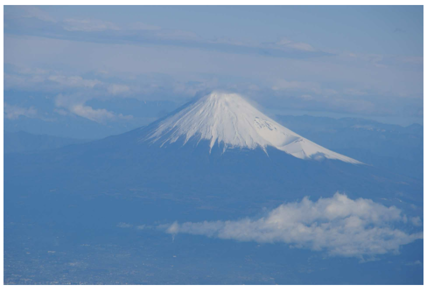
Dines Bjørner

DOMAIN ENGINEERING

Technology Management, Research and Engineering

February 16, 2009

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY
1990



Author's past affiliation

Dines Bjørner, Professor Emeritus
DTU Informatics
DK–2800 Kgs. Lyngby
Denmark

Home address

Dines Bjørner
Fredsvej 11
DK–2840 Holte
Denmark

Drs. Arimoto Yasuhito, Chen Xiaoyi and Xiang Jianwen were co-partners in the study that led to Chap. 10.

Final edit: February 16, 2009, 16:58

Dedication

Joseph A. Goguen 1941–2006
Søren Prehn 1955–2006

Foreword

This book is a collection of works done by Professor Dines Bjørner during his one year stay at JAIST's Graduate School of Information Science. He stayed at JAIST as an invited visiting professor of the 21st Century COE (Center of Excellence) project *Verifiable and Evolvable e-Society* from January of 2006.

The JAIST COE project is an advanced and unique research project aiming at applying computer science based approaches to analyses and designs of such concepts as policies, laws, regulations and standards in our society. Our current society is widely and heavily based on the world-wide network of information systems, and it seems to be not only natural but also inevitable to look into the fundamental structure of our society from the stand point of computer/information science.

Professor Bjørner has long lasting and dominant research achievements in software engineering. His recent research on formal descriptions of domains is a most advanced and challenging topic in software engineering, and also the most important foundation for *Verifiable and Evolvable e-Society*. Scientific analysis and design of any kind of system in a domain should be based on formal descriptions of basic facts and properties of the domain.

Formal description has been a main topic in formal methods, and has formed an important area of formal specification languages. Formal description of domains is also a most important challenge in formal specification languages and formal methods. During the one year stay of Professor Bjørner at JAIST, several activities of developing formal descriptions of domains in `CafeOBJ` formal specification language are started. These activities are based on Professor Bjørner's works contained in this volume. Domain descriptions in `CafeOBJ` are executable and is better to be analysed and verified, and we hope that these descriptions give new possibilities for domain engineering.

Kokichi FUTATSUGI
Professor
JAIST

Also by Dines Bjørner

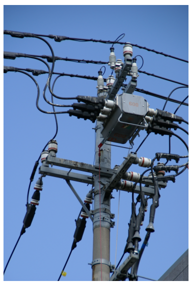
Edited / Co-edited Books

- Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer–Verlag, 1978. This was the first monograph on *Meta-IV*. DB contributions: .
- Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [71, 90, 95, 101, 120, 132, 176, 184, 214]*. EATCS Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
- Dines Bjørner and Ole N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer–Verlag, 1980.

Authored / Co-authored Books

- Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.
- Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
- Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.
- Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
- Dines Bjørner. *Domain Engineering*. JAIST Press, March 2009. This Research Monograph is based on the following 2006 Technical Memoranda from JAIST School of Information Science: [25, 27–30, 34, 36–38, 62]. *This is a self-reference!*
- Dines Bjørner. *Software Engineering, Vol. I: The Triptych Approach, Vol. II: A Model Development*. To be submitted to Springer for evaluation in 2009, Expected published 2010. Either this book ms. is submitted or that listed next is submitted.
- Dines Bjørner. *Domain Engineering*. To be submitted to Springer for evaluation in 2009, Expected published 2010. Either this book ms. is submitted or that listed just above is submitted.

Preface



This monograph is basically about ‘Domain Engineering’, in the author’s opinion, a significant phase of software engineering, a phase which precedes ‘Requirements Engineering’. It is not a textbook. For textbooks we refer to [33, Part IV, Chaps. 8–16] and the planned [44] or [43].

