343,377] — all come, or will soon come, with suitable Petri net, message or live sequence chart, statechart, duration calculus or TLA+ [210,239] augmentations.

RSL variants of UML's Class Diagrams may also be advisable (Vol. 2, Chap. 10).

28. *When have we specified enough — minimum/maximum?*

You have specified enough, both informally and formally, when what is left to describe are such things as identifier formats. That is, when you have specified everything but possibly that, then you have specified the necessary and sufficient amounts. The trivial things left unspecified are those things that one can safely trust the software designer to make final and trustworthy design decisions about. Also, certain aspects of graphical user interfaces, specific handling of tactile input, etc., seem to belong to this class of initially unspecified things.

32.3.2 Domains

29. *Why domain engineering by computing scientists and software engineers?*

Because computing science has the tools, namely the specification languages, and because computing science has the principles and techniques of abstract modelling. Mathematicians — in some sense — could be claimed to have similar such tools, but they really do not. Their abstractions go well beyond those that are needed for domain modelling. They are not interested in proof systems, for example, for formal specifications — but in the more general notions of power of such proof systems, etc. Finally, the computing scientists interface, daily, with software engineers — and, in the hard realities of the day, domain theories are the first to be demanded by software engineers.

30. *Should one use normative and/or instantiated domain descriptions?*

This is a contentious issue. For a specific requirements development one may be tricked into developing only an instantiated domain description, that is, a domain description that is already

instantiated to the specific domain. But, as we have seen in this volume, Part IV, it is oftentimes far more convenient to develop highly reusable, cf. item 24(b) above, domain descriptions.

Some authors seem, in their writing, to assume instantiated domain descriptions. The author of this volume advises normative, i.e., generic, domain descriptions.

31. *Who should research and develop domain theories?*

There are basically three possibilities, listed in causal order:

- initially university and academic research centre **computing science** departments, i.e., their staff,
- eventually **domain-specific** university and academic research centre departments, and
- finally, domain-specific **commercial companies.**

Initially it is advised that university and academic research centre **computing science** scientists research and develop domain models. As mentioned above, in item 29, initially the computing scientists have the basic methods needed to do domain theory research, and are also interested in the engineering of large-scale documentation, etc.

But eventually, within years, say 3–5 years after the initial start of computing science R&D in domain theories, it should also be undertaken by **domain-specific** research groups: transportation, in healthcare, in financial services, in marketing and sales (e-marketing), etc. Just as such university departments are, today, using (applied) mathematics, we can foresee that they will also be able, soon, to use even fairly sophisticated computing science ideas.

And, finally, private, **commercial companies,** for example, software houses strong in a particular application domain, will embark on such domain theory R&D, as will suppliers of any form of technology to companies within the domain.

32. *What is the timeframe for the R&D of domain theories?*

It is strongly believed that the timeframe for the R&D of domain theories is of the order of 10 to 20 years, or in cases up to 30 years, before one can safely say that a domain theory has been established.

In other words: Patience is called for. Conviction that establishing such theories is of utmost importance is called for.

To do research and development on domain theories seems to belong to the category of "Grand Challenge" endeavours (Sect. 32.4.2).

32.3.3 Requirements

To us, there are basically only two questions concerning requirements development:

33. *Requirements always change, so why formalise?*

    No! It may be true that people conceive of requirements "always changing". But we venture to claim that such "changes" are really not so much "changing requirements" as they are, or reflect, increased, and hence better, understanding of the domain.

    In other words: Given that one had an established, i.e., a reasonably comprehensive, domain theory, we will then claim that requirements do not change "so much" (as before conceived)!

34. *Must we formulate requirements strictly before software design?* This question could also appear in the previous section as: *Must we determine domain descriptions strictly before requirements prescriptions?*

    In both cases the answer is: Yes, for the time being. Till such a time when we do indeed have (i) reasonably firmly established domain theories, and (ii) an insufficient body of knowledge, i.e., experience with requirements "strictly derived" from domain theories, until such a time we are, due to commercial, i.e., competitive, pressures, more or less forced to develop domain descriptions hand in hand with requirements prescriptions, and the latter hand in hand with early stages of software design. The special approach to software development advocated in Sect. 29.4 shows a way in which to develop domain descriptions "staggered" with the development of requirements prescriptions, "staggered" with the development of software architecture design — where, by "staggering", we mean that one phase follows almost right "on the heels" of the preceding phase.

32.4 Research and Tool Development

These three volumes of textbooks on software engineering represent a state of the art as of the winter of 2005/2006.

32.4.1 Evolving Principles, Techniques and Tools

As programming methodology and computing science (i.e., foundational) research progress, the present development principles and techniques will evolve, and more elegant forms of these can be expected. New, formal specification languages will emerge. And tools for their use, including verification, model checking and testing tools will be constructed. One thing seems, however, to

be an assurance: these new principles techniques and tools (the latter including the new languages), will not deviate radically from what these volumes have shown.

32.4.2 Grand Challenges

To *put a man on the moon* was a grand technological as well as a scientific grand challenge. To embark upon, conduct and complete the *human genome project* was likewise a grand challenge.

Three Dimensions of Grand Challenges

Related to the material of this series of textbooks on software engineering we can formulate three sets of grand challenges: (i) *integration of formal techniques*, (ii) *trustworthy evolutionary systems development*, and (iii) *domain theories*. We will briefly remark on these.

Integration of Formal Techniques

Volume 2, Chaps. 10, 12–15 introduced UML class diagrams, Petri nets, message and live sequence charts, statecharts and the duration calculus. The chapters suggested that, when appropriate, these other notational, mostly diagrammatic systems be used in conjunction with, for example, `RSL`. The formal issue is: How does the semantics of `RSL` fit with the semantics of UML class diagrams, Petri nets, message and live sequence charts, statecharts and the duration calculus?

The referenced chapters gave some hints. But "the jury is still out!"

Much research and much experimental development still has to be done before we deploy these combinations or integrations in common industrial practice. For now they can be used in carefully monitored and integrated formal techniques guru-tutored industrial developments. We refer to a series of conferences on *IFM: (Integrated Formal Methods)*, which are held annually, for references to ongoing R&D [16, 43, 52, 132, 299].

- *We consider it a 'Grand Challenge' to achieve a set of formal techniques and formally based tools which together cover software development for all of today's and the immediately foreseeable applications.*

Trustworthy Evolutionary Systems Development

Software systems evolve. From when they are first delivered till they are finally disposed of they usually undergo many, many changes, that is, they are maintained: Corrected (for bugs), perfected (new functionalities are added, old functionalities are, resource-consumption-wise, made more efficient), and adapted (to new platforms). Software systems evolution, the proper handling

of legacy systems, i.e., systems that have been in use, say, for decades, is a major problem. The use of formal techniques in the initial development of these is no hindrance, but, we strongly believe, the non use of formal techniques and/or the absence of proper, fully comprehensive documentation, is an obstacle to smooth, problem-free evolution.

- *We consider it a grand challenge to achieve a set of development principles and techniques as well as a set of management practices which together cover all of today's and the immediately foreseeable applications and which by careful use — and reuse — can ensure software systems whose evolution, from initial development, via repeated adaptive and perfective maintenance, to final disposition, perhaps decades later, ensure as near bug-free software as is humanly conceivable.*

Domain Theories

We repeat our adage: software cannot be designed before we have a reasonable grasp of its requirements; requirements cannot be prescribed before we have a reasonable grasp of the domain of the software; and hence it is of utmost importance, as this volume attests, to (somehow) build requirements development on domain theories. The somehow hedge makes room for the developers to codevelop the domain description and the requirements prescription.

- *We consider the following to be examples of grand challenges: to achieve domain theories for such domains as railways, transportation in general, the market (buyers and sellers: consumers, retailers, wholesalers, producers, brokers, distributors, etc.), healthcare, financial services (banks, insurance companies, securities instrument brokers and traders, stock (etc.) exchanges, portfolio management, etc.), and production (i.e., manufacturing), etc.*

On the Nature of "Grand Challenges"

Tony Hoare has formulated 17 criteria for a research topic to be a grand challenge. We borrow the topic lines from [169], but edit, i.e., shorten Hoare's, as usual, poignant, discussion. In other words, we strongly encourage the reader to study Hoare's paper.

The "it" below refers to "a grand challenge".

1. **Fundamental:** It relates strongly to foundations, and the nature and limits of a discipline.
2. **Astonishing:** It implies constructing something ambitious, heretofore not imagined.
3. **Testable:** It must be objectively decidable whether a grand challenge project endeavour is succeeding or failing.
4. **Revolutionary:** It must imply radical paradigm shifts.

5. **Research-oriented:** It can be achieved by methods of academic research — and is not likely to be met solely by commercial interests.
6. **Inspiring:** Almost the entire research community must support it, enthusiastically, even while not all may be engaged in the endeavour.
7. **Understandable:** Comprehensible by — and captures the imagination of — the general public.
8. **Challenging:** Goes beyond what is initially possible and requires insight, techniques and tools not available at the start of the project.
9. **Useful:** Results in scientific or other rewards — even if the project as a whole may fail.
10. **International:** It has international scope: Participation would increase the research profile of a nation.
11. **Historical:** It will eventually be said: It was formulated years ago, and will stand for years to come.
12. **Feasible:** Reasons for previous failures are now understood and can now be overcome.
13. **Incremental:** Decomposes into identified individual research goals.
14. **Cooperative:** Calls for loosely planned cooperation between research teams.
15. **Competitive:** Encourages and benefits from competition among individuals and teams — with clear criteria on who is winning, or who has won.
16. **Effective:** General awareness and spread of results changes attitudes and activities of scientists and engineers.
17. **Risk-Managed:** Risks of failure are identified and means to meet will be applied.

32.5 Application Areas

The present three volumes on software engineering, of which the one you have in front of you now is the third, have, in their very many examples, hinted at a great number of application areas.

32.5.1 Additional Areas

With this section we shall try to complement that list with yet some more examples. But the examples will only be dealt with in a discursive manner. For each of a number of such examples, we will briefly outline the application area and then refer to a monograph, a book, in which the example is covered to some non-trivial depth.

We mention the following books:

- I. Hayes (ed.): *Specification Case Studies* (Prentice Hall, 1987), [156].

- C. Jones, R. Shaw (eds.): *Case Studies in Systematic Software Development* (Prentice Hall, 1990), [199].
- H.D. Van, C. George, T. Janowski, R. Moore (eds.) [75]: *Specification Case Studies in RAISE* (Springer, April 2002), [75].

Needless to say: they (should) all belong in the reference library of the professional software engineer.

32.5.2 The Examples

In the list below chapter references are to chapters in the above mentioned and below repeated first references. Second, separately bracketed references are to individual papers (chapters).

