Dines Bjørner

SOFTWARE ENGINEERING, 3

Domains, Requirements and Software Design

The Nancy Lectures, 2007

October 17, 2007

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

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.

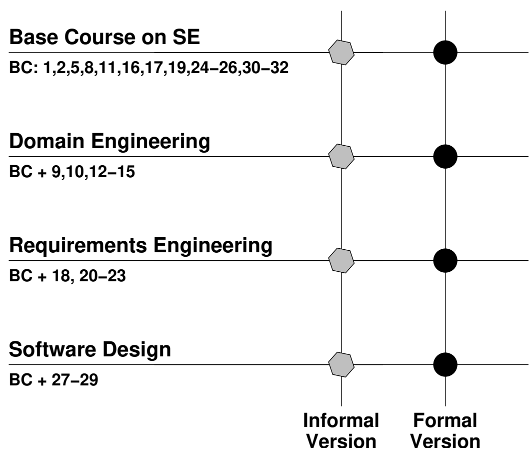


Fig. 1. Course alternatives

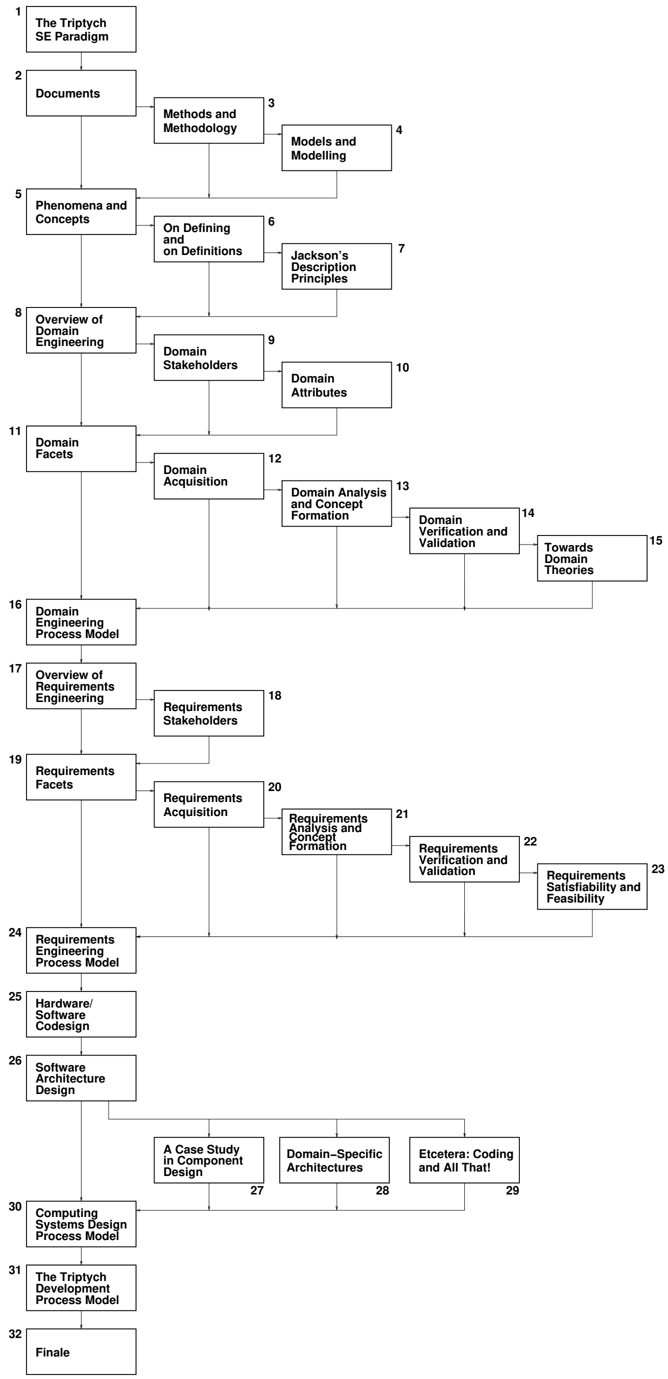
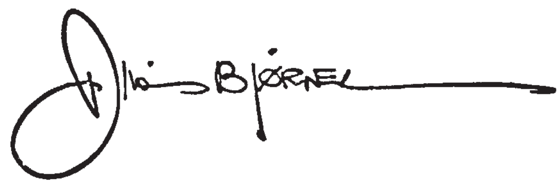


Fig. 2. Chapter precedence graph

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 [**?**] 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

Preface V
General V
Brief Guide to Volume 3 V
Acknowledgements VIII

8 Overview of Domain Engineering 1
8.1 Introduction 1
8.2 A Review of *Why Domain Engineering?* 2
8.3 Overview of Part and Chapter 2
8.4 Domain Stakeholders and Their Perspectives 3
8.5 Domain Acquisition and Validation 4
8.6 Domain Analysis and Concept Formation 4
8.7 Domain Facets 5
8.8 Auxiliary Stages of Domain Development 5
8.9 The Domain Model Document 6
8.9.1 A Preview of Things to Come 6
8.9.2 Contents of a Domain Model Document 6
8.10 Further Structure of This Part 7
8.11 Bibliographical Notes 7
8.12 Exercises 7

11 Domain Facets 9
11.1 Introduction 9
11.1.1 Separation of Concerns 11
11.1.2 Discussion of the Separation Principle 11
11.1.3 Structure of Chapter 11
11.2 Domain Facilitators: Business Processes 11
11.2.1 Business Processes 12
11.2.2 Overall Principles 15
11.2.3 Informal and Formal Examples 16
11.2.4 Discussion 20

11.2.5 Summary 21
11.2.6 Reminder 21
11.3 Domain Intrinsics 22
11.3.1 Overall Principles 22
11.3.2 Conceptual Versus Actual Intrinsics 26
11.3.3 Methodological Consequences 28
11.3.4 Discussion 28
11.3.5 Utter Barebones Intrinsics 28
11.3.6 Reminder 29
11.4 Domain Support Technologies 29
11.4.1 Overall Principles 30
11.4.2 Methodological Consequences 33
11.4.3 Discussion 34
11.4.4 Reminder 34
11.5 Domain Management and Organisation 34
11.5.1 Overall Principles 35
11.5.2 A Conceptual Analysis, I 36
11.5.3 Methodological Consequences, I+II 37
11.5.4 Conceptual Analysis, II 37
11.5.5 Methodological Consequences, III 39
11.5.6 Discussion 40
11.5.7 Reminder 40
11.6 Domain Rules and Regulations 40
11.6.1 Overall Principles 41
11.6.2 Methodological Consequences 42
11.6.3 Rules and Regulation Languages 44
11.6.4 Principles and Techniques 44
11.6.5 Reminder 45
11.7 Domain Scripts 45
11.7.1 The Description of Scripts 45
11.7.2 Methodological Consequences 65
11.7.3 Reminder + More 66
11.8 Domain Human Behaviour 66
11.8.1 Overall Principles 67
11.8.2 Methodological Consequences 71
11.8.3 Human Behaviour and Knowledge Engineering 73
11.8.4 Discussion 73
11.8.5 Reminder 73
11.9 Other Domain Facets? 74
11.10 Composition of Domain Models 74
11.10.1 Collating Domain Facet Descriptions 74
11.10.2 Technical Issues 76
11.11 Exercises 76
11.11.1 A Preamble 76
11.11.2 The Exercises 76