[1] On This Monograph

This monograph is based on a number of reports [25,27–30,34,36–38,62] that I wrote during my one year sabbatical at JAIST’s School of Information Science (IS) as the guest of Prof. Kokichi Futatsugi, February 7, 2006 to January 30, 2007. Electronic versions of all the memoranda have been available on the Internet since their first writing http://www.jaist.ac.jp/˜bjorner/,

Several of these memoranda are in a rather rough state. I therefore welcomed the kind suggestion by Prof. Kokichi Futatsugi to assemble them all, in a slightly more polished form in the present monograph.

The intention of the present monograph is therefore just that: to record — more publicly — i.e., bear witness of the work of these memoranda, and hence to one of the facets of the COE (Verifiable and Evolvable E-Society) project at JAIST.

[2] On Chapters and Appendices

[2.1] Technology Management

Chapters 1–2 [Part I]: On Domains and On Domain Engineering [28] and Possible Collaborative 'Domain' Projects [29] are intended for consumption by an audience of software technology — from mid-level to high-level — managers. [28, 29] were thus presented at meetings with such managers in Tokyo during the Spring of 2006.

The aim of these memoranda and the presentations were to inform and their objective were to engage selected Japanese software houses and JAIST IS in joint R&D projects — á la those of the ESPRIT (European Strategic Programme of Research in IT), ESPRIT BRA (Basic Research Auctions) and the FP (Framework Programme) of the EU (European Union).[1]

Chapter 1 covers a lot of ground: sketches the new science and engineering of software. Appendices A–E (originally part of [28]) provide experimental evidence of domain engineering. Chapter 1 is a "bit longish" (Pages 3–38).

Chapter 2, in contrast, is relatively short (Pages 39–53) and less technical than Chap. 1. Chap. 2 spells out the modalities (i.e., practicalities) of possible joint industry/academia (JAIST) projects — such as seen from the academic side.

[2.2] A Science & Engineering of Domain Models

Chapters 3–6 [Part II] covers issues of Domain Engineering that are not covered in [33, Part IV, Chaps. 8–16]. Thus they can be read as extending that textbook bum monograph.

Chapter 3, The Rôle of Domain Engineering in Software Development [37] was written for and presented as an invited talk at the October 2006 meeting of the IPSJ (Information Processing Society of Japan) Software Engineering Symposium 2006, Oct. 21, 2006, Tokyo. Chap. 3 gives a capsule, a summary, view of Domain Engineering, a short view of Requirements Engineering — a summary which emphasises how requirements prescriptions can be systematically "derived" from domain descriptions. This 'derivation' methodology is new. It further supports the maturity of Domain Engineering.

Chapter 4, Verified Software for Ubiquitous Computing [34] was written for and presented as an invited talk at the 29 October 2006 1AWCVS (First Asian Working Conference on Verified Systems). The present Chap. 4 represents an extended version of the talk as recorded on the 1AWCVS CD ROM.

Chapter 5, The Triptych Process Model [38], was written for and presented as an invited talk at JASPIC 2006 (Japan Association for Software Process

[1]The current author has, since 1998 [16], encouraged Japan to pursue the question of *[Issues in Inter]National Cooperative Research — Why not* [Far East] *Asian, African or Latin American* ESPRIT*s ?*

Improvement) meeting October 12-13, 2006 at Tsukuba. Chapter 5, for the first time, represents software management principles, techniques and tools for the *Triptych* approach put forward in [31–33].

Chapter 6, Domains and Problem Frames [27], was written for and presented as an invited talk at IWAAPF (International Workshop on Advances and Applications of Problem Frames), a satellite event of ICSE 2006 (International Conference on Software Engineering) Shanghai, May 2006. Chapter 6 investigates possible relationships between Michael Jackson's *Problem Frame* [147] approach and that of the *Triptych* approach [31–33].

[2.3] Experimental Evidence

Chapters 7–10 [Part III] represent less polished material than Chaps. 7–10. As the title of this part suggests they represent examples of the diversity of applications to which Domain Engineering can be put.

Chapter 7, Documents — A Domain Analysis [25], is a torso. The chapter suggests how a simple notion like 'document' can be subjected to precise analysis — where the concept of 'documents' is viewed semantically (and certainly not syntactically, as for the ODA (Open Document Architecture, http://en.wikipedia.org/wiki/Open_Document_Architecture)).