1. **The UNIX Filing System:** Chap. 4 [156] [250]
   Title explains the application.
2. **CAVIAR: Visitor Information System:** Chap. 5 [156] [107]
   A reasonably sophisticated company visitor and meeting (room reservation) system is developed.
3. **The IBM CICS Transaction System:** Chaps. 14–17 [156] [157]
   A number of papers outline the major legacy system reengineering of the IBM Customer Information and Control System (CICS).
4. **A Proof Assistant:** Chap. 4 [199] [248]
   The design of a proof assistant system, with theorem store, proof verification, etc., is carefully argued.
5. **Unification:** Chaps. 5, 6 [199] [105]
   Two chapters outline fundamental aspects of unification, a technique used extensively in proof systems, and in rewrite systems, including interpreters for, for example, logic programming languages.
6. **Storage:** Chaps. 7, 8 [199] [119]
   Two papers investigate heap storage and garbage collection.
7. **Graphics:** Chap. 13 [199] [232]
   Paper investigates and formalises line representations on graphics devices.
8. **A University Library System:** Chap. 3 [75] [267]
   A reasonably sophisticated library system is developed.
9. **A Radio Communications-Based Telephone Switching System:** Chap. 4 [75] [91]
   In a fascinating development, a system for radio communication-based telephony for The Philippines is developed. It involved a centralised station and some 40 (Philippine island remote) stations, time-division multiplexing (TDM), and many other technology-based hardware equipment factors. This careful, stepwise development unfolds towards an implementable system.
10. **A Ministry of Finance Information System:** Chap. 5 [75] [218]
    Developed for the Vietnam Ministry of Finance, this system involves the Taxation, the Budget and the Treasury Departments as well as all the

actions within and between them: from assessment of tax bases, via the budgeting for all ministries, to the collection of taxes.
We refer to exercise item 15 of Sect. 1.7.1 for an abstract description of the domain of this project.

11. **Multilingual Document Processing:** Chap. 6 [75] [97]
A system is developed for processing (creating, editing, communicating and displaying) documents containing any number of scripts for any combination of the four script directions: horizontal left-to-right (say English), horizontal right-to-left (say Arabic), vertical left-to-right (say Mongol) and vertical right-to-left (say old Chinese and Japanese).
12. **Production Processes:** Chap. 7 [75] [266]
A manufacturing system is developed, one which involves production cells, stock handling and all the related processes.
13. **Travel Planning:** Chap. 8 [75] [334]
A reasonably sophisticated travel planning system is developed.
14. **Authentication:** Chap. 9 [75] [351]
Some safety properties of authentication protocols are formulated and proven.
15. **Spatial Graphics:** Chap. 10 [75] [260]
A model of (what is called) the Realm data structure and its operations is given. The Realm data structure is used in representing three-dimensional spatial data and operations on these.

32.6 Closing Remarks

32.6.1 On Programming, Engineering and Management

Most, if not all, software engineering texts and handbooks concentrate on the management aspects and on the informal, human-centred facets of programming and engineering.

We have, in these three volumes, focused on abstraction and both informal and formal modelling of systems and languages; we have focused on the development principles and techniques of a new kind of engineering: domain engineering; and we have brought an altogether new focus to bear on the phases of requirements engineering and software design. Perhaps for that reason we have yet to cover the management aspects. The new focus, to remind the reader, was based on all software development being initially based on extensive and serious domain engineering. The stage of domain requirements, and the stages of design software from either and all of the three domain, interface and machine requirements, highlight the new focus — as does the insistence on codeveloping both informal and formal specifications.

From this triptych view of software engineering springs a new awareness of software development management. Such management is predicated on the

systematic ("light") or rigorous or even formal use of the principles and techniques of these three volumes. Traditional engineering management is predicated by laws of natural science and must consider human factors. Software engineering management, in contrast, is predicated on the more mathematical theories of computing science and the application domains, and then must consider human factors. Software development management is a fascinating area. But it is not one that we feel competent to "preach" about.

32.6.2 Current Software Engineering Edifices

In today's software engineering there are usually no two similar software engineering solutions to identical or near-identical problems. Software systems, from different suppliers, but for near-identical application problems usually offer significantly different user interfaces; and oftentimes vastly different ("hidden") implementations. Each such software system usually requires significant training. Users switching from one product to what ought be a similar product most often require significant retraining. Such "reusers" typically do not recognize that these distinct software products are providing near-identical solutions. As a result users become "religious" about software systems that they are using. Companies, for fear of retraining costs, when seeking new staff usually advertise that they are using "such-and-such" software products and that applicants must have the proverbial "two and a half years" prior experience with this product. I consider this a disgrace to our industry. An airline pilot with Airbus airplane flight experience can with predictable and acceptable costs be retrained for Boeing airplanes. And conversely. Many application solutions require that their users learn a whole vocabulary of concepts. Typically these vocabularies are not (theory of) domain-oriented; sometimes they are somewhat requirements-oriented; and usually they are strongly implementation-oriented. In any case such vocabularies are detrimental to the intellect of their (forced) users.

32.6.3 Current Software Engineering Jargon

Not all software is end-user software — in the sense of these users being people who have not been trained in IT in general. Two categories of software (packages) that can be characterised as not being end-user software (packages) are computing systems base software like database systems, compilers, multiple-user operating systems, and so on, and so-called middleware. The current jargon defines middleware software as *software that allows "front ends" like web browsers/servers or other end-user software packages to communicate with "back end" base software like database management systems (i.e., databases).*

32.6.4 A New View on Software Engineering

The main messages of these volumes were: The diligent reader should by now have gained a view of software engineering which is rather different from the view we think is usually propagated by traditional textbooks. The message here is that software engineering is a highly intellectual activity. In addition to good engineering analysis, software engineering emphasises writing beautiful documents. The view is also characterised by reasoning over texts, and calculation, in the form of transformation (refinement and reification), verification, model checking and tests, calculations that take formal texts and yield formal texts. Software engineering is only to a small extent based on the natural sciences. Software engineering is primarily based on computing science and hence on the mathematical disciplines of logic, recursive function theory and modern algebra. The above view applies also when the principles and techniques of these three volumes are applied in their informal version. When applied in the formal version the message covers a spectrum of "formality": from systematic ("lightweight"), via rigorous, to (fully) formal uses of formal techniques. This view and this message is carried most forcefully by Vols. 1 and 2 of this series. If the reader only follows the first message (and therefore hardly deploys even the lightweight formal techniques approach), or follows either of the systematic to formal approaches and then finds, after having studied these volumes, that her view on software engineering has changed accordingly, then the author has achieved a main objective.

Part VIII

APPENDIXES

A

An RSL Primer

A.1 Types

This is a very brief refresher on the RAISE Specification Language RSL. The reader is kindly asked to study first the decomposition of this section into its subparts and sub-subparts.

A.1.1 Type Expressions

RSL has a number of *built-in* types. There are the Booleans, integers, natural numbers, reals, characters and texts. 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 the following (save the [i] line numbers) are generic type expressions:

Formal Expressions

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

- [7] A-**set**
- [8] A-**infset**
- [9] A × B × ... × C
- [10] A^*
- [11] A^ω
- [12] A $\xrightarrow[m]{}$ B
- [13] A → B

[14] A $\xrightarrow{\sim}$ B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

Annotations:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., $-2, -1, 0, 1, 2, \ldots$.
3. The natural number type of positive integer values $0, 1, 2, \ldots$.
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", ...[1]
6. The text type of character string values "aa", "aaa", ..., "abc", ...
7. The set type of finite set values, see below.
8. The set type of infinite set values.
9. The Cartesian type of Cartesian values, see below.
10. The list type of finite list values, see below.
11. The list type of infinite list values.
12. The map type of finite map values, see below.
13. The function type of total function values, see below.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf. 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.
16. The (postulated disjoint) union of types A, B, ..., and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., and 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, ..., and bv are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

[1] RSL uses double quotes " on both sides of a character and a character string rather than usual balanced quotes "..."

A.1.2 Type Definitions

Concrete Types:

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

Formal Expressions

```
type
   A = Type_expr
```

Some schematic type definitions are:

Formal Expressions

```
[1] Type_name = Type_expr /∗ without |s or sub-types ∗/
[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) |}
```

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

Formal Expressions

```
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)
```

Subtypes

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

Formal Expressions

```
type
   A = {| b:B • 𝒫(b) |}
```

Sorts (Abstract Types)

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

Formal Expressions

type
A, B, ..., C

A.2 The RSL Predicate Calculus

A.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values. Then

Formal Expressions

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

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

A.2.2 Simple Predicate Expressions

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

Formal Expressions

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

are simple predicate expressions.

A.2.3 Quantified Expressions

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

Formal Expressions

$$\forall\ \text{x:X} \bullet \mathcal{P}(x)$$
$$\exists\ \text{y:Y} \bullet \mathcal{Q}(y)$$
$$\exists\ !\ \text{z:Z} \bullet \mathcal{R}(z)$$

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.*

A.3 Concrete RSL Types

A.3.1 Set Enumerations

Let the below as denote values of type A, then the below designate simple set enumerations:

Formal Expressions

$$\{\{\}, \{\text{a}\}, \{\text{a}_1,\text{a}_2,...,\text{a}_m\}, ...\} \in \text{A-}\mathbf{set}$$
$$\{\{\}, \{\text{a}\}, \{\text{a}_1,\text{a}_2,...,\text{a}_m\}, ..., \{\text{a}_1,\text{a}_2,...\}\} \in \text{A-}\mathbf{infset}$$

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

Formal Expressions

type
  A, B
  $\text{P} = \text{A} \to \mathbf{Bool}$
  $\text{Q} = \text{A} \xrightarrow{\sim} \text{B}$
value
  comprehend: A-**infset** $\times$ P $\times$ Q $\to$ B-**infset**
  $\text{comprehend}(\text{s},\mathcal{P},\mathcal{Q}) \equiv \{\ \mathcal{Q}(\text{a}) \mid \text{a:A} \bullet \text{a} \in \text{s} \wedge \mathcal{P}(\text{a})\ \}$

A.3.2 Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, ..., C (allowing indexing for solving ambiguity), then the below expressions are simple Cartesian enumerations:

Formal Expressions

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

A.3.3 List Enumerations

Let a range over values of type A (allowing indexing for solving ambiguity), then the below expressions are simple list enumerations:

Formal Expressions

```
   {⟨⟩, ⟨a⟩, ..., ⟨a1,a2,...,am⟩, ...} ∈ A*
   {⟨⟩, ⟨a⟩, ..., ⟨a1,a2,...,am⟩, ..., ⟨a1,a2,...,am,... ⟩, ...} ∈ Aω

   ⟨ ei .. ej ⟩
```

The last line above assumes e_i and e_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.

The last line below expresses list comprehension.

Formal Expressions

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

A.3.4 Map Enumerations

Let a and b range over values of type A and B, respectively (allowing indexing for solving ambiguity). Then the below expressions are simple map enumerations:

Formal Expressions

```
type
   A, B
   M = A ~m~> B
value
   a,a1,a2,...,a3:A, b,b1,b2,...,b3:B

   [], [a↦b], ..., [a1↦b1,a2↦b2,...,a3↦b3] ∀ ∈ M
```

The last line below expresses map comprehension:

Formal Expressions

```
type
   A, B, C, D
   M = A ~m~> B
   F = A ~→ C
   G = B ~→ D
   P = A → Bool
value
   comprehend: M×F×G×P → (C ~m~> D)
   comprehend(m,F,G,P) ≡
      [ F(a) ↦ G(m(a)) | a:A • a ∈ dom m ∧ P(a) ]
```

A.3.5 Set Operations

Formal Expressions

```
value
   ∈: A × A-infset → Bool
   ∉: A × A-infset → Bool
   ∪: A-infset × A-infset → A-infset
   ∪: (A-infset)-infset → A-infset
   ∩: A-infset × A-infset → A-infset
   ∩: (A-infset)-infset → A-infset
   \: A-infset × A-infset → A-infset
   ⊂: A-infset × A-infset → Bool
```

$\subseteq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
$=$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
$\neq$: A-**infset** $\times$ A-**infset** $\to$ **Bool**
card: A-**infset** $\xrightarrow{\sim}$ **Nat**

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

Annotations:

- $\in$ The membership operator expresses that an element is member of a set.
- $\notin$ The nonmembership operator expresses that an element is not member of a set.
- $\cup$ 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.
- $\cap$ The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- $\setminus$ 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.
- $\subseteq$ The proper subset operator expresses that all members of the left operand set are also in the right operand set.
- $\subset$ 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.
- $=$ The equal operator expresses that the two operand sets are identical.
- $\neq$ The nonequal operator expresses that the two operand sets are *not* identical.
- **card** The cardinality operator gives the number of elements in a (finite) set.