17 Overview of Requirements Engineering . . . 79
17.1 Introduction . . . 82
17.1.1 Further Characterisation of 'Requirement' . . . 83
17.1.2 The "Machine" . . . 83
17.2 Why Requirements, and for What? . . . 84
17.2.1 Why Requirements? . . . 84
17.2.2 Requirements for What? . . . 84
17.2.3 What Does 'Implements' Mean? . . . 84
17.3 Getting Started on Requirements Development . . . 85
17.3.1 Initial Informative Documentation . . . 85
17.3.2 Requirements Eurekas . . . 87
17.3.3 Pragmatic Prescriptive Documentation . . . 88
17.3.4 Planning Requirements Development . . . 89
17.4 On Domains, Requirements and the Machine . . . 89
17.5 Overview: Requirements Engineering Stages . . . 91
17.6 The Requirements Document . . . 92
17.6.1 A Preview of Things to Come . . . 92
17.6.2 Contents of a Requirements Document . . . 92
17.6.3 Comments on Requirements Documents . . . 93
17.7 The Structure of the Rest of the Part . . . 93
17.8 Bibliographical Notes . . . 94
17.9 Exercises . . . 94
17.9.1 A Preamble . . . 94
17.9.2 The Exercises . . . 94

19 Requirements Facets . . . 97
19.1 Introduction . . . 98
19.2 Rough Sketching and Terminology . . . 98
19.2.1 Initial Requirements Modelling . . . 98
19.2.2 Rough-Sketch Requirements . . . 99
19.2.3 Requirements Terminology . . . 106
19.2.4 Systematic Narration . . . 111
19.3 Business Process Reengineering Requirements . . . 112
19.3.1 Michael Hammer's Ideas on BPR . . . 112
19.3.2 What Are *BPR Requirements?* . . . 113
19.3.3 Overview of BPR Operations . . . 114
19.3.4 BPR and the Requirements Document . . . 114
19.3.5 Intrinsics Review and Replacement . . . 115
19.3.6 Support Technology Review and Replacement . . . 115
19.3.7 Management and Organisation Reengineering . . . 116
19.3.8 Rules and Regulations Reengineering . . . 117
19.3.9 Human Behaviour Reengineering . . . 117
19.3.10 Script Reengineering . . . 118
19.3.11 Discussion: Business Process Reengineering . . . 118
19.4 Domain Requirements . . . 119

19.4.1 Domain-to-Requirements Operations . . . 119
19.4.2 Domain Reqs. and the Reqs. Document . . . 120
19.4.3 A Domain Example . . . 121
19.4.4 Domain Projection . . . 122
19.4.5 Domain Determination . . . 127
19.4.6 Domain Instantiation . . . 130
19.4.7 Domain Extension . . . 131
19.4.8 Domain Requirements Fitting . . . 134
19.4.9 Discussion: Domain Requirements . . . 137
19.5 Interface Requirements . . . 137
19.5.1 Shared Phenomena and Concept Identification . . . 138
19.5.2 Interface Requirements Facets . . . 138
19.5.3 Interface Reqs. and the Reqs. Document . . . 139
19.5.4 Shared Data Initialisation . . . 140
19.5.5 Shared Data Refreshment . . . 141
19.5.6 Computational Interface Requirements . . . 141
19.5.7 Man-Machine Dialogue . . . 142
19.5.8 Man-Machine Physiological Interface . . . 143
19.5.9 Machine-Machine Dialogue . . . 150
19.5.10 Discussion: Interface Requirements . . . 151
19.6 Machine Requirements . . . 153
19.6.1 Machine Requirements Facets . . . 153
19.6.2 Machine Reqs. and the Reqs. Document . . . 153
19.6.3 Performance Requirements . . . 154
19.6.4 Dependability Requirements . . . 156
19.6.5 Fault Tree Analysis . . . 162
19.6.6 Maintenance Requirements . . . 178
19.6.7 Platform Requirements . . . 179
19.6.8 Documentation Requirements . . . 181
19.6.9 Discussion: Machine Requirements . . . 181
19.7 Composition of Requirements Models . . . 182
19.7.1 General . . . 182
19.7.2 Collating Requirements Facet Prescriptions . . . 182
19.8 Discussion: Requirements Facets . . . 182
19.8.1 General . . . 182
19.8.2 Principles, Techniques and Tools . . . 182
19.9 Bibliographical Notes . . . 183
19.10 Exercises . . . 183
19.10.1 A Preamble . . . 183
19.10.2 The Exercises . . . 183

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 **??**) and computing systems design (Part **??**). 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. **??** 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 **??**). ▪

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. **??**

- domain acquisition, Sect. 8.5 and Chap. ??
- domain analysis and concept formation, Sect. 8.6 and Chap. ??
- domain modelling, Sect. 8.7 and Chaps. ??–11
- domain validation and verification, Sect. 8.5 and Chap. ??
- domain theory formation, Chap. ??

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 ??–?? 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. ??. Chap. ?? 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. ??.

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. ??.

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. **??**.

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. **??**.

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. **??**.

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. **??**.

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. **??**.

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 **??** 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. **??**). 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. **??**), 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. **??**). Then we cover domain analysis and concept formation (Chap. **??**) — on which the domain models build. Once domain models are believed ready, they can be validated (Section **??**), and stages and steps of domain modelling work can be verified (Sect. **??**) — often during domain modelling. Chapters **??** and **??** 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. **??**), concept analysis and formation work (Chap. **??**), we must understand what the form and contents of proper domain models should desirably be (Chaps. **??** and 11). Hence Chaps. **??** and 11 come before Chaps. **??** and **??**. 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 [**?**, **?**, **?**, **?**, **?**].

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.

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 [**?**]:

- **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` [**?**, **?**, **?**, **?**] was covered in detail in Vol. 1, Chap. 21.

[5] Petri Nets [**?**, **?**, **?**, **?**, **?**] 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.

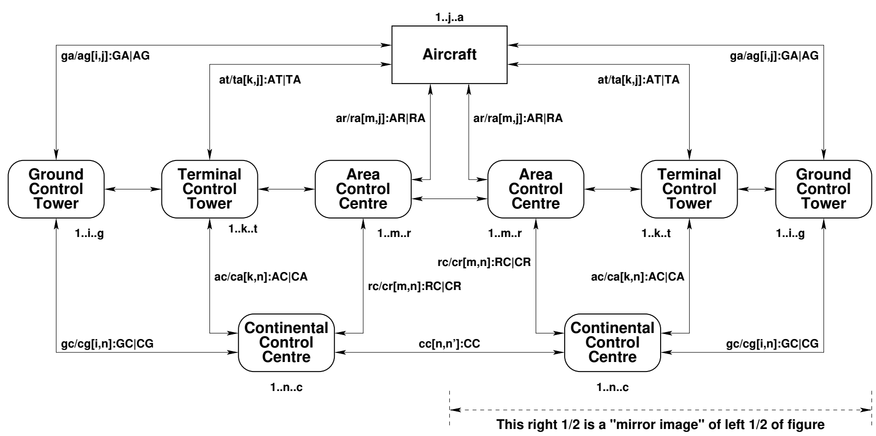


Fig. 11.1. An air traffic behavioural system abstraction

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 [?, ?, ?] and live sequence charts [?, ?, ?] were covered in detail in Vol. 2, Chap. 13.