Chapter 8, Public Government — A Domain Analysis [30] was first presented at an E–Government conference, in Macau, organised by UNU–IST and the Macau SAR Government in May 2006. It is still a torso. I consider it an important example of how one may look at conventional public government to derive inspiration to upcoming E–Government (http://www.egov.iist.unu.edu/). The work of Chap. 8 was explained to Miss Chen Xiaoyi and it became the basis for her PhD study.

Chapter 9: Towards a Model of IT Security — Security Rules & Regulations: An Interpretation [36], arose as the result of a working group of colleagues from JAIST IS and the IBM Tokyo Research Laboratory. This working group met a number of times in the period from March 2006 to and including January 2007. The current status of [36] as well as of Chap. 9 is still far from satisfactory. But the ideas expressed: discussed and narrated, and also partly formalised, are such, I strongly think, that the reader should find it worth a read.

Chapter 10, A Family of Script Languages [62], like Chapter 9 (i.e., like [36]) is also in a far from satisfactory state. Work on this topic started in February 2006 as the initial work of the JAIST COE *Digital Rights: Consumers and Producers in a Digital World* project: http://www.ldl.jaist.ac.jp/drcp/. The work three JAIST students: Mr. Arimoto Yasuhito (then an MSc student), Miss Chen Xiaoyi (then a PhD student) and Mr. Xiang Jianwen (PhD, and then a Postdoc student). We first studied a number of reports and published papers [3,9,10,72,74–76,113,117,141,154,166,167,177,188,189,206,207,220]. We then selected [113] and [206, 207] and modelled these in `RAISE` and in `OTS/CafeOBJ`. An example is Sect. H. We were all somewhat alarmed to find that Gunter's "formal–looking" model was fraught with errors. Imagine the embarrassment that I had as a tutor of future researchers when having to "gloss-over" the rather sloppy work of [113] (repeated inquiries with Prof. Carl A. Gunter, IoIUC, have so far remained unanswered).

[2.4] Example Appendices

Appendices A, Business Processes (BP) and BP Reengineering (PBR), B, Towards a Domain Model of Transportation, C, Towards a Domain Model of Manufacturing, D, Towards a Model of CyberRail and E, Towards a Domain Model of "The Market", were part of [28] (On Domains and On Domain Engineering) but were editorially moved to these Example Appendices. These appendices were meant to "convince", by example, readers of [28, 29] and now of Chaps. 1–2 that the idea of domain engineering is not just a "gleam in the eye" of Dines Bjørner.

Appendix A illustrates some of the techniques that go into the domain engineering modelling of business processes and the requirements engineering of business process reengineering (BPR). The applications that are shown in Appendix A are of "real", very large systems

Appendix B is an example of a carefully worked–out model basic aspects (i.e., the transport nets) of a general notion of transportation systems.

Appendix C is a sketch narrative and formalisation some work flow aspects of machining and assembly manufacturing.

Appendix D is a sketch narrative and formalisation of Japanese technology research done at RTRI, the Japan Railway Technical Research Institute, notably by Dr. Takahiko Ogino of RTRI.

Appendix E, for which re-publication has been granted, attempts to model a conventional notion of some aspect of the domain of the consumer to retailer to wholesaler to producer market, all in preparation for work on E-market IT.

• • •

Since I left JAIST, at the end of January 2007, I have worked on further domain models. I do so in order to "sharpen" the principles, techniques and tools of the Domain Engineering method.

You can find these models through the Internet:

- **A Container Line Industry:** http://www2.imm.dtu.dk/˜db/container-paper.pdf
- **A Petroleum Industry:** http://www2.imm.dtu.dk/˜db/de-p.pdf. This document represents lecture notes and the petroleum industry example is in Vol. II of that "book" [43].
- **Transportation:** http://www2.imm.dtu.dk/˜db/transport.ps and http://www2.imm.dtu.dk/˜db/tseb.ps. The latter represents lecture notes and the transportation example is in Vol. II of that "book" [44].