The operations can be defined as follows:

Formal Expressions

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

A.3.6 Cartesian Operations

Formal Expressions

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

A.3.7 List Operations

Formal Expressions

value
hd: $A^\omega \xrightarrow{\sim} A$
tl: $A^\omega \xrightarrow{\sim} A^\omega$
len: $A^\omega \xrightarrow{\sim}$ **Nat**
inds: $A^\omega \rightarrow$ **Nat-infset**
elems: $A^\omega \rightarrow$ A-**infset**
.(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
$\frown$: $A^* \times A^\omega \rightarrow A^\omega$
=: $A^\omega \times A^\omega \rightarrow$ **Bool**
$\neq$: $A^\omega \times A^\omega \rightarrow$ **Bool**

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⟩$\frown$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ $\neq$ ⟨a,b,d⟩

Annotations:

- **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 gives the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems** Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$ Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$ Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- = The equal operator expresses that the two operand lists are identical.
- $\neq$ The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

Formal Expressions

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }

  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩^q′ in a end
          else chaos end
      else q(i−1) end

  fq ^ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
      | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

A.3.8 Map Operations

Formal Expressions

```
value
  m(a): M → A ~→ B, m(a) = b
```

dom: M → A-**infset** [domain of map]
  dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

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

†: M × M → M [override extension]
  [a↦b,a′↦b′,a″↦b″] † [a′↦b‴,a‴↦b′] = [a↦b,a′↦b‴,a‴↦b′]

∪: M × M → M [merge ∪]
  [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]

\: M × A-**infset** → M [restriction by]
  [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]

/: M × A-**infset** → M [restriction to]
  [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

=,≠: M × M → **Bool**

°: (A $\xrightarrow[m]{}$ B) × (B $\xrightarrow[m]{}$ C) → (A $\xrightarrow[m]{}$ C) [composition]
  [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]

Annotations:

- $m(a)$ Application gives the element of which 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.
- \: 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.

The map operations can also be defined as follows:

Formal Expressions

```
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
```

A.4 λ-Calculus and Functions

RSL supports function expressions for λ-abstraction.

A.4.1 The λ-Calculus Syntax

Formal Expressions

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

```
    ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
    ⟨L⟩: e, f, a, ...
    ⟨V⟩: x, ...
    ⟨F⟩: λ x • e, ...
    ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

A.4.2 Free and Bound Variables

Formal Expressions

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).

A.4.3 Substitution

In RSL, the following rules for substitution apply:

Formal Expressions

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,
  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),
  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),
  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

A.4.4 α-Renaming and β-Reduction

Formal Expressions

- 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: $(\lambda x \bullet M)(N)$
  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.
  $(\lambda x \bullet M)(N) \equiv$ **subst**$([N/x]M)$

A.4.5 Function Signatures

For some functions, we want to abstract from the function body:

Formal Expressions

```
value
  obs_Pos_Aircraft: Aircraft → Pos,
  move: Aircraft × Dir → Aircraft,
```

A.4.6 Function Definitions

Functions — with body — can be defined explicitly

Formal Expressions

```
value
  f: A × B × C → D
  f(a,b,c) ≡ Value_Expr

  g: B-infset × (D →m C-set) ~→ A*
  g(bs,dm) ≡ Value_Expr
  pre P(dm)
```

or implicitly

Formal Expressions

```
value
  f: A × B × C → D
  f(a,b,c) as d
  post P1(d)

  g: B-infset × (D →m C-set) ~→ A*
  g(bs,dm) as al
```

pre $\mathcal{P}_2$(dm)
post $\mathcal{P}_3$(al)

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

A.5 Further Applicative Expressions

A.5.1 Let Expressions

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

Formal Expressions

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

is an "expanded" form of

Formal Expressions

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

Recursive **let** expressions are written as:

Formal Expressions

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F ≡ λg•λa•(E(g)) and YF = F(YF)

Predicative **let** expressions:

Formal 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).

Patterns and *wild cards* can be used:

Formal Expressions

```
let {a} ∪ s = set in ... end
let {a,_} ∪ s = set in ... end

let (a,b,...,c) = cart in ... end
let (a,_,...,c) = cart in ... end

let ⟨a⟩^ℓ = list in ... end
let ⟨a,_,b⟩^ℓ = list in ... end

let [a↦b] ∪ m = map in ... end
let [a↦b,_] ∪ m = map in ... end
```

A.5.2 Conditionals

Various kinds of conditional expressions are offered by RSL:

Formal Expressions

```
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_exprt_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
```

A.5.3 Operator/Operand Expressions

Formal Expressions

⟨Expr⟩ ::=
⟨Prefix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Suffix_Op⟩
| ...
⟨Prefix_Op⟩ ::=
− | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
⟨Infix_Op⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | °
⟨Suffix_Op⟩ ::= !

A.6 Imperative Constructs

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.

A.6.1 Variables and Assignment

Formal Expressions

0. **variable** v:Type := expression
1. v := expr

A.6.2 Statement Sequences and skip

Sequencing is done using the ";" operator. **skip** is the empty statement having no value or side effect.

Formal Expressions

2. **skip**
3. stm_1;stm_2;...;stm_n

A.6.3 Imperative Conditionals

Formal Expressions

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

A.6.4 Iterative Conditionals

Formal Expressions

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

A.6.5 Iterative Sequencing

Formal Expressions

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

A.7 Process Constructs

A.7.1 Process Channels

Let A, B stand for types of channel messages and KIdx stand for channel array indexes. Then

Formal Expressions

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and an array of channels, k, whose individual channels, k[i], are able to communicate values of the designated types.

A.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels.

Let P() and Q(i) stand for process expressions[2] then

Formal Expressions

P() ‖ Q(i)	Parallel composition
P() [] Q(i)	Nondeterministic External Choice (either/or)
P() ⌈⌉ Q(i)	Nondeterministic Internal Choice (either/or)
P() ‡ Q()	Interlock Parallel composition

expresses the parallel (‖) of two processes, 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.

A.7.3 Input/Output Events

Let c and k[i] designate channels of type A, and let e designate an expression also of type A. Then

Formal Expressions

c ?, k[i] ?	Input expression (a clause)
c ! e, k[i] ! e	Output clause (a statement)

expresses the willingness to engage in an event that "reads" an input, and respectively "writes" an output.

A.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Formal Expressions

value

[2] Both expressions (P() and (Q(i)) name process definitions (P respectively Q). P has no formal parameters. Q has, as only parameter, a channel array index. The former, P(), thus invokes P with no arguments and the latter, Q(i), invokes Q with a channel array index argument.

```
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.

A.8 Simple RSL Specifications

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, one or more values (including functions), zero, one or more variables, zero, one or more channels and zero, one or more axioms listed under respective **type, variable, channel, value**and **axiom** "headers". We prefer to list their order as shown:

Formal Expressions

```
type
  ...
variable
  ...
channel
  ...
value
  ...
axiom
  ...
```

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[3]

[3] For schemes, classes and objects we refer to Vol. 2, Chap. 10 of this series of textbooks.

B

Glossary

The glossary of Appendix B in Vol. 1 listed 788 terms. Here we define a few additional terms.

$\mathcal{B}$

1. **Back end:** By "back end" (software), using a kind of slang or jargon, we either mean software that "sits close" to the hardware, i.e., the computer, or we mean that part of a compiler which is target machine dependent and typically generates code for specific target computer hardware (or for an abstract code interpreter). (See front end.)

$\mathcal{F}$

2. **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].)
3. **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.)

4. **Front end:** By "front end" (software), using a kind of slang or jargon, we either mean software that "sits close" to the user, or we mean that part of a compiler which is target machine independent and hence analyses programs being compiled for syntactic well-formedness and other programming language or specific program properties. (See back end.)

$\mathcal{H}$

5. **Hazard:** A hazard is a source of danger.
6. **Hazard analysis:** Hazard analysis is a process used to determine how a device can cause hazards to occur and then reducing the risks to an acceptable level. (The process consists of: (1) the developer of the system determining what could go wrong with the device, (2) determining how the effects of the failure can be mitigated, and (3) implementing and testing mitigations.)

$\mathcal{M}$

7. **Middleware:** By middleware, using a kind of slang or jargon, we mean software that allows "front ends" like web browsers/servers or other end-user software packages to communicate with "back end" base software like database management systems (i.e., databases).

$\mathcal{R}$

8. **Risk:** The Concise Oxford Dictionary [222] 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.

C

Indexes

- The **prerequisite** for studying this appendix is that you need look up where a term has been defined or is used.
- The **aim** is to illustrate the breadth and depth, the variety and multitude of terms used in these volumes.
- The **objective** is to satisfy your needs.
- The **treatment** is systematic and exhaustive!

Volume 1 Appendix B contains an extensive glossary of 788 glosses. Appendix B (of this volume) augments that glossary with eight more glosses.

C.1 Concepts Index

Conceive: To grasp with the mind.

Conception: The act of conceiving, apprehension, imagination.

Concept: The product of the faculty of conception, an idea of a class of objects, a general notion.

The SHORTER OXFORD ENGLISH DICTIONARY On Historical Principles [222]

The terms: a concept, an idea, a notion, an apprehension and an imagination are treated as similar terms. The concept index also lists common abbreviations.

C.2 Characterisations and Definitions Index

Definition: The setting of bounds, limitation.
The action of determining a question at issue, of defining.
A precise statement of the essential nature of a thing.
A declaration of the signification of a word or phrase.

The SHORTER OXFORD ENGLISH DICTIONARY
On Historical Principles [222]

We shall list both characterisations and definitions. The latter are usually more formally expressed than the former.

C.3 Authors Index

Author: The person who originates or gives existence to anything; an inventor, constructor, or founder.

He who gives rise to an action, event, circumstance, or state of things.

One who sets forth written statements; the writer or composer of a treatise or book.

The SHORTER OXFORD ENGLISH DICTIONARY
On Historical Principles [222]

The authors listed here (many with [references] to (usually) their main books) are (co)authors of publications cited on the referenced page(s). Not all referenced publications have their authors listed here — but a very high proportion have been listed here!