[7] Statecharts [?, ?, ?, ?, ?] 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. ■

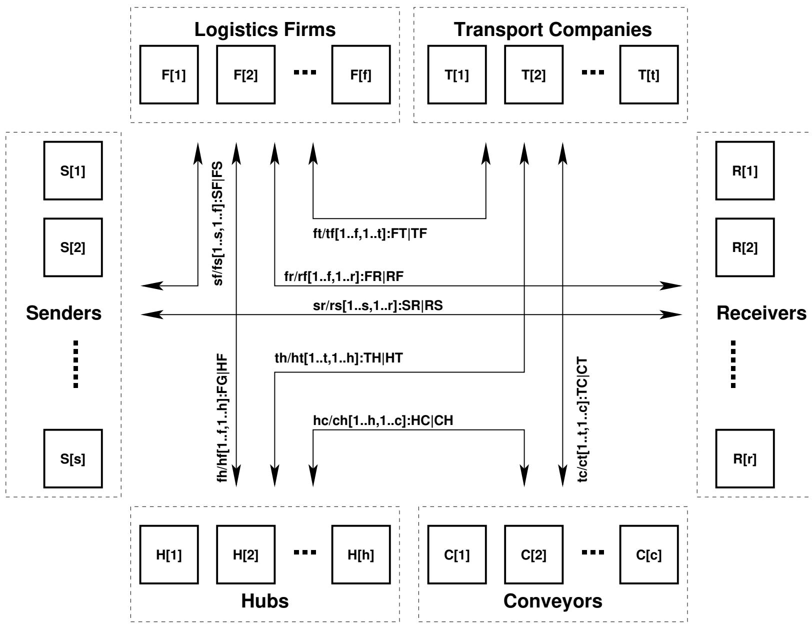


Fig. 11.2. A freight logistics behavioural system abstraction

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. ∎

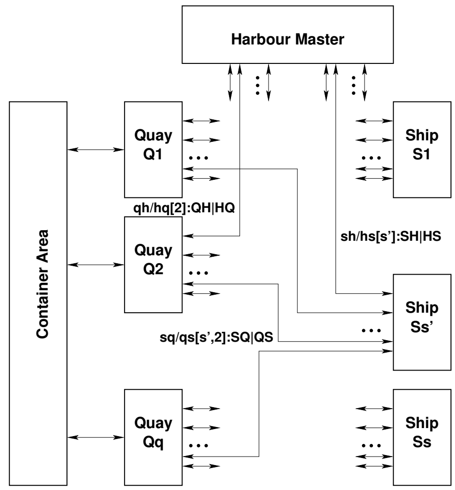


Fig. 11.3. A harbour behavioural system abstraction

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. ∎

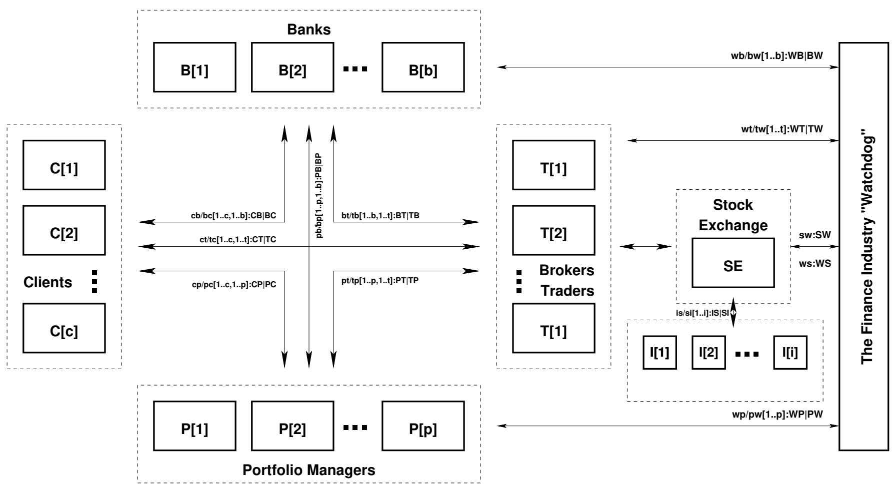


Fig. 11.4. A financial behavioural system abstraction

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. ∎

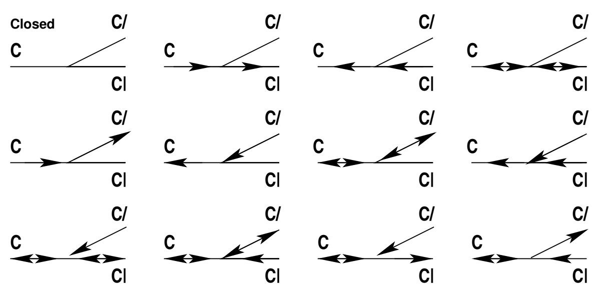


Fig. 11.5. Possible states of a rail switch

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 three-fold. 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. ■

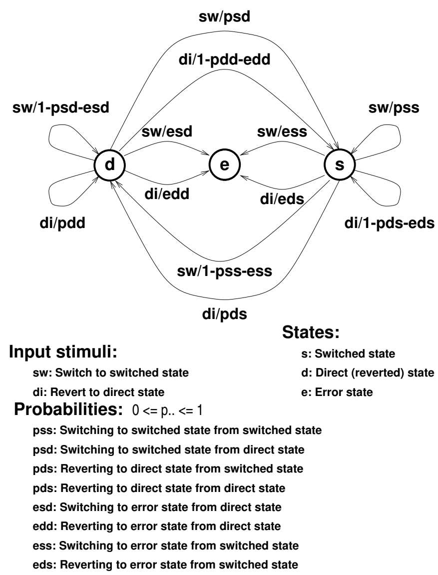


Fig. 11.6. Probabilistic state switching

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 $\overrightarrow{m}$ P)
  iTF = T → NetTraffic
  sTF = T $\overrightarrow{m}$ NetTraffic
  oG = iTF $\tilde{\rightarrow}$ sTF
value
  [close] c: NetTraffic × TN × NetTraffic $\tilde{\rightarrow}$ **Bool**
axiom
  ∀ itt:iTF, og:OG • **let** stt = og(itt) **in**
    ∀ t:T • t ∈ **dom** stt •
      t ∈ $\mathcal{D}$ 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
  $\mathrm{ST} = \Theta_i \to \Theta_a$
axiom
  $\forall$ sts:ST-**set**, st:ST • st $\in$ sts $\Rightarrow \forall\ \theta_i{:}\Theta_i, \exists\ \theta_a{:}\Theta_a \bullet \mathrm{st}(\theta_i) = \theta_a$

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. [**?**] exemplify the use of the duration calculus [**?**, **?**] 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 **??** (Chap. **??**, Sect. **??**) 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 **??**) — 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 **??**.

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!