The resulting "sharpening" the principles, techniques and tools of the Domain Engineering method has been published:

- [39] *Transportation Systems Development*
- [35] *Domain Theory: Practice and Theories, Discussion of Possible Research Topics*
- [42] *Domain Engineering*
- [40] *Believable Software Management*
- [41] *From Domains to Requirements*
- [48] *Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering*

[3] The Monograph: A Repository of Work in Progress

My work at JAIST was a bit too hectic for doing serious research. My assistance was called for from many quarters. It was my assessment that if my Japanese colleagues asked for this-and-that, then had had, I thought, considered their request rather seriously, discussed it with colleagues, etc. Therefore it was difficult, if not impossible, to say no. I had hoped to work, in more depth, on a few research issues. These became several research issues, too many for serious consideration. In addition came many similarly requested "speaking engagements". For most of these I wrote a special paper. I do not regret, at all, these debilitating "excursions". All this to say that there is "stuff", in this Monograph, for many a research study.

[4] On Reading The Formalisations

This monograph contains a great number of formalisations. Each is accompanied by mostly highly "stylised" narratives: That is, enumerated and tersely formulated English descriptions with the formalisation line numbers referring to lines of the narrative. The formalisations are, in this monograph, expressed in the `RAISE` Specification Language `RSL`. Most of the formalisations could, inter alia, have been rather similarly expressed in either of a number of other model-oriented formal specification languages:

- `Alloy` [146],
- `Event B` [1,71],
- `VDM` [55,56,95,96],
- `Z` [132,133,229,230,242]

combined, in several cases, with

- `CSP` [137,138,218,222].

The monograph, after almost 30 years of frequent such formalisations — widely published — assumes that the reader is reasonably professionally educated, say in one of the above-mentioned model-oriented formal specification languages as well as in `CSP`. Appendix!I does, however, provide a terse overview of `RSL`. We can otherwise refer to the following three set of sections for easy-to-grasp and short, "guided-tour-introductions" to `RSL`:

- Sects. 3.2.1–3.2.2 (Pages 58–60)
- Sect. 4.3.2 (Pages 86–90) and
- Sect. 6.2 (Pages 140–167).

Have fun!

[5] The Photos

I am grateful to the JAIST Press for their kind willingness to print, in colour, 77 photos that I took while at JAIST:

- Page V: A Kanazawa Temple
- Page VII: Fushimi Inari Shrine, Kyoto
- Page XI: Kanazawa telephone poles
- Page XVII: Train stations and flowers
- Page 1: Kanazawa houses
- Page 55: Kanazawa sushi places
- Page 177: Kanazawa &c. plans
- Page 329: Street art
- Page 443: Kanazawa fish market
- Page 471: Sakura + author in 2006
- Page 474: Kanazawa fireworks
- Page 493: Takayama lofts

[6] Summing Up

We emphasize that this monograph represents a snapshot of one year of work at JAIST.

No attempt has been made to bring the chapters in line with one another. Similarly for the example appendices

For textbooks on Domain Engineering we refer to [31–33] and to the forthcoming [44].

[7] Acknowledgements

I thank DTU Informatics, The Technical University of Denmark, my former work place, through Prof. Kaj Madsen, for allowing me a year's sabbatical at JAIST.

I thank prof. Kokichi Futatsugi (http://www.jaist.ac.jp/~kokichi/) for inviting me to be a visiting research professor at JAIST School of Information Science's Language Design Laboratory (http://www.ldl.jaist.ac.jp/index-e.html) in the period February 7, 2006 to January 30, 2007; for providing comfortable conditions for work and for suggesting this monograph. I have known Kokichi since the summer of 1984 when we first met at SRI Intl., Menlo Park where he was working with the late Joseph A. Goguen on what became known as OBJ–3. So to celebrate 25 years of fond memories we present this monograph!

This acknowledgement also extends to Prof. Takuya Katayama, now President of JAIST (http://www.jaist.ac.jp/english/president/index.html). I first

met Katayama-san at Tokyo Institute of Technology more than 25 years ago! Always a pleasant acquaintance.

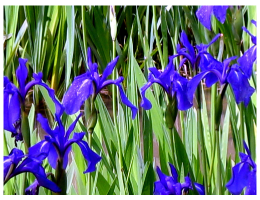
To Dr. Kazuhiro Ogata I also extend warmest acknowledgements: You helped make our stay in Kanazawa most pleasant, you helped me with our students, introducing them, ever so gently, to `OTS/CafeOBJ` and you were otherwise a most pleasant colleague in achieving this monograph.

I also acknowledge the pleasure of having worked with three very pleasant JAIST students: Mr. Arimoto Yasuhito (MSc student, soon to get a PhD), Miss Chen Xiaoyi (PhD student