References

1. G. Abowd, R. Allen, D. Garlan: *Using Style to Understand Descriptions of Software Architecture.* SIGSOFT Software Engineering Notes **18**, 5 (1993) pp 9–20
2. G. Abowd, R. Allen, D. Garlan: *Formalizing Style to Understand Descriptions of Software Architecture.* ACM Transactions on Software Engineering and Methodology **4**, 4 (1995) pp 319–364
3. J. Abrial: (1) The Specification Language Z: Basic Library, 30 pgs.; (2) The Specification Language Z: Syntax and "Semantics", 29 pp; (3) An Attempt to Use Z for Defining the Semantics of an Elementary Programming Language, 3 pp; (4) A Low Level File Handler Design, 18 pp; (5) Specification of Some Aspects of a Simple Batch Operating System, 37 pp Internal Reports, Programming Research Group, Oxford Univ., UK (1980)
4. J.-R. Abrial: *The B Book: Assigning Programs to Meanings* (Cambridge University Press, UK 1996)
5. J.-R. Abrial, L. Mussat. *Event B Reference Manual (Editor: Thierry Lecomte)*, 2001. Report of EU IST Project Matisse IST-1999-11435.
6. A.V. Aho, R. Sethi, J.D. Ullman: *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Mass., USA 1986)
7. R. Allen, D. Garlan: A Formal Approach to Software Architectures. In: *IFIP Transactions A (Computer Science and Technology); IFIP World Congress; Madrid, Spain*, vol.A-12 (North-Holland, Amsterdam, Netherlands 1992) pp 134–141
8. R. Allen, D. Garlan: Formalizing Architectural Connection. In: *16th International Conference on Software Engineering (Cat. No.94CH3409-0); Sorrento, Italy* (IEEE Comput. Soc. Press, CA, USA 1994) pp 71–80
9. R. Allen, D. Garlan: A Case Study in Architectural Modeling: the AEGIS System. In: *8th International Workshop on Software Specification and Design; Schloss Velen, Germany* (IEEE Comput. Soc. Press, CA, USA 1996) pp 6–15
10. G. Allwein, J. Barwise: *Logical Reasoning with Diagrams* (Oxford University Press, NY, USA 1996)
11. J. Alves-Foss (ed.): *Formal Syntax and Semantics of Java* (Springer, 1998)
12. M. Andersen, R. Elmstrøm, P.B. Lassen, P.G. Larsen: *Making Specifications Executable – Using IPTES Meta-IV.* Microprocessing and Microprogramming **35**, 1-5 (1992) pp 521–528

13. ANSI: *Database Language SQL* (American National Standards Institute, NY, USA 1992)
14. A. Appel: *Compiler Construction Using Java* (Addison-Wesley, 1999)
15. K.R. Apt: *Principles of Constraint Programming* (Cambridge University Press, August 2003)
16. K. Araki, A. Galloway, K. Taguchi (eds.): *IFM 99: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
17. K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language* (Addison-Wesley, US 1996)
18. K. Åström, B. Wittenmark: *Adaptive Control* (Addison-Wesley, 1989)
19. R.-J. Back, J. von Wright: *Refinement Calculus: A Systematic Introduction* (Springer, Heidelberg, Germany 1998)
20. F. Bauer, H. Wössner: *Algorithmic Language and Program Development* (Springer, 1982)
21. K. Beck: *Test-Driven Development: By Example* (Addison-Wesley, 2003)
22. K. Beck: *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 1999)
23. K. Beck, M. Fowler: *Planning Extreme Programming* (Addison-Wesley, 2000)
24. C. Bell: The Aesthetic Hypothesis. In: *The Philosophy of Art: Readings Ancient and Modern* (McGraw-Hill, 1995)
25. A. Benveniste, M.L. Borgne, P.L. Guernic. *SIGNAL as a Model for Real-Time and Hybrid Systems*, 20–38. Lecture Notes in Computer Science, Springer, 1992.
26. B. Berard, M. Bidoit, A. Finkel et al.: *Systems and Software Verification* (Springer, Heidelberg, Germany 2001)
27. G. Berry, G. Gonthier: *The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation.* Science of Computer Programming **19**, 2 (1992) pp 83–152
28. M. Bidoit, P.D. Mosses: CASL *User Manual* (Springer, 2004)
29. J. Billingsley: *Controlling with Computers: Control Theory and Practical Digital Systems* (McGraw-Hill, 1989)
30. G. Birtwistle, O.-J. Dahl, B. Myhrhaug, K. Nygaard: *SIMULA* `begin` (Studentlitteratur, Sweden, 1974)
31. P.G. Bishop (ed.): *Dependability of Critical Computer Systems.* Vol. 3, Elsevier Applied Science (1988)
32. D. Bjørner: Programming in the Meta-Language: A Tutorial. In: *The Vienna Development Method: The Meta-Language, [35]*, ed by D. Bjørner, C.B. Jones (Springer, 1978) pp 24–217
33. D. Bjørner: Realization of Database Management Systems. In: *See [36]* (Prentice Hall, 1982) pp 443–456
34. D. Bjørner: The Grand Challenge – FAQs of the R&D of a Railway Domain Theory. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic, Amsterdam, The Netherlands 2004)
35. D. Bjørner, C. Jones (eds.): *The Vienna Development Method: The Meta-Language*, vol 61 of *LNCS* (Springer, 1978)
36. D. Bjørner, C. Jones (eds.): *Formal Specification and Software Development* (Prentice Hall, 1982)

37. D. Bjørner, C. Jones, M. Mac an Airchinnigh, E. Neuhold (eds.): *VDM — A Formal Method at Work.* Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer, Lecture Notes in Computer Science, Vol. 252, March 1987.
38. D. Bjørner, H.H. Løvengreen: Formal Semantics of Databases. In: *8th Int'l. Very Large Data Base Conf.* (1982)
39. D. Bjørner, H.H. Løvengreen: Formalization of Data Models. In: *Formal Specification and Software Development, [36]* (Prentice Hall, 1982) pp 379–442
40. D. Bjørner, O. Oest: *The DDC Ada Compiler Development Project.* [41] (1980) pp 1–19
41. D. Bjørner, O. Oest (eds.): *Towards a Formal Description of Ada*, vol 98 of *LNCS* (Springer, 1980)
42. B. Boehm: *Software Engineering Economics* (Prentice Hall, NJ, USA, 1981)
43. E.A. Boiten, J. Derrick, G. Smith (eds.): *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, UK 2004. Springer. Proceedings of 4th Intl. Conf. on IFM.
44. G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide* (Addison-Wesley, 1998)
45. J. Bowen, M. Hinchey: Seven More Myths of Formal Methods. IEEE Software, **7**, 5 (1990) pp 34–41
46. J. Bowen, M. Hinchey: Ten Commandments of Formal Methods. IEEE Computer (April 1995) pp 56–63
47. J. Bowen, M. Hinchey: Ten Commandments of Formal Methods ... Ten Years Later IEEE Computer, pp 58–66 (January 2006).
48. A. Brogi, J.-M. Jacquet: Modelling Coordination via Asynchronous Communication. In: *Proceedings of the Second International Conference on Coordination Languages and Models, Eds.: D. Garlan and D. Le Métayer*, vol 1282 of *Lecture Notes in Computer Science* (Springer, 1997) pp 238–255
49. A. Brogi, J.-M. Jacquet, A. Linden: *On Modelling Coordination via Asynchronous Communication and Enhanced Matching.* Electronic Notes in Theoretical Computer Science **68**, 3 (2002)
50. S.D. Brown, R.J. Francis, J. Rose: *Field-Programmable Gate Arrays* (Kluwer Academic, April 10, 2003)
51. Bureau Veritas. The Bureau Veritas Home Page. Electronically, on the Web: `http://www.bureauveritas.com/homepage_frameset.html`, 2005.
52. M.J. Butler, L. Petre, K. Sere (eds.): *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Springer.
53. X. Cao: *A Comparison of the Dynamics of Continuous and Discrete Event Systems.* Proc. of the IEEE **77**, 1 (1989) pp 7–13
54. X. Cao, Y. Ho: *Models of Discrete Event Dynamic Systems.* IEEE Control System Magazine **10**, 4 (1990) pp 69–76
55. Carnegie Mellon University Model Checking Group home page. Electronically, on the Web: `http://www-2.cs.cmu.edu/~modelcheck/`, 2004.
56. C. Cassandras, P. Ramadge: *Toward a Control Theory for Discrete Event Systems.* IEEE Control System Magazine **10**, 4 (1990) pp 66–68
57. C. Cassandras, S. Strickland: *Sample Path Properties of Timed Discrete Event Systems.* Proc. of the IEEE **77**, 1 (1989) pp 59–71
58. A.M.K. Cheng: *Real-Time Systems Scheduling, Analysis, and Verification* (Wiley, NJ, USA 2002)
59. E.M. Clarke, O. Grumberg, D.A. Peled: *Model Checking* (MIT Press, MA, USA 2000)

60. G. Clemmensen, O. Oest: Formal Specification and Development of an Ada Compiler – A VDM Case Study. In: *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida* (IEEE Press, 1984) pp 430–440
61. E.F. Codd: *A Relational Model for Large Shared Databanks.* Communications of the ACM **13**, 6 (1970) pp 377–387
62. CoFI (The Common Framework Initiative): CASL *Reference Manual*, vol 2960 of *Lecture Notes in Computer Science (IFIP Series)* (Springer, 2004)
63. P. Cousot: Semantic Foundation of Program Analysis. In: *Program Flow Analysis: Theory and Applications*, ed by S. Muchnick, N. Jones (Prentice Hall, 1981) pp 303–342
64. P. Cousot: *Abstract Interpretation.* ACM Computing Surveys **28**, 2 (1996) pp 324–328
65. P. Cousot: *Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract).* Theoretical Computer Science **6** (1997) p 25
66. P. Cousot, R. Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *4th POPL: Principles of Programming and Languages* (ACM Press, 1977) pp 238–252
67. P. Cousot, R. Cousot: Systematic Design of Program Analysis Frameworks. In: *6th POPL: Principles of Programming and Languages* (ACM Press, 1979) pp 269–282
68. P. Cousot, R. Cousot: Induction Principles for Proving Invariance Properties of Programs. In: *Tools & Notions for Program Construction, Ed. D. Néel* (Cambridge University Press, 1982) pp 43–119
69. P. Cousot, R. Cousot: Inductive Definitions, Semantics and Abstract Interpretation. In: *19th POPL: Principles of Programming and Languages* (ACM Press, 1992) pp 83–94
70. P. Cousot, R. Cousot: Higher-Order Abstract Interpretation (and application to comportment analysis generalising strictness, termination, projection and PER analysis of functional languages). In: *1994 ICCL* (IEEE Comp. Sci. Press, 1994) pp 95–112
71. P. Cousot, R. Cousot: Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In: *7th FPCA* (ACM Press, 1995) pp 170–181
72. CVS. Concurrent Versions System Home Page. Electronically, on the Web: `www.cvshome.org`, 2005.
73. W. Damm, D. Harel: *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design **19** (2001) pp 45–80
74. J. Dancy, E. Sosa (eds.): *The Blackwell Companion to Epistemology* (Blackwell, Oxford, UK 1994)
75. H. Dang Van, C. George, T. Janowski, R. Moore (eds.): *Specification Case Studies in RAISE* (Springer, 2002)
76. A. Dardenne, S. Fikas, A. van Lamsweerde: Goal-Directed Concept Acquisition in Requirements Elicitation. In: *Proc. IWSSD-6, 6th Intl. Workshop on Software Specification and Design* (IEEE Computer Society Press, USA 1991) pp 14–21
77. A. Dardenne, A. van Lamsweerde, S. Fikas: *Goal-Directed Requirements Acquisition.* Science of Computer Programming **20** (1993) pp 3–50