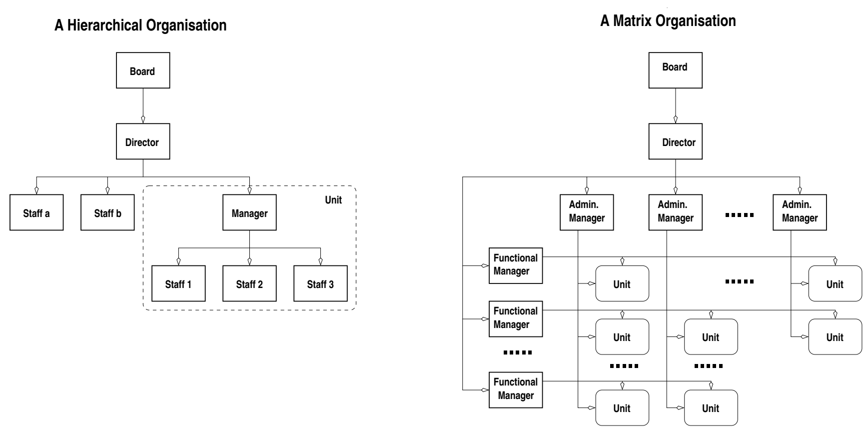


Fig. 11.7. Organisational structures

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 sσ(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)
```

$\lceil\rceil$
(2) (**let** (σ',msg) = stf_out(σ) **in** ms[i]!msg; stf(i)(σ') **end**)

stf_in: MSG $\to \Sigma \to \Sigma$,
stf_out: $\Sigma \to \Sigma \times$ 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 [?, ?, ?, ?]) processes.[13]

11.5.5 Methodological Consequences, III

The *strategic, tactical and operations resource management* example of Example **??** (Sect. **??**) 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. **??**). 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 **??** Resource Management) may be offered in the form of constraint-satisfaction solvers [**?**]. 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: $\Theta \rightarrow \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, sy_rul:Rule, stands for, i.e., has as its semantics, its meaning, rul:RUL, a predicate over current and next configurations, $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$, 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 = $\Theta \rightarrow \Theta$
  RUL = $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$
value
  meaning: Stimulus $\rightarrow$ STI
  meaning: Rule $\rightarrow$ RUL

  valid: Stimulus $\times$ Rule $\rightarrow \Theta \rightarrow \mathbf{Bool}$
  valid(sy_sti,sy_rul)(θ) $\equiv$ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

  valid: Stimulus $\times$ RUL $\rightarrow \Theta \rightarrow \mathbf{Bool}$
  valid(sy_sti,se_rul)(θ) $\equiv$ 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 = $\Theta \times \Theta \rightarrow$ **Bool**
  Act_REG = $\Theta \rightarrow \Theta$
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 θ' = (meaning(sy_sti))(θ). Let θ' violate the rule, ∼valid(sy_sti,sy_rul)(θ), then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,(meaning(sy_sti))(θ)), then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ), to remedy the situation.

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))

$\Rightarrow \exists$ nθ:Θ • act_reg(θ)=nθ $\wedge$ se_rul(θ,nθ)
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* [**?**].

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 $\rightarrow$ Bank $\rightarrow$ Bank $\times$ Reply

    int_Cmd(mkOA(c))(ρ,α,μ,ℓ) $\equiv$
        let a:A • a $\notin$ **dom** α **in**
        let as = **if** c $\in$ **dom** ρ **then** ρ(c) **else** {} **end** $\cup$ {a} **in**
        let ρ' = ρ $\dagger$ [c$\mapsto$as],
            α' = α $\cup$ [a$\mapsto$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

$$
\begin{array}{l}
\text{int\_Cmd}(\text{mkCA}(c,a))(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let}\ \rho' = \rho \dagger [\,c \mapsto \rho(c) \setminus \{a\}\,], \\
\quad\quad \alpha' = \alpha \setminus \{a\}\ \textbf{in} \\
\quad ((\rho',\alpha',\mu,\ell),\alpha(a))\ \textbf{end} \\
\quad \textbf{pre}\ c \in \textbf{dom}\ \rho \wedge a \in \rho(c)
\end{array}
$$

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

$$
\begin{array}{l}
\text{int\_Cmd}(\text{mkD}(c,a,p))(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let}\ \alpha' = \alpha \dagger [\,a \mapsto \alpha(a)+p\,]\ \textbf{in} \\
\quad ((\rho,\alpha',\mu,\ell),\text{ok})\ \textbf{end} \\
\quad \textbf{pre}\ c \in \textbf{dom}\ \rho \wedge a \in \rho(c)
\end{array}
$$

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

$$
\begin{array}{l}
\text{int\_Cmd}(\text{mkW}(c,a,p))(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{if}\ \alpha(a) \geq p \\
\quad\quad \textbf{then} \\
\quad\quad\quad \textbf{let}\ \alpha' = \alpha \dagger [\,a \mapsto \alpha(a)-p\,]\ \textbf{in} \\
\quad\quad\quad ((\rho,\alpha',\mu,\ell),p)\ \textbf{end} \\
\quad\quad \textbf{else} \\
\quad\quad\quad ((\rho,\alpha,\mu,\ell),\text{nok})
\end{array}
$$

$$
\begin{array}{l}
\quad\textbf{end}\\
\quad\textbf{pre}\ c \in \textbf{dom}\ \rho \wedge a \in \textbf{dom}\ \alpha
\end{array}
$$

Open Mortgage Account Transaction

Formal Presentation: Semantics of Open Mortgage Account Transaction

$$
\begin{array}{l}
\text{int\_Cmd}(\text{mkOM}(c,p))(\rho,\alpha,\mu,\ell) \equiv\\
\quad\textbf{let}\ m{:}M \bullet m \notin \textbf{dom}\ \ell\,\textbf{in}\\
\quad\textbf{let}\ ms = \textbf{if}\ c \in \textbf{dom}\ \mu\ \textbf{then}\ \mu(c)\ \textbf{else}\ \{\}\ \textbf{end} \cup \{m\}\ \textbf{in}\\
\quad\textbf{let}\ mu' = \mu \dagger [\,c \mapsto ms\,],\\
\qquad \alpha' = \alpha \dagger [\,a_\ell \mapsto \alpha(a_\ell) - p\,],\\
\qquad \ell' = \ \ell\ \cup [\,m \mapsto p\,]\ \textbf{in}\\
\quad((\rho,\alpha',\mu',\ell'),m)\ \textbf{end end end}
\end{array}
$$

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

$$
\begin{array}{l}
\text{int\_Cmd}(\text{mkCM}(c,m))(\rho,\alpha,\mu,\ell) \equiv\\
\quad\textbf{if}\ \ell(m) = 0\\
\qquad\textbf{then}\\
\qquad\quad\textbf{let}\ \mu' = \rho \dagger [\,c \mapsto \mu(c) \setminus \{m\}\,],\\
\qquad\qquad \ell' = \ell \setminus \{m\}\ \textbf{in}\\
\qquad\quad((\rho,\alpha,\mu',\ell'),ok)\ \textbf{end}\\
\qquad\textbf{else}\\
\qquad\quad((\rho,\alpha,\mu,\ell),nok)\\
\quad\textbf{end}\\
\quad\textbf{pre}\ c \in \textbf{dom}\ \mu \wedge m \in \mu(c)
\end{array}
$$

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,aᵢ↦α(aᵢ)+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) $\equiv$
    let a:A • a $\notin$ **dom** α **in**
    let as = **if** c $\in$ **dom** ρ **then** ρ(c) **else** {} **end** $\cup$ {a} **in**
    let ρ' = ρ † [c$\mapsto$as],
      α' = $\alpha \cup$ [a$\mapsto$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″) $\equiv$
  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) $\equiv$
    let ρ' = ρ † [c$\mapsto\rho$(c)\{a}],
      α' = α \ {a} **in**
    ((ρ',α',μ,ℓ),α(a)) **end**
    pre c $\in$ **dom** $\rho \wedge$ a $\in \rho$(c)

Derived Bank Script: Close Account Transaction

routine close_account(c **in** ″client″,a **in** ″account″ **out** ″monies″) $\equiv$
  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

$$
\begin{array}{l}
\text{int\_Cmd(mkD(c,a,p))}(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let } \alpha' = \alpha \dagger [\,\text{a}\mapsto\alpha(\text{a})+\text{p}\,] \textbf{ in} \\
\quad ((\rho,\alpha',\mu,\ell),\text{ok}) \textbf{ end} \\
\quad \textbf{pre } \text{c} \in \textbf{dom } \rho \wedge \text{a} \in \rho(\text{c})
\end{array}
$$

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

$$
\begin{array}{l}
\text{int\_Cmd(mkW(c,a,p))}(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{if } \alpha(\text{a})\geq\text{p} \\
\quad\quad \textbf{then} \\
\quad\quad\quad \textbf{let } \alpha' = \alpha \dagger [\,\text{a}\mapsto\alpha(\text{a})-\text{p}\,] \textbf{ in} \\
\quad\quad\quad ((\rho,\alpha',\mu,\ell),\text{p}) \textbf{ end} \\
\quad\quad \textbf{else}
\end{array}
$$

((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** ℓ**in**
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) $\equiv$
if ℓ(m) = 0
then
let $\mu' = \rho \dagger [\mathrm{c}\mapsto\mu(\mathrm{c})\setminus\{\mathrm{m}\}]$,
$\ell' = \ell \setminus \{\mathrm{m}\}$ **in**
((ρ,α,μ',ℓ'),ok) **end**
else
((ρ,α,μ,ℓ),nok)
end
pre $\mathrm{c} \in$ **dom** $\mu \wedge \mathrm{m} \in \mu(\mathrm{c})$

Derived Bank Script: Close Loan Transaction

routine close_loan(c **in** ″`client`″,m **in** ″`loan number`″) $\equiv$
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: ($\Sigma\times$Check$\times$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 cl)(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)
```