78. R. Darimont, A. van Lamsweerde: Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In: *Proc. FSE'4, Fourth ACM SIGSOFT Symp. on the Foundations of Software Enginering* (ACM, 1996) pp 179–190
79. Data base task group (DBTG), CODASYL report. Assoc. for Computing Machinery, NY, USA, 1971.
80. C. Date: *An Introduction to Database Systems, I* (Addison-Wesley, 1981)
81. C. Date: *An Introduction to Database Systems, II* (Addison-Wesley, 1983)
82. C. Date, H. Darwen: *A Guide to the SQL Standard* (Addison-Wesley, 1996)
83. J. Davies. Announcement: Electronic Version of Communicating Sequential Processes (CSP). Published electronically: `http://www.usingcsp.com/`, 2004. Announcing revised edition of [168].
84. B. Denvir: Enriching VDM with CCS: a Study. Technical Report, Standard Telecommunication Laboratories Ltd. (1985)
85. R. Diaconescu, K. Futatsugi, K. Ogata: *CafeOBJ: Logical Foundations and Methodology.* Computing and Informatics **22**, 1–2 (2003)
86. E. Dijkstra: *A Discipline of Programming* (Prentice Hall, 1976)
87. E. Dijkstra, W. Feijen: *A Method of Programming* (Addison-Wesley, 1988)
88. E. Dijkstra, C. Scholten: *Predicate Calculus and Program Semantics* (Springer: Texts and Monographs in Computer Science, 1990)
89. R. Dorf: *Modern Control Systems* (Addison-Wesley Publishing Company, 1967 (fifth ed. 1989))
90. J. Dugan, S. Bavuso, M. Boyd: *Fault Trees and Markov Models for Reliability Analysis of Fault-Tolerant Digital Systems.* In: Reliability Engineering and System Safety, **39**:291–307 (1993)
91. R.W. Durmiendo, C.W. George: Development of a Distributed Telephone Switch. In: *[75]* (Springer, April 2002) pp 99–130
92. E.H. Dürr, J. van Katwijk: VDM^{++} – A Formal Specification Language for Object-Oriented Designs. In: *Technology of Object-Oriented Languages and Systems*, (Prentice Hall, 1992) pp 63–78
93. E.H. Dürr, J. van Katwijk: VDM++, A Formal Specification Language for Object-Oriented Designs. In: *COMP EURO 92* (IEEE, 1992) pp 214–219
94. E.H. Dürr, W. Lourens, J. van Katwijk: The Use of the Formal Specification Language VDM^{++} for Data Acquisition Systems. In: *New Computing Techniques in Physics Research II*, ed by D. Perret-Gallix (World Scientific, Singapore 1992) pp 47–52
95. Encyclopædia Brittanica. Encyclopædia Brittanica. Merriam-Webster/Brittanica: Access over the Web: http://www.eb.com:180/, 1999.
96. E. Engeler: *Symposium on Semantics of Algorithmic Languages*, vol 188 of *Lecture Notes in Mathematics* (Springer, 1971)
97. M. Erdenechimeg, Y. Namstrai, R.C. Moore: Multi-lingual Document Processing. In: *[75]* (Springer, April 2002) pp 155–186
98. R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi: *Reasoning About Knowledge* (MIT Press, Mass., 1996)
99. A. Fantechi: On Combining Meta-IV and CCS. Technical Report, Dept. of Comp. Sci., Techn. Univ. of Denmark (1984)
100. D. Favrholdt: *Filosofisk Codex — Om begrundelsen af den menneskelige erkendelse* (Gyldendal, Denmark 1999)
101. D. Favrholdt: *Æstetik og filosofi* (Høst & Søn, Denmark 2000)

102. M. Feather, S. Fikas, A. van Lamsweerde, C. Ponsard: Reconciling System Requirements and Runtime Behaviours. In: *Proc. IWSSD'98, 9th Intl. Workshop on Software Specification and Design* (IEEE Computer Society Press, Isobe, Japan 1998)
103. W. Feijen, A. van Gasteren, D. Gries, J. Misra (eds.): *Beauty Is Our Business*, Texts and Monographs in Computer Science, NY, USA, 1990. Springer. A Birthday Salute to Edsger W. Dijkstra.
104. J.S. Fitzgerald, P.G. Larsen: *Developing Software Using VDM-SL* (Cambridge University Press, UK 1997)
105. J.S. Fitzgerald, S. Vadera: Unification: Specification and Development, and: Building a Theory of Unification. In: *[199]* (Prentice Hall, 1990) pp 127–162 and 163–194
106. W. Fleming (ed.): *Report of the Panel on Future Directions in Control Theory: A Mathematical Perspective* (SIAM, 1988)
107. B. Flinn, I.H. Sørensen: CAVIAR: a case study in specification. In: *[156]* (Prentice Hall, 1987) pp 79–110
108. A. Fraenkel, Y. Bar-Hillel, A. Levy: *Foundations of Set Theory*, 2nd revised edn (Elsevier Science, The Netherlands, 1973)
109. G. Franklin, J. Powell, M. Workman: *Digital Control of Dynamic Systems* (Addison-Wesley, 1990)
110. K. Futatsugi, R. Diaconescu: *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification* (World Scientific, Singapore 1998)
111. K. Futatsugi, A. Nakagawa, T. Tamai (eds.): *CAFE: An Industrial-Strength Algebraic Formal Method*, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.
112. B. Ganter, R. Wille: *Formal Concept Analysis — Mathematical Foundations* (Springer, January 1999)
113. D. Garlan: *Research Directions in Software Architecture.* ACM Computing Surveys **27**, 2 (1995) pp 257–261
114. D. Garlan: Formal Approaches to Software Architecture. In: *Studies of Software Design. ICSE '93 Workshop. Selected Papers* (Springer, Berlin, Germany 1996) pp 64–76
115. D. Garlan, M. Shaw: Experience with a Course on Architectures for Software Systems. In: *Software Engineering Education. SEI Conference 1992; San Diego, CA, USA* (Springer, Berlin, Germany 199) pp 23–43
116. D. Garlan, M. Shaw. *An Introduction to Software Architecture*, pages 1–39. World Scientific, Singapore, 1993.
117. C.W. George, P. Haff, K. Havelund et al.: *The RAISE Specification Language* (Prentice Hall, UK 1992)
118. C.W. George, A.E. Haxthausen, S. Hughes et al.: *The RAISE Method* (Prentice Hall, UK 1995)
119. C.W. George, M.I. Wolczko: Heap Storage, and Garbage Collection. In: *[199]* (Prentice Hall, 1990) pp 195–233
120. C.W. George, Y. Xia: An Operational Semantics for Timed RAISE. In: *FM'99 — Formal Methods*, ed by J.M. Wing, J. Woodcock, J. Davies (Springer, 1999) pp 1008–1027
121. C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering* (Prentice Hall, 2002)

122. T.C. Giras: A Stochastic Framework for Train Domain Theories. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic, Amsterdam, The Netherlands 2004)
123. J.A. Goguen, M. Girotka (eds.): *Requirements Engineering: Social and Technical Issues* (Academic Press, 1994)
124. J.A. Goguen, C. Linde: Techniques for Requirements Elicitation. In: *Proc. RE'93, First IEEE Symposium on Requirements Engineering* (IEEE Computer Society Press, CA, USA 1993) pp 152–164
125. J. Gosling, F. Yellin: *The Java Language Specification* (ACM Press Books, 1996)
126. S.J. Greenspan, J. Mylopoulos, A. Borgida: Capturing More World-Knowledge in Requirements Specification. In: *Proc. 6th ICSE: Intl. Conf. on Software Engineering* (IEEE Computer Society Press, Tokyo, Japan 1982)
127. S.J. Greenspan, J. Mylopoulos, A. Borgida: *A Requirements Modelling Language.* Information Systems **11**, 1 (1986) pp 9–23
128. J.-C. Grégoire, G.J. Holzmann, D. Peled (eds.): *The SPIN Verification System*, volume 32 of *DIMACS Series.* American Mathematical Society, 1997. ISBN 0-8218-0680-7, 203 p.
129. D. Gries: *Compiler Construction for Digital Computers* (Wiley, NY, 1971)
130. D. Gries: *The Science of Programming* (Springer, 1981)
131. D. Gries, F.B. Schneider: *A Logical Approach to Discrete Math* (Springer, 1993)
132. W. Grieskamp, T. Santen, B. Stoddart (eds.): *IFM 2000: Integrated Formal Methods*, volume 1945 *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1-3, 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
133. C. Gunther, D. Scott: Semantic Domains. In: *[368] — vol. B.*, ed by J. van Leeuwen (North-Holland, Amsterdam, 1990) pp 633–674
134. P. Haff (ed.): *The Formal Definition of CHILL* (ITU (International Telecommunications Union), Geneva, Switzerland 1981)
135. P. Haff, A. Olsen: Use of VDM Within CCITT. In: *[37]* (Springer, 1987) pp 324–330
136. N. Halbwachs: *Synchronous Programming of Reactive Systems* (Kluwer Academic, 1993)
137. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud: *The Synchronous Dataflow Programming Language Lustre.* Proceedings of the IEEE **79**, 9 (1991) pp 1305–1320
138. A. Hall: *Seven Myths of Formal Methods.* IEEE Software **7**, 5 (1990) pp 11–19
139. M. Hammer, J.A. Champy: *Reengineering the Corporation: A Manifesto for Business Revolution* (HarperCollins, UK May 1993)
140. M. Hammer, S.A. Stanton: *The Reengineering Revolution: The Handbook* (HarperCollins, UK 1996)
141. K.M. Hansen: Linking Safety Analysis to Safety Requirements. PhD Thesis, Department of Computer Science, Technical University of Denmark (1996)
142. M.R. Hansen, H. Rischel: *Functional Programming in Standard ML* (Addison-Wesley, 1997)
143. D. Harel: *Algorithmics — The Spirit of Computing* (Addison-Wesley, 1987)
144. D. Harel: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming **8**, 3 (1987) pp 231–274
145. D. Harel: *On Visual Formalisms.* Communications of the ACM **33**, 5 (1988)

146. D. Harel: *The Science of Computing — Exploring the Nature and Power of Algorithms* (Addison-Wesley, 1989)
147. D. Harel, E. Gery: *Executable Object Modeling with Statecharts.* IEEE Computer **30**, 7 (1997) pp 31–42
148. D. Harel, H. Lachover, A. Naamad et al.: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems.* Software Engineering **16**, 4 (1990) pp 403–414
149. D. Harel, R. Marelly: *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine* (Springer, 2003)
150. D. Harel, A. Naamad: *The STATEMATE Semantics of Statecharts.* ACM Transactions on Software Engineering and Methodology (TOSEM) **5**, 4 (1996) pp 293–333
151. A.E. Haxthausen, T. Gjaldbæk: Modelling and Verification of Interlocking Systems for Railway Lines. In: *10th IFAC Symposium on Control in Transportation Systems, Tokyo, Japan* (2003)
152. A.E. Haxthausen, J. Peleska: Formal Development and Verification of a Distributed Railway Control System. In: *Proceedings of Formal Methods World Congress FM'99*, Vol. 1709 of *Lecture Notes in Computer Science* (Springer, 1999) pp 1546 – 1563
153. A.E. Haxthausen, J. Peleska: *Formal Development and Verification of a Distributed Railway Control System.* IEEE Transaction on Software Engineering **26**, 8 (2000) pp 687–701
154. A.E. Haxthausen, J. Peleska: A Domain Specific Language for Railway Control Systems. In: *Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California* (Society for Design and Process Science, TX, USA 2002)
155. A.E. Haxthausen, X. Yong: Linking DC together with TRSL. In: *Proceedings of 2nd International Conference on Integrated Formal Methods (IFM 2000), Schloss Dagstuhl, Germany, November 2000*, Vol. 1945 of *Lecture Notes in Computer Science* (Springer, 2000) pp 25–44
156. I. Hayes (ed.): *Specification Case Studies* (Prentice Hall, 1987)
157. I. Hayes, S. King: Chapters 13–17 on the formal modelling of the IBM CICS Transaction Processing System. In: *[156]* (Prentice Hall, 1987) pp 179–243
158. E. Hehner: *The Logic of Programming* (Prentice Hall, 1984)
159. E. Hehner: *A Practical Theory of Programming*, 2nd edn (Springer, 1993)
160. M. Heidegger: *Sein und Zeit (Being and Time)* (Oxford University Press, 1927, 1962)
161. A. Hejlsberg, S. Wiltamuth, P. Golde: *The C# Programming Language* (Addison-Wesley, MA, USA 2003)
162. M.C. Henson, S. Reeves, J.P. Bowen: *Z Logic and Its Consequences.* Computing and Informatics **22**, 1–2 (2003)
163. M. Heymann: *Concurrency and Discrete Event Control.* IEEE Control System Magazine **10**, 4 (1990) pp 103–112
164. M.G. Hinchey, J.P. Bowen (eds.): *Applications of Formal Methods* (Prentice Hall, 1995)
165. M.G. Hinchey, J.P. Bowen (eds.): *Industrial-Strength Formal Methods in Practice* (Springer, 1999)
166. Y. Ho: *Dynamics of Discrete Event Systems.* Proc. of the IEEE **77**, 1 (1989) pp 3–6

167. Y. Ho: *Special Issue on the Dynamics of Discrete Event Systems.* Proc. of the IEEE **77**, 1 (1989)
168. C.A.R. Hoare: *Communicating Sequential Processes* (Prentice Hall, 1985. See also [170])
169. C.A.R. Hoare: *The Verifying Compiler: A Grand Challenge for Computing Research.* Journal of the ACM **50** (2003) pp 63–69
170. C.A.R. Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [168]. See also `http://www.usingcsp.com/`.
171. C.A.R. Hoare, J.F. He: *Unifying Theories of Programming* (Prentice Hall, 1997)
172. G.J. Holzmann: *Design and Validation of Computer Protocols* (Prentice Hall, NJ, 1991)
173. G.J. Holzmann: *The SPIN Model Checker, Primer and Reference Manual* (Addison-Wesley, Mass., 2003)
174. N. Horspool, P. Gorman: *The ASIC Handbook* (Prentice Hall, 2001)
175. W. Humphrey: *Managing the Software Process* (Addison-Wesley, 1989)
176. V.D. Hunt: *Process Mapping: How to Reengineer Your Business Processes* (Wiley, NY, USA 1996)
177. A. Hunter, B. Nuseibeh: *Managing Inconsistent Specifications: Reasoning, Analysis and Action.* ACM Transactions on Software Engineering and Methodology **7**, 4 (1998) pp 335–367
178. IEEE Computer Society: IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical Report, IEEE Press, Washington DC, USA (1990)
179. IEEE: The Institute of Electrical and Electronics Engineers. The IEEE Home Page. Electronically, on the Web: `http://www.ieee.org`, 2005.
180. ISO: *Information Technology — Database Languages — SQL* (American National Standards Institute, NY, USA 1992)
181. ISO: The International Organisation for Standardization. The ISO Home Page. Electronically, on the Web: `http://www.iso.org`, 2005.
182. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
183. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
184. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
185. ITU: The International Telecommunication Union. The ITU Home Page. Electronically, on the Web: `http://www.itu.org`, 2005.
186. J.M. Jacka, P.J. Keller: *Business Process Mapping: Improving Customer Satisfaction* (Wiley, NY, USA 2002)
187. M.A. Jackson: *Principles of Program Design* (Academic Press, 1969)
188. M.A. Jackson: *System Design* (Prentice Hall, 1985)
189. M.A. Jackson: *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices* (Addison-Wesley, UK 1995)
190. M.A. Jackson: *Software Hakubutsushi: Sekai to Kikai no Kijutsu (Software Requirements & Specifications: a lexicon of practice, principles and prejudices)* (Toppan, Japan 1997)
191. M.A. Jackson: *Problem Frames — Analyzing and Structuring Software Development Problems* (Addison-Wesley, UK 2001)
192. M.A. Jackson, G. Twaddle: *Business Process Implementation — Building Workflow Systems* (Addison-Wesley, UK 1997)

193. I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process* (Addison-Wesley, 1999)
194. J.-M. Jacquet, A. Linden: On Methodologies for Coordinating Programs. In: *Proceedings of the 18th ACM Symposium of Applied Computing, Florida, USA* (ACM Press, 2003) pp 115–121
195. A. Jantsch: *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation* (Morgan Kaufmann, June 2003)
196. K. Jensen: *Coloured Petri Nets*, vol 1: Basic Concepts (234 pages), Vol. 2: Analysis Methods (174 pages), Vol. 3: Practical Use (265 pages) of *EATCS Monographs in Theoretical Computer Science* (Springer, Heidelberg 1997)
197. C.B. Jones: *Systematic Software Development Using VDM* (Prentice Hall, 1986)
198. C.B. Jones: *Systematic Software Development Using VDM*, 2nd edn (Prentice Hall, 1990)
199. C.B. Jones, R.C. Shaw: *Case Studies in Systematic SoFtware Development* (Prentice Hall, 1990)
200. N.D. Jones: *Computability and Complexity — From a Programming Point of View* (MIT Press, Mass., USA, 1996)
201. N.D. Jones, C. Gomard, P. Sestoft: *Partial Evaluation and Automatic Program Generation* (Prentice Hall, 1993)
202. M.H. Kay: XML five years on: a review of the achievements so far and the challenges ahead. In: *Proceedings of the 2003 ACM Symposium on Document Engineering, [367]* (2003) pp 29 – 31
203. J. Klose, H. Wittke: An Automata Based Interpretation of Live Sequence Charts. In: *TACAS 2001*, ed by T. Margaria, W. Yi (Springer, 2001) pp 512–527
204. D. Knuth: *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (Addison-Wesley, Mass., USA, 1968)
205. D. Knuth: *The Art of Computer Programming, Vol. 2.: Seminumerical Algorithms* (Addison-Wesley, Mass., USA, 1969)
206. D. Knuth: *The Art of Computer Programming, Vol. 3: Searching & Sorting* (Addison-Wesley, Mass., USA, 1973)
207. R. Kurki-Suonio: *A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors* (Springer, Germany, April 2005)
208. I. Lakatos: *Proofs and Refutations: The Logic of Mathematical Discovery (Eds.: J. Worrall and E.G. Zahar)* (Cambridge University Press, UK 1976)
209. L. Lamport: *The Temporal Logic of Actions.* Transactions on Programming Languages and Systems **16**, 3 (1995) pp 872–923
210. L. Lamport: *Specifying Systems* (Addison-Wesley, Mass., USA 2002)
211. R.D. Landtsheer, E. Letier, A. van Lamsweerde: Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models. In: *RE'03, 11th IEEE Joint International Requirements Engineering Conference* (IEEE CS Press, USA 2003) pp 200–210
212. J. Laprie (ed.): *Dependability: Basic Concepts and Terminology*, vol 5 of *Dependable Computing and Fault-Tolerant Systems* (Springer, Vienna 1992)
213. P.A. Lee, T. Anderson: *Fault Tolerance, Principles and Practice* (Springer 1990)
214. E. Letier, A. van Lamsweerde: Agent-Based Tactics for Goal-Oriented Requirements Elaboration. In: *Proceedings ICSE 2002 - 24th International Conference on Software Engineering* (IEEE CS Press, USA 2002)

215. E. Letier, A. van Lamsweerde: Deriving Operational Software Specifications from System Goals. In: *Proceedings FSE'10 - 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering* (ACM, NC, USA 2002)
216. S. Levi, A. Agrawala: *Real-Time System Design* (McGraw-Hill, NY, USA 1990)
217. Y. Li, W. Wonham: *On Supervisory Control of Real-Time Discrete Event Systems.* Information Sciences **46**, 3 (1988) pp 159–183
218. T.M. Lien, L.L. Chi, P.P. Nam et al.: Developing a National Financial Information System. In: *[75]* (Springer, 2002)
219. M.P. Lindegaard, P. Viuf, A.E. Haxthausen: Modelling Railway Interlocking Systems. In: *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13–15, 2000, Braunschweig, Germany* (2000) pp 211–217
220. J. Lindenau: Eine Deskriptive Anfragesprache für das Netzwerk-Datenmodell mit formaler Definition der Semantik in Meta-IV. MSc Thesis, Inst. f. Informatik, Christian-Albrechts-Univ., Kiel (1981) pp 1–175
221. T. Lindholm, F. Yellin: *The Java Virtual Machine Specification* (ACM Press, 1996)
222. W. Little, H. Fowler, J. Coulson, C. Onions: *The Shorter Oxford English Dictionary on Historical Principles* (Clarendon Press, Oxford, UK, 1987)
223. Z. Liu, A.P. Ravn, E.V. Søresen, C.C. Zhou: *A Probabilistic Duration Calclus.* in H. Kopetz, Y. Kakuda (eds.) Responsive Computer Systems, Vol. 7 of *Dependable Computing and FauLt-Tolerant Systems,* pp 29–52, Springer (1993)
224. Lloyd's Register. The Lloyd's Register Home Page. Electronically, on the Web: `http://www.lr.org/code/home.htm`, 2005.
225. D. Luenberger: *Introduction to Dynamic Systems Theory: Theory, Models & Applications* (Wiley, 1979)
226. M.R. Lyu: *Software Fault Tolerance* (Chichester, UK, 1995)
227. Z. Manna: *Mathematical Theory of Computation* (McGraw-Hill, 1974)
228. Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems: Specification* (Springer, NY, USA 1992)
229. Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems: Safety* (Springer, NY, USA 1995)
230. Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems: Progress* (Unpublished, Stanford University, Computer Science Department, http://theory.stanford.edu/~zm/tvors3.html 2004)
231. Z. Manna, R. Waldinger: *The Logical Basis for Computer Programming, Vols. 1–2* (Addison-Wesley, 1985–90)
232. L.C. Marshall: Line Representation on Graphics Devices. In: *[199]* (Prentice Hall, 1990) pp 337–364
233. D.J. Mayhew: *Principles and Guidelines in Software User Interface Design* (Prentice Hall 1992)
234. J.A. McDermid, D.J. Pumfrey: *A Development of Hazard Analysis to Aid Software Design.* In: COMPASS '94: Proceedings of the Ninth Annual IEEE Conference on Computer Assurance (1994) pp 17–25
235. J.A. McDermid, D.J. Pumfrey: *Software Safety: Why Is There No Consensus?* In: Proceedings of the 19th International System Safety Conference, System Safety Society (2001)

236. A.A. McEwan, J. Woodcock: The need for integrated formal methods in specifying models of railways. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic, Amsterdam, The Netherlands 2004)
237. D.H. Mellor, A. Oliver: *Properties* (Oxford University Press, May 1997)
238. Merriam–Webster. Online Dictionary: `http://www.m-w.com/home.htm`, 2004.
239. S. Merz: *On the Logic of TLA+.* Computing and Informatics **22**, 1–2 (2003)
240. U. Meyer-Baese: *Digital Signal Processing with Field Programmable Gate Arrays* (Springer, Berlin Heidelberg, Germany 2001)
241. 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)
242. 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)
243. R. Milne, C. Strachey: *A Theory of Programming Language Semantics* (Chapman and Hall, London and Wiley, New York 1976)
244. R. Milner: *Calculus of Communication Systems*, vol 94 of *Lecture Notes in Computer Science* (Springer, 1980)
245. R. Milner: *Communication and Concurrency* (Prentice Hall, 1989)
246. R. Milner: *Communicating and Mobile Systems: The π-Calculus* (Cambridge University Press, 1999)
247. R. Milner, M. Tofte, R. Harper: *The Definition of Standard ML* (MIT Press, Cambridge, Mass., USA and London, UK, 1990)
248. R.C. Moore: Muffin: A Proof Assistant. In: *[199]* (Prentice Hall, 1990) pp 91–126
249. C.C. Morgan: *Programming from Specifications* (Prentice Hall, UK 1990)
250. C.C. Morgan, B. Suffrin: Specification of the UNIX filing system. In: *[156]* (Prentice Hall, 1987) pp 45–78
251. T. Mossakowski, A.E. Haxthausen, D. Sanella, A. Tarlecki: *CASL — The Common Algebraic Specification Language: Semantics and Proof Theory.* Computing and Informatics **22**, 1–2 (2003)
252. P.D. Mosses: *Action Semantics* (Cambridge University Press: Tracts in Theoretical Computer Science, 1992)
253. J. Mylopoulos: *Information Modelling in the Time of Revolution.* Information Systems **23**, 3/4 (1998) pp 127–155
254. J. Mylopoulos, L. Chung, B. Nixon: *Representing and Using Non-functional Requirements: A Process-Oriented Approach.* IEEE Trans. on Software Engineering **18**, 6 (1992) pp 483–497
255. J. Mylopoulos, L. Chung, E. Yu: *From Object-Oriented to Goal-Oriented Requirements Analysis.* CACM: Communications of the ACM **42**, 1 (1999) pp 31–37
256. P. Naur: *The Design of the GIER Algol Compiler.* BIT, Nordisk Tidsskrift for Informations Behandling **3**, 2 (1963)
257. P. Naur, B. Randall (eds.): *Software Engineering: The Garmisch Conference.* NATO Science Committee, Brussels, 1969.
258. A. Neal, M. Humphreys, D. Leadbetter, P. Lindsay: *Development of Hazard Analysis Techniques for Human-Computer Systems.* In: Innovation and Consolidation in Aviation, Ashgate Publ., eds G. Edkins, P. Pfister (2003) pp 255–262