$\rightarrow$ ″`Monies withdrawn:`″ σ(p),
mkMN(m)
$\rightarrow$ ″`Your loan number:`″ σ(m),
OK
$\rightarrow$ ″`Transaction was successful`″,
NOK
$\rightarrow$ ″`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) $\equiv$
let av:A • av $\notin$ **dom** α **in**
let $\sigma' = \sigma \dagger [\,a \mapsto av\,]$,
as = **if** c $\in$ **dom** ρ **then** ρ(c) **else** {} **end**,
$\rho' = \rho \dagger [\,c \mapsto as \cup \{av\}\,]$,
$\alpha' = \alpha \cup [\,av \mapsto 0\,]$ **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) $\equiv$
let mv:M • mv $\notin$ **dom** ℓ**in**
let $\sigma' = \sigma \dagger [\,m \mapsto mv\,]$,
ms = **if** c $\in$ **dom** μ **then** μ(c) **else** {} **end**,
$\mu' = \mu \dagger [\,c \mapsto ms \cup \{mv\}\,]$,
$\ell' = \ell \cup [\,mv \mapsto p\,]$ **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:
  $\sigma(c) \in$ **dom** ρ.
- AisReg: Client named in c has account named in a:
  $\sigma(c) \in$ **dom**$\rho \wedge \sigma(\sigma(a)) \in \rho(\sigma(c))$.
- AhasP: Account named in a has at least the balance given in p:
  $\alpha(\sigma(a)) \geq \sigma(p)$.

- CisMReg: Client named in c has a mortgage:
  $\sigma(c) \in \textbf{dom}\ \mu$.
- MisReg: Client named in c has mortgage named in m:
  $\sigma(c) \in \textbf{dom}\mu \wedge \sigma(m) \in \mu(\sigma(c))$.
- Mhas0: Mortgage named in m is paid up fully:
  $\ell(\sigma(m)){=}0$.

Then it should be easier to "decipher" the logics:

$$
\begin{array}{l}
\text{Int\_Clause(mkChk(cex))}(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad (\sigma, \textbf{case}\ \text{cex}\ \textbf{of} \\
\qquad\quad \text{CisAReg(c)} \rightarrow \sigma(\text{c}) \in \textbf{dom}\ \rho, \\
\qquad\quad \text{AisReg(a,c)} \rightarrow \sigma(\text{c}) \in \textbf{dom}\rho \wedge \sigma(\sigma(\text{a})) \in \rho(\sigma(\text{c})), \\
\qquad\quad \text{AhasP(a,p)} \rightarrow \alpha(\sigma(\text{a})) \geq \sigma(\text{p}), \\
\qquad\quad \text{CisMReg(c)} \rightarrow \sigma(\text{c}) \in \textbf{dom}\ \mu, \\
\qquad\quad \text{MisReg(m,c)} \rightarrow \sigma(\text{c}) \in \textbf{dom}\mu \wedge \sigma(\text{m}) \in \mu(\sigma(\text{c})), \\
\qquad\quad \text{Mhas0(m)} \rightarrow \ell(\sigma(\text{m}))=0 \\
\qquad \textbf{end},(\rho,\alpha,\mu,\ell))
\end{array}
$$

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

$$
\begin{array}{l}
\text{Int\_Clause(mkA(p,t))}(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{case}\ \text{t}\ \textbf{of} \\
\qquad \text{mkAN(a)} \rightarrow (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}\mapsto\alpha(\sigma(\text{a}))+\sigma(\text{p})\,],\mu,\ell)) \\
\qquad \text{mkMN(m)} \rightarrow (\sigma,\textbf{true},(\rho,\alpha,\mu,\ell\dagger[\,\sigma(\text{m})\mapsto\ell(\sigma(\text{m}))+\sigma(\text{p})\,])) \\
\qquad \text{bank\_i} \rightarrow (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}_i\mapsto\alpha(\text{a}_i)+\sigma(\text{p})\,],\mu,\ell)) \\
\qquad \text{bank\_c} \rightarrow\ (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}_\ell\mapsto\alpha(\text{a}_\ell)+\sigma(\text{p})\,],\mu,\ell)) \\
\quad \textbf{end}
\end{array}
$$

The case, as above for adding, also holds for subtraction.

$$
\begin{array}{l}
\text{Int\_Clause(mkS(p,t))}(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{case}\ \text{t}\ \textbf{of} \\
\qquad \text{mkAN(a)} \rightarrow (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\sigma(\text{a})\mapsto\alpha(\sigma(\text{a}))-\sigma(\text{p})\,],\mu,\ell)) \\
\qquad \text{mkMN(m)} \rightarrow (\sigma,\textbf{true},(\rho,\alpha,\mu,\ell\dagger[\,\sigma(\text{m})\mapsto\ell(\sigma(\text{m}))-\sigma(\text{p})\,])) \\
\qquad \text{bank\_i} \rightarrow (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}_i\mapsto\alpha(\text{a}_i)-\sigma(\text{p})\,],\mu,\ell)) \\
\qquad \text{bank\_c} \rightarrow\ (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}_\ell\mapsto\alpha(\text{a}_\ell)-\sigma(\text{p})\,],\mu,\ell)) \\
\quad \textbf{end}
\end{array}
$$

And it holds as for subtraction, but subtracting two amounts, of values designated in p and i.

$$
\begin{array}{l}
\text{Int\_Clause}(\text{mk2S}(\text{p,i,t}))(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let } \text{pi} = \sigma(\text{p}) - \sigma(\text{i}) \textbf{ in} \\
\quad \textbf{case } \text{t } \textbf{of} \\
\quad\quad \text{mkAN(a)} \to (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\sigma(\text{a})\mapsto\alpha(\sigma(\text{a}))-\text{pi}\,],\mu,\ell)) \\
\quad\quad \text{mkMN(m)} \to (\sigma,\textbf{true},(\rho,\alpha,\mu,\ell\dagger[\,\sigma(\text{m})\mapsto\ell(\sigma(\text{m}))-\text{pi}\,])) \\
\quad\quad \text{bank\_i} \to (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}_i\mapsto\alpha(\text{a}_i)-\text{pi}\,],\mu,\ell)) \\
\quad\quad \text{bank\_c} \to (\sigma,\textbf{true},(\rho,\alpha\dagger[\,\text{a}_\ell\mapsto\alpha(\text{a}_\ell)-\text{pi}\,],\mu,\ell)) \\
\quad \textbf{end end}
\end{array}
$$

To delete an account is to remove it from both the account register and the accounts.

$$
\begin{array}{l}
\text{Int\_Clause}(\text{mkDA(c,a)})(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad (\sigma\backslash\{\text{a}\},\textbf{true},(\rho\dagger[\,\sigma(\text{c})\mapsto\alpha(\sigma(\text{c}))\backslash\{\sigma(\text{a})\}\,],\alpha\backslash\{\sigma(\text{a})\},\mu,))
\end{array}
$$

Similarly, to delete a mortgage is to remove it from both the mortgage register and the mortgages.

$$
\begin{array}{l}
\text{Int\_Clause}(\text{mkDM(c,m)})(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad (\sigma\backslash\{\text{m}\},\textbf{true},(\rho,\alpha,\mu\dagger\sigma(\text{c})[\,\mapsto\mu(\sigma(\text{c}))\backslash\{\sigma(\text{m})\}\,],\ell\backslash\{\beta(\text{m})\}))
\end{array}
$$

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.

$$
\begin{array}{l}
\text{Int\_Clause}(\text{mkCP(i,fn,aal)})(\sigma)(\text{tf})(\rho,\alpha,\mu,\ell) \equiv \\
\quad \textbf{let } \text{fct} = \sigma(\text{fn}) \textbf{ in} \\
\quad \textbf{let } \text{val} = \textbf{case } \text{fn } \textbf{of} \\
\quad\quad\quad\quad {''}\texttt{interest}{''} \to \\
\quad\quad\quad\quad\quad \textbf{let } \langle \text{m,d} \rangle = \text{aal } \textbf{in } \text{fct}(\langle \mu(\sigma(\text{m})),\text{d?} \rangle) \textbf{ end} \\
\quad\quad\quad\quad ... \to ... \\
\quad\quad\quad \textbf{end in} \\
\quad (\sigma\dagger[\,\sigma(\text{i})\mapsto\text{val}\,],\textbf{true},(\rho,\alpha,\mu,\ell)) \textbf{ end end}
\end{array}
$$

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 [?, ?, ?]. 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: $\Sigma \rightarrow$ 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 $\rightarrow \Sigma \rightarrow \Sigma$

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(...) $\equiv$ ... b_p $\lceil\rceil$ b_s $\lceil\rceil$ b_d $\lceil\rceil$ 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 **??**.

11.8.3 Human Behaviour and Knowledge Engineering

We refer to Sect. **??** 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 [?].

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 [**?**].

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. **??** for the list of **??** running domain (requirements and software design) examples; and we refer to the introductory remarks of Sect. **??** 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.

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 [**?**]): *"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. **??** 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. **??** 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 **??**: “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. **??**'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. **??**–**??**, deals with this "systematics".

17.3.3 Pragmatic Prescriptive Documentation

We now refer to Chap. **??**'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).

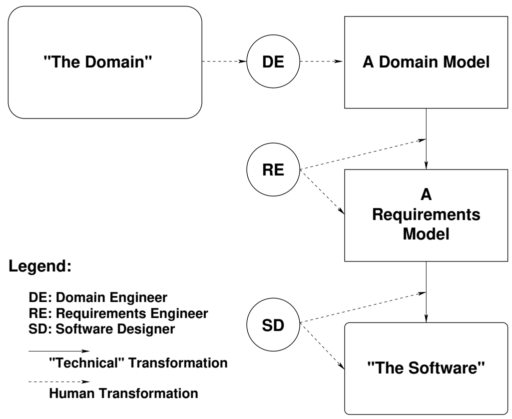


Fig. 17.1. A picture of a development process

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.

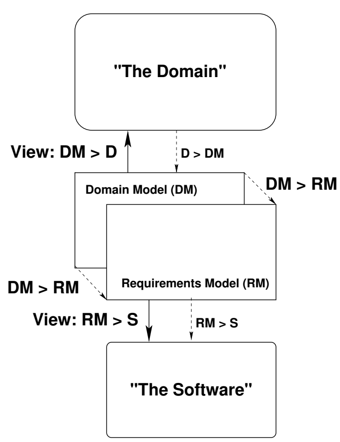


Fig. 17.2. Another picture of the development process

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. **??**. Requirements engineering then goes on with *requirements acquisition* in Chap. **??**. Then we move on to *requirements analysis and concept formation* in Chap. **??**. 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. **??**, Sect. **??**. 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. **??**, Sect. **??**. A final stage of *requirements satisfiability and feasibility* is needed to complete a full and proper requirements development, see Chap. **??**. Some of the satisfiability and feasibility study may be performed "in line" with the acquisition and/or analysis of requirements (Chaps. **??** or **??**), 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. **??**) and domain analysis and concept formation techniques (Chap. **??**). 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 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. **??**.

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. **??**), followed by requirements analysis and concept formation (Chap. **??**). Finally, we treat requirements validation (Chap. **??**), and ideas on studying the satisfiability and feasibility of requirements (Chap. **??**). We start with discussing the concept of requirements stakeholders (Chap. **??**).

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 [?, ?, ?, ?, ?].

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.

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. **??**), 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 quay-side, 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 $r : < (t_1, s_1), \ldots, (t_i, s_i), (t_{i+1}, s_{i+1}), \ldots, (t_n, \mathbf{nil}) >$ 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. **??**, 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 [**?**] (a list of terms from the P&O Nedlloyd shipping company). We stress that we have copied freely from [?] 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 [?] 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 [**?**]:

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: [?, **?**, **?**, **?**].