259. F. Nekoogar, F. Nekoogar: *From ASICs to SOCs: A Practical Approach* (Prentice Hall, 2003)
260. Q.T. Ngo, H.D. Van: Formalsiation of Realm-Based Spatial Data Types. In: *[75]* (Springer, 2002) pp 259–286
261. T. Nipkow, L.C. Paulson, M. Wenzel: *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, vol 2283 of *LNCS* (Springer, Heidelberg, Germany, 2002)
262. Norske Veritas. The DNV (Det Norske Veritas) Home Page. Electronically, on the Web: `http://www.dnv.com/`, 2005.
263. B. Nuseibeh, J. Kramer, A. Finkelstein: *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications.* IEEE Transactions on Software Engineering **20**, 10 (1994) pp 760–773
264. Object Management Group: *OMG Unified Modeling Language Specification*, version 1.5 edn (OMG/UML, http://www.omg.org/uml/ 2003)
265. T. Ogino: CyberRail and the TRain R&D. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic, Amsterdam, The Netherlands 2004)
266. A. Ojo, T. Janowski: Formalising Production Processes. In: *[75]* (Springer, 2002) pp 187–217
267. P.J. Ok, R.H. Sul, C.W. George: A University Library System. In: *[75]* (Springer, 2002) pp 81–98
268. S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, CA, 1999.
269. S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, CA, 1999.
270. D.L. Parnas: *Software Fundamentals: Collected Papers, Eds.: David M. Weiss and Daniel M. Hoffmann* (Addison-Wesley, 2001)
271. L. Paulson: *Logic and Computation: Interactive Proof with Cambridge LCF* (Cambridge University Press, 1987)
272. M. Pěnička: From railway resource planning to train operation. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic, Amsterdam, The Netherlands 2004)
273. C.A. Petri: *Kommunikation mit Automaten* (Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962)
274. C. Petzold: *Programming Windows with C# (Core Reference)* (Microsoft Corporation, Redmond, WA, USA 2001)
275. S.L. Pfleeger: *Software Engineering, Theory and Practice*, 2nd edn (Prentice Hall, 2001)
276. P&O Nedlloyd. A–Z Shipping Terms. Electronically, on the Web: `http://-www.ponl.com/topic/home_page/language_en/about_us/useful_information/az_of_shipping_terms`, 2004.
277. K.R. Popper: *Logik der Forschung* (Julius Springer, Vienna, Austria 1934 (1935))
278. K.R. Popper: *The Logic of Scientific Discovery* (Routledge, UK 1982)
279. K.R. Popper: *Conjectures and Refutations. The Growth of Scientific Knowledge* (Routledge. London, UK 1981)
280. K.R. Popper: *Autobiography of Karl Popper* (Open Court, IL, USA 1976)
281. K.R. Popper: *Unended Quest: An Intellectual Autobiography* (Fontana/Collins, England 1976–1982)
282. K.R. Popper: *A Pocket Popper* (Fontana, England 1983)

283. K.R. Popper: *The Myth of the Framework. In Defence of Science and Rationality* (Routledge, UK 1996)
284. R.S. Pressman: *Software Engineering, A Practitioner's Approach*, 5th edn (McGraw-Hill, 2001)
285. A. Ralston, P. Rabinowitz: *A First Course in Numerical Analysis* (Dover, 2nd rev edition, 2001)
286. P. Ramadge, W. Wonham: *The Control of Discrete Event Systems.* Proc. of the IEEE **77**, 1 (1989) pp 81–98
287. B. Randell: On Failures and Faults. In: *FME 2003: Formal Methods*, vol 2805 of *Lecture Notes in Computer Science* (Springer, 2003) pp 18–39
288. B. Randell, L. Russell: *ALGOL 60 Implementation, The Translation and Use of* ALGOL 60 *Programs on a Computer* (Academic Press, 1964)
289. A. Ravn, H. Rischel, K.M. Hansen: *Specifying and Verifying Requirements of Real-Time Systems.* IEEE Trans. Software Engineering **19** (1992) pp 41–55
290. A. Ravn, H. Rischel, E. Sørensen: Control Program for a Gas Burner: Requirements, ProCoS Case Study 0. Technical Report, Dept. of Computer Science, Technical University of Denmark (1989)
291. E.T. Ray: *Learning XML, Guide to Creating Self-describing Data* (O'Reilly, UK, January 2001)
292. W. Reif: Integrated formal methods for safety analysis. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic Press, Amsterdam, The Netherlands 2004)
293. W. Reisig: *Petri Nets: An Introduction*, vol 4 of *EATCS Monographs in Theoretical Computer Science* (Springer, 1985)
294. W. Reisig: *A Primer in Petri Net Design* (Springer, 1992)
295. W. Reisig: *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets* (Springer, 1998)
296. J.C. Reynolds: *The Craft of Programming* (Prentice Hall, 1981)
297. J.C. Reynolds: *Theories of Programming Languages* (Cambridge University Press, UK 1998)
298. J.C. Reynolds: *The Semantics of Programming Languages* (Cambridge University Press, 1999)
299. J.M.T Romijn, G.P. Smith, J.C. van de Pol (eds.): *IFM 2004: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, London, UK 2005. Springer. Proceedings of 5th Intl. Conf. on IFM.
300. A.W. Roscoe (ed.): *A Classical Mind: Essays in Honour of C.A.R. Hoare* (Prentice Hall, 1994)
301. A.W. Roscoe: *Theory and Practice of Concurrency* (Prentice Hall, 1997)
302. C. Rowen, S. Leibson: *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors* (Prentice Hall, 2004)
303. J. Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual* (Addison-Wesley, 1998)
304. J. Rushby: Formal Methods and the Certification of Critical Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, CA, USA (1993)
305. J. Rushby: Formal Methods and Their Role in the Certification of Critical Systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, CA (1995)

306. D. Sabatier: Domain oriented formal models in industry. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic Press, Amsterdam, The Netherlands 2004)
307. K.B. Sall: *XML Family of Specifications* (Pearson/Addison-Wesley 2002)
308. U. Schmidt: Ein neuartiger, auf VDM basierender Codegenerator-Generator. PhD Thesis, Christian-Albrechts-Universität, Kiel, Germany (1983)
309. U. Schmidt, H.-M. Hörcher: Programming with VDM Domains. In: *VDM '90 VDM and Z — Formal Methods in Software Development*, ed by D. Bjørner, C.A.R. Hoare, H. Langmaack (Springer, 1990) pp 122–134
310. U. Schmidt, H.-M. Hörcher: The VDM Domain Compiler a VDM Class Library Generator. In: *VDM '91: Formal Software Development Methods* (Springer, 1991) pp 675–676
311. S. Schneider: *Concurrent and Real-Time Systems — The CSP Approach* (Wiley, UK 2000)
312. B. Shneiderman: *Designing the User Interface (2nd ed.): Strategies for Effective Human-Computer Interaction*, 3rd ed. (Addison-Wesley, Mass., USA 1997)
313. E. Schnieder: TRain: transportation engineering meets computing science. In: *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain* (Kluwer Academic, Amsterdam, The Netherlands 2004)
314. J. Schwartz: The SETL Language and Examples of Its Use. Technical Report, Courant Institute of Mathematics, New York University, USA (1973)
315. J. Schwartz: *Programming with Sets: An Introduction to SETL* (Springer, NY, 1986)
316. D. Scott: The Lattice of Flow Diagrams. In: *[96]* (1970) pp 311–366
317. D. Scott: Outline of a Mathematical Theory of Computation. In: *Proc. 4th Ann. Princeton Conf. on Inf. Sci. and Sys.* (1970) p 169
318. D. Scott: Continuous Lattices. In: *Toposes, Algebraic Geometry and Logic*, ed by F. Lawvere (Springer, Lecture Notes in Mathematics, Vol. 274, 1972) pp 97–136
319. D. Scott: Data Types as Lattices. Unpublished Lecture Notes, Amsterdam (1972)
320. D. Scott: Lattice Theory, Data Types and Semantics. In: *Symp. Formal Semantics*, ed by R. Rustin (Prentice Hall, 1972) pp 67–106
321. D. Scott: Lattice-Theoretic Models for Various Type Free Calculi. In: *Proc. 4th Int'l. Congr. for Logic Methodology and the Philosophy of Science,* Bucharest (North-Holland, Amsterdam, 1973) pp 157–187
322. D. Scott: *Data Types as Lattices.* SIAM Journal on Computer Science **5**, 3 (1976) pp 522–587
323. D. Scott: Domains for Denotational Semantics. In: *International Colloquium on Automata, Languages and Programming, European Association for Theoretical Computer Science* (Springer, 1982) pp 577–613
324. D. Scott: Some Ordered Sets in Computer Science. In: *Ordered Sets*, ed by I. Rival (Reidel, 1982) pp 677–718
325. D. Scott, C. Strachey: Towards a Mathematical Semantics for Computer Languages. In: *Computers and Automata*, vol 21 of *Microwave Research Inst. Symposia* (1971) pp 19–46
326. R. Sengupta, S. Lafortune: A Deterministic Optimal Control Theory for Discrete Event Systems: Formulation and Existence Theory. Technical Report # CGR-93-7, Control Group, College of Engineering, Univ. of Michigan, USA (1993)