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 ($\sqcap$) 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. **??** 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
        $JR' = (T \times An \times T)^*$
        $JR = \{| jr{:}JR' \bullet \mathbf{len}\ jr \geq 2 \wedge ... |\}$
        $TT = Fn \xrightarrow[m]{} JR$
    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. ∎

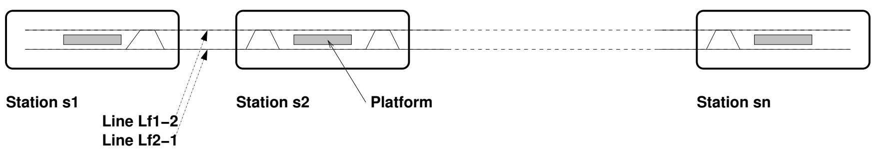


Fig. 19.1. A schematic local region railway net

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
                M_q(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 $\stackrel{\sim}{\to}$ VAL
        pre obs_A_VAL(s,a): a ∈ obs_As(s)

    $\preceq_a$: VAL × VAL → **Bool**

    is_in_range: S × (A × (VAL × VAL)) → **Bool**
    is_in_range(s,(a,(v,v′))) ≡
        v$\preceq_a$obs_A_VAL(s,a)$\preceq_a$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$\preceq_a$obs_A_VAL(s,a)$\preceq_a$ 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
      $\text{BTT}' = \text{BLn} \underset{m}{\rightarrow} (\text{BSn} \times \text{Min})^*$
      $\text{BTT} = \{|\ \text{btt:BTT}' \bullet \text{wf\_BTT(btt)}\ |\}$

```
    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 technologies. 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 23). 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 (`MSC`s [?, ?, ?], respectively `LSC`s [?, ?, ?]), 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 [?, ?, ?], respectively LSCs [?, ?, ?]), 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 →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 ∧
$\exists$ 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

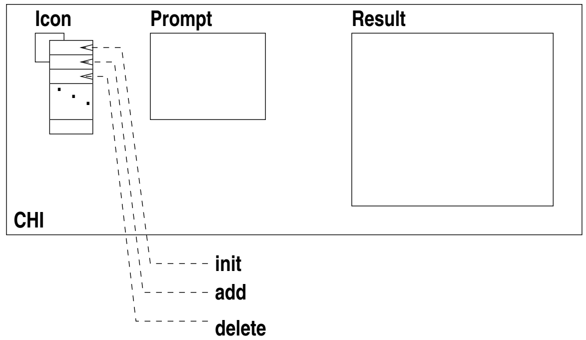


Fig. 19.2. An example CHI: staff clicking icon

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(),$\mathcal{M}_q$(mk_Brws())(tt)),
      display
        → **let** fn:Fn • fn ∈ **dom** tt ∨ ... **in**
          (display,mk_Disp(fn),$\mathcal{M}_q$(mk_Disp(fn))(tt)) **end**,
      connection
        → **let** ℓ:**Nat**,da,ta:An•{da,ta}⊆Ans(tt) ∧ ... **in**
          (connection,
            mk_Conn(ℓ,da,ta),
              $\mathcal{M}_q$(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 nondeterministically 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′) = ℳu(mk_init())(tt) in ((init,tt′,r),tt′) end,
      add → let fn:Fn,j:Journey • fn ∉ dom tt ∨ ... in
             let (r,tt′) = ℳu(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′) = ℳu(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
```

$$(\textbf{let}\ \text{q:Query}\ \textbf{in}\ \textbf{let}\ \text{v} = \mathcal{M}_q(\text{q})(\text{tt})\ \textbf{in}\ \text{system(tt)}\ \textbf{end}\ \textbf{end})$$
$$\lceil\rceil\ (\textbf{let}\ \text{u:Update}\ \textbf{in}\ \textbf{let}\ (\text{r,tt}') = \mathcal{M}_u(\text{q})(\text{tt})\ \textbf{in}\ \text{system(tt}')\ \textbf{end}\ \textbf{end})$$

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

$$\text{system: GUI} \rightarrow \text{TT} \rightarrow \textbf{Unit}$$
$$(\textbf{let}\ \text{gui}' = \text{client(gui)(tt)}\ \textbf{in}\ \text{system(gui}')\text{(tt)}\ \textbf{end})$$
$$\lceil\rceil\ (\textbf{let}\ (\text{gui}',\text{tt}') = \text{staff(gui)(tt)}\ \textbf{in}\ \text{system(gui}')(\text{tt}')\ \textbf{end})$$

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, [**?**] 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. **??** for a discourse on "What Is Art?". ■

If referring to special textbooks [**?**, **?**] 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 [?]. ■

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 [?]. ■

Characterisation. The adjudged (i.e., the 'so-judged') or hypothesised cause of an error is a *fault* [?]. ■

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 [?]. ▪

Characterisation. *Dependability* is defined as the property of a machine such that reliance can justifiably be placed on the service it delivers [?]. ▪

We continue, less formally, by characterising the above defined concepts [?]. "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 [**?**, **?**, **?**] 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 [?].

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 [**?**]. 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 [?].

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 [**?**]. 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.

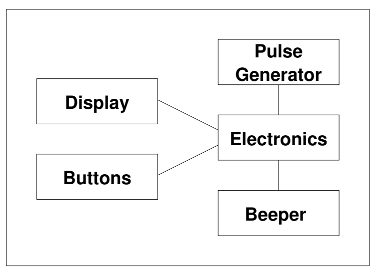


Fig. 19.3. Alarm clock

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

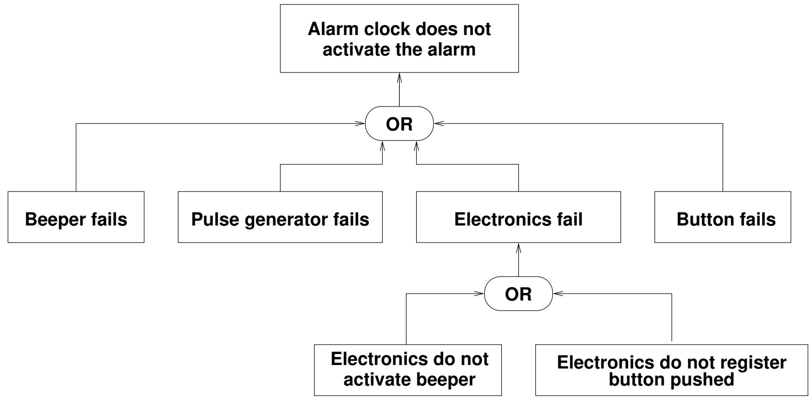


Fig. 19.4. Fault tree for an alarm clock

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 [**?**], 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.

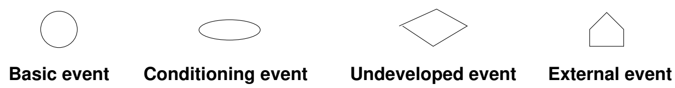


Fig. 19.5. Primary event symbols

- **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.

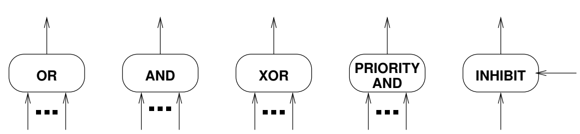


Fig. 19.6. Gate symbols

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.

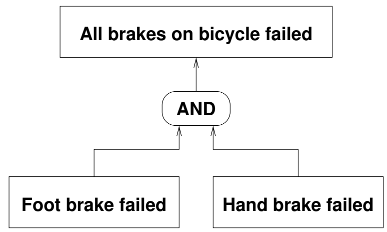


Fig. 19.7. Fault tree with AND gate

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.

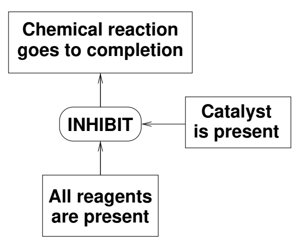


Fig. 19.8. Fault tree with INHIBIT gate

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

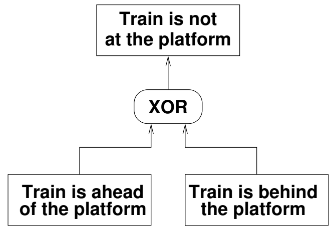


Fig. 19.9. Fault tree with XOR gate

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.

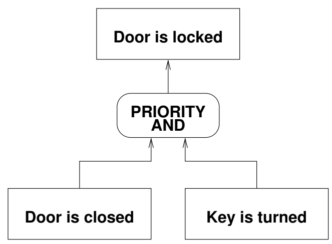


Fig. 19.10. Fault tree with PRIORITY AND gate

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 [?]: "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 [?, ?]. 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.

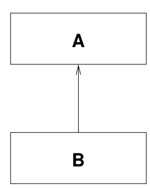


Fig. 19.11. Fault tree with no gates

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 [?, **?**], 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 $\mathrm{B_1}, \ldots, \mathrm{B_n}$ is $B_1, \ldots, B_n$. We define the semantics of A to be

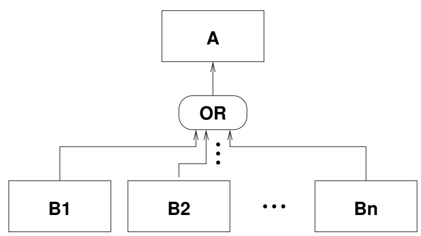


Fig. 19.12. Fault tree with OR gate

$$A = B_1 \vee \ldots \vee B_n,$$

i.e., A holds iff either B_1 or ... 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 $\mathrm{B_1}, \ldots, \mathrm{B_n}$ is $B_1, \ldots, B_n$.

We then define the semantics of A to be

$$A = B_1 \wedge \ldots \wedge B_n,$$

i.e., A holds iff $B_1, \ldots, B_n$ hold simultaneously. We have considered a more liberal interpretation of AND gates in which B_1 to B_n need not hold simultaneously, namely $A = \Diamond B_1 \wedge \ldots \wedge \Diamond B_n$. This has been rejected since this formula "remembers any occurrence of a B_i", such that if B_2 becomes true 1 year after B_1, and B_3 becomes true 3 years after B_2, and ..., then A holds. This is clearly not the intended meaning of an AND gate.

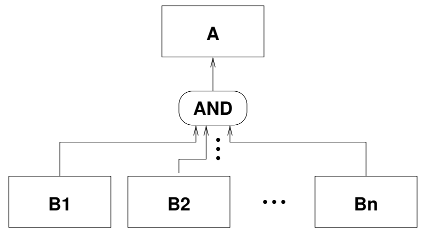


Fig. 19.13. Fault tree with AND gate

INHIBIT: We only consider INHIBIT gates in which the condition is *not* a probability statement. According to the fault tree handbook, [?], the fault tree

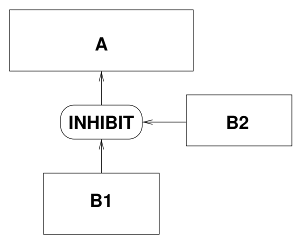


Fig. 19.14. Fault tree with INHIBIT gate

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

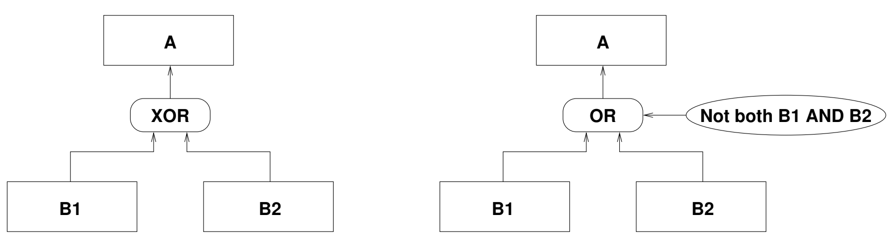


Fig. 19.15. Fault trees. Left with XOR gate. Right with OR gate and Condition

to the fault tree handbook, [?], 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

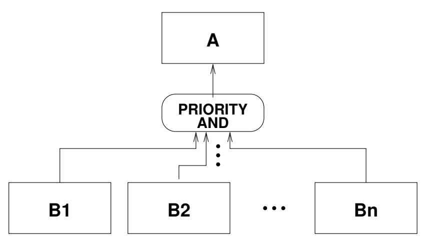


Fig. 19.16. Fault tree with PRIORITY AND gate

all the input events occur in a left to right order. Assuming that $\mathrm{B}_1, \ldots, \mathrm{B}_\mathrm{n}$ 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.)

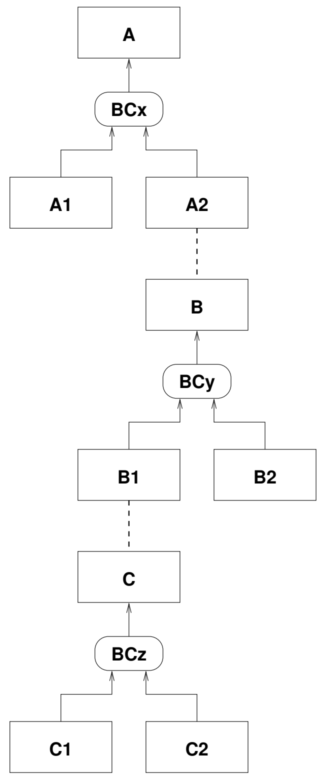


Fig. 19.17. Fault tree with three abstraction levels

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

$$\neg B \Rightarrow \neg A$$

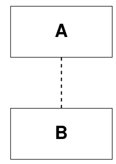


Fig. 19.18. Simple fault tree with refinement

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

$$\begin{aligned} A = &(B_1 \land \neg(B_2 \lor \ldots \lor B_n)) \\ &\lor \\ &\vdots \\ &\lor \\ &(B_n \land \neg(B_1 \lor \ldots \lor B_{n-1})). \end{aligned}$$

A safety commitment $\Box\neg A$ must be implemented by

$$\begin{aligned} \Box\neg\ (&(B_1 \land \neg(B_2 \lor \ldots \lor B_n)) \\ &\lor \\ &\vdots \\ &\lor \\ &(B_n \land \neg(B_1 \lor \ldots \lor B_{n-1}))) \end{aligned}$$

which is equivalent to

$$\begin{aligned} \Box\ (&(\neg B_1 \lor B_2 \lor \ldots \lor B_n) \\ &\land \\ &\vdots \\ &\land \\ &(\neg B_n \lor B_1 \lor \ldots \lor B_{n-1})). \end{aligned}$$

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.,

$$\Box(\text{All-false} \lor \text{Two-true})$$

where

$$\text{All-false} \equiv\ \neg(B_1 \lor \ldots \lor B_n),$$

$$\begin{aligned} \text{Two-true} \equiv\ &((B_1 \land B_2) \lor \ldots \lor (B_1 \land B_n) \\ &\lor \\ &\vdots \\ &\lor \\ &(B_n \land B_1) \lor \ldots \lor (B_n \land B_{n-1})). \end{aligned}$$

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$^+$ [**?**,**?**,**?**] and linear temporal logic [**?**,**?**,**?**], 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 [**?**], in which a probabilistic duration calculus based on discrete Markov chains [**?**] is defined and in [**?**] 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. **??** 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. **??** 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 **??**), requirements engineering (Sects. 17.6 and **??**), and software design (Sect. **??**).

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 **??**): (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 [?, ?, ?, ?]. Section 19.6.4 similarly relied almost exclusively on the delightful [?] and [?]. Section 19.6.5 is a mere editing of Chap. 4 of the splendid [?].

19.10 Exercises

19.10.1 A Preamble

We refer to Sect. **??** for the list of **??** running domain (requirements and software design) examples. We refer also to the introductory remarks of Sect. **??** 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.