327. K. Sere, E. Troubitsyna: *Safety Analysis in Formal Specification.* In: *FM'99 — Formal Methods*, ed by J.M. Wing, J. Woodcock, J. Davies (Springer, 1999) pp 1564
328. P. Sestoft: *Java Precisely* (MIT Press, 2002)
329. N. Shankar: *Metamathematics, Machines and Gödel's Proof* (Cambridge University Press, UK 1994)
330. N. Shankar, S. Owre, J.M. Rushby. *PVS Tutorial.* Computer Science Laboratory, SRI International, CA, 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, 1993.
331. N. Shankar, S. Owre, J.M. Rushby, D.W.J. Stringer-Calvert. *PVS Prover Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.
332. R. Sharp: *Principles of Protocol Design* (Prentice Hall, 1994)
333. C. Shekaran, D. Garlan, et al.: The role of software architecture in requirements engineering. In: *First International Conference on Requirements Engineering* (IEEE Comput. Soc. Press, CA, USA 1994) pp 239–245
334. N. Shresta, T. Janowski: Model-Based Travel Planning. In: *[75]* (Springer, 2002) pp 219–242
335. J.U. Skakkebæk: Development of Provably Correct Systems., M.Sc. Thesis, Dept. of Computer Science, Technical University of Denmark, (1991)
336. J.U. Skakkebæk, A.P. Ravn, H. Rischel, C.C. Zhou: Specification of Embedded, Real-Time Systems. In: *Proceedings of 1992 Euromicro Workshop on Real-Time Systems* (IEEE Computer Society Press, 1992) pp 116–121
337. J.U. Skakkebæk, A.P. Ravn, H. Rischel, C.C. Zhou: Specification of Embedded, Real-Time Systems. Technical Report, Dept. of Computer Science, Technical University of Denmark (1991)
338. I. Sommerville: *Software Engineering*, 6th edn (Addison-Wesley, 2001)
339. E. Sørensen, N. Hansen, J. Nordahl: *From CSP Models to Markov Models: A Case Study.* IEEE Trans. Software Engineering **19** (1993) pp 554–570
340. J.F. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations* (Brooks/Cole, Thomson, CA, USA 1999)
341. J.M. Spivey: *Understanding Z: A Specification Language and Its Formal Semantics*, vol 3 of *Cambridge Tracts in Theoretical Computer Science* (Cambridge University Press, 1988)
342. J.M. Spivey: *The Z Notation: A Reference Manual* (Prentice Hall, UK 1989)
343. J.M. Spivey: *The Z Notation: A Reference Manual*, 2nd edn (Prentice Hall, 1992)
344. J. Staunstrup, W. Wolff (eds.): *Hardware/Software Co-Design: Principles and Practice* (Kluwer Academic, Dordrecht, The Netherlands 1997)
345. J. Stein (ed.): *The Random House American Everyday Disctionary* (Random House, NY, USA 1949, 1961)
346. J. Stoy: *Denotational Semantics: the Scott–Strachey Approach to Programming Language Theory* (MIT Press, 1977)
347. J. Stoy, C. Strachey: *OS6 – An Experimental Operating System for a Small Computer,* Part 1: *General Principles and Structure,* and Part 2: *Input-Output and Filing System.* Computer Journal **15**, 2–3 (1972) pp 117–124, 194–203
348. C. Strachey: Fundamental Concepts in Programming Languages. Unpubl. Lecture Notes, NATO Summer School, Copenhagen, 1967, and Programming Research Group, Oxford Univ., UK (1968)

349. C. Strachey: The Varieties of Programming Languages. Techn. Monograph 10, Programming Research Group, Oxford Univ., UK (1973)
350. C. Strachey: Continuations: A Mathematical Semantics Which Can Deal with Full Jumps. Techn. Monograph, Programming Research Group, Oxford Univ., UK (1974)
351. T. Tanaka, C.W. George: Proving Safety of Authentication Protocols. In: *[75]* (Springer, 2002) pp 243–258
352. W. Tatarkiewicz: *What is Art? Problem of Definition Today.* The British Journal of Aesthetics **11**, 4 (1971)
353. J.R. Taylor: *A Background to Risk Analysis.* Volume 2, Technical Report, Risø Research Center, Denmark (1979)
354. K.S. Trivedi: *Probability and Statistics with Reliability, Queueing and Computer Science Application.* Prentice Hall (1982)
355. E. Troubitsyna: *Specifying Safety-Related Hazards Formally.* Åbo Akademi University, Department of Computer Science, TUCS Technical Report No 270 (1999)
356. TÜV. The TÜV Certification Home Page. Electronically, on the Web: `http://www.tuev-cert.de/index_en.html`, 2005.
357. University of California at Irvine. Business Process Re-engineering, Administrative and Business Services Department. Electronically, on the Web: `http://www.abs.uci.edu/depts/vcabs/4-1.html`, 2004.
358. US Nuclear Regulatory Commission: *Fault Tree Handbook.* Washington, DC, USA (1981)
359. J. van Benthem: *The Logic of Time*, vol 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*, 2nd edn (Kluwer Academic, Dordrecht, The Netherlands 1991)
360. A. van Lamsweerde: Requirements Engineering in the Year 00: A Research Perspective. In: *Proceedings 22nd International Conference on Software Engineering* (IEEE Computer Society Press, 2000)
361. A. van Lamsweerde: Building Formal Requirements Models for Reliable Software. In: *6th International Conference on Reliable Software Technologies, Ada-Europe 2001*, vol 2043 of *Lecture Notes in Computer Science* (Springer, Leuven, Belgium 2001)
362. A. van Lamsweerde: Goal-Oriented Requirements Engineering: A Guided Tour. In: *RE'01 — 5th IEEE International Symposium on Requirements Engineering* (IEEE CS Press, Toronto, Canada 2001) pp 249–263
363. A. van Lamsweerde, R. Darimont, E. Letier: *Managing Conflicts in Goal-Driven Requirements Engineering.* IEEE Transaction on Software Engineering (1998)
364. A. van Lamsweerde, E. Letier: Integrating Obstacles in Goal–Driven Requirements Engineering. In: *Proc. ICSE 98: 20th International Conference on Software Enginereering* (IEEE Computer Society Press, Kyoto, Japan 1998)
365. A. van Lamsweerde, L. Willemet: *Inferring Declarative Requirements Specification from Operational Scenarios.* IEEE Transaction on Software Engineering (1998) pp 1089–1114
366. A. van Lamsweerde, L. Willemet: *Handling Obstacles in Goal-Driven Requirements Engineering.* IEEE Transaction on Software Engineering (2000)
367. C. Vanoirbeek (ed.): *Proceedings of the 2003 ACM Symposium on Document Engineering*, NY, USA, 2003. ACM Press.

368. J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Volumes A and B* (Elsevier, 1990)
369. H. van Vliet: *Software Engineering: Principles and Practice* (Wiley, UK 2000)
370. B. Venners: *Inside the Java 2.0 Virtual Machine (Enterprise Computing)* (McGraw-Hill, 1999)
371. J. Wamberg: Kunsbegrebets forældelse. In: *Kunstteori: Positioner i nutidig kunstdebat* (Borgen, Copenhagen, Denmark, 1999) pp 187–188
372. R. Wilhelm: *Compiler Design* (Addison-Wesley, 1995)
373. J. Willems: *Paradigms and Puzzles in the Theory of Dynamical Systems.* IEEE Trans. on Automatic Control **36**, 3 (1991) pp 259–294
374. L.J.J. Wittgenstein: *Tractatus Logico-Philosophicus* (Oxford Univ. Press, London (1921) 1961)
375. L.J.J. Wittgenstein: *Philosophical Investigations* (Oxford Univ. Press, 1958)
376. W. Wonham: A Control Theory for Discrete Event Systems. In: *Advanced Computing Concepts and Techniques in Control Engineering*, vol 47 of *Computer and Systems Sciences*, ed by M. Denham, A. Laub (Springer, 1988) pp 129–169
377. J.C.P. Woodcock, J. Davies: *Using Z: Specification, Proof and Refinement* (Prentice Hall, 1996)
378. J.C.P. Woodcock, M. Loomes: *Software Engineering Mathematics* (Pitman, London, 1988)
379. Y. Xia, C.W. George: An Operational Semantics for Timed RAISE. In: *FM'99 — Formal Methods*, ed by J.M. Wing, J. Woodcock, J. Davies (Springer, 1999) pp 1008–1027
380. E. Yu, J. Mylopoulos: Understanding "Why" in Software Process Modelling, Analysis and Design. In: *Proc. 16th ICSE: Intl. Conf. on Software Engineering* (IEEE Press, 1994)
381. C.C. Zhou, M.R. Hansen: *Duration Calculus: A Formal Approach to Real-Time Systems* (Springer, 2004)
382. C.C. Zhou, C.A.R. Hoare, A.P. Ravn: *A Calculus of Durations.* Information Proc. Letters **40**, 5 (1992)
383. C.C. Zhou, A.P. Ravn, M.R. Hansen: An Extended Duration Calculus for Real-Time Systems. Research Report 9, UNU/IIST, Macau (1993)
384. C.C. Zhou, J. Wang, A.P. Ravn: A Formal Description of Hybrid Systems. Research Report 57, UNU/IIST, Macau (1995)

Monographs in Theoretical Computer Science · An EATCS Series

K. Jensen
Coloured Petri Nets
Basic Concepts, Analysis Methods
and Practical Use, Vol. 1
2nd ed.

K. Jensen
Coloured Petri Nets
Basic Concepts, *Analysis Methods*
and Practical Use, Vol. 2

K. Jensen
Coloured Petri Nets
Basic Concepts, Analysis Methods
and *Practical Use*, Vol. 3

A. Nait Abdallah
The Logic of Partial Information

Z. Fülöp, H.Vogler
Syntax-Directed Semantics
Formal Models Based
on Tree Transducers

A. de Luca, S. Varricchio
Finiteness and Regularity
in Semigroups and Formal Languages

E. Best, R. Devillers, M. Koutny
Petri Net Algebra

S.P. Demri, E.S. Orlowska
Incomplete Information:
Structure, Inference, Complexity

J.C.M. Baeten, C.A. Middelburg
Process Algebra with Timing

L.A. Hemaspaandra, L. Torenvliet
Theory of Semi-Feasible Algorithms

E. Fink, D. Wood
Restricted-Orientation Convexity

Zhou Chaochen, M.R. Hansen
Duration Calculus
A Formal Approach to Real-Time
Systems

M. Große-Rhode
Semantic Integration
of Heterogeneous Software
Specifications

H. Ehrig, K. Ehrig,
U. Prange, G. Taentzer
Fundamentals of Algebraic
Graph Transformation

Texts in Theoretical Computer Science · An EATCS Series

K. Sikkel
Parsing Schemata
A Framework for Specification and Analysis of Parsing Algorithms

H. Vollmer
Introduction to Circuit Complexity
A Uniform Approach

W. Fokkink
Introduction to Process Algebra

K. Weihrauch
Computable Analysis
An Introduction

J. Hromkovič
Algorithmics for Hard Problems
Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics
2nd ed.

S. Jukna
Extremal Combinatorics
With Applications in Computer Science

P. Clote, E. Kranakis
Boolean Functions and Computation Models

L.A. Hemaspaandra, M. Ogihara
The Complexity Theory Companion

C.S. Calude
Information and Randomness
An Algorithmic Perspective
2nd ed.

J. Hromkovič
Theoretical Computer Science
Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication and Cryptography

K. Schneider
Verification of Reactive Systems
Formal Methods and Algorithms

S. Ronchi Della Rocca, L. Paolini
The Parametric Lambda Calculus
A Metamodel for Computation

Y. Bertot, P. Castéran
Interactive Theorem Proving and Program Development
Coq'Art: The Calculus of Inductive Constructions

L. Libkin
Elements of Finite Model Theory

M. Hutter
Universal Artificial Intelligence
Sequential Decisions Based on Algorithmic Probability

G. Păun, G. Rozenberg, A. Salomaa
DNA Computing
New Computing Paradigms
2nd corr. printing

W. Kluge
Abstract Computing Machines
A Lambda Calculus Perspective

J. Hromkovič
Dissemination of Information in Communication Networks
Broadcasting, Gossiping, Leader Election, and Fault Tolerance

F. Drewes
Grammatical Picture Generation
A Tree-Based Approach

J. Flum, M. Grohe
Parameterized Complexity Theory

D. Bjørner
Software Engineering 1
Abstraction and Modelling

D. Bjørner
Software Engineering 2
Specification of Systems and Languages

D. Bjørner
Software Engineering 3
Domains, Requirements, and Software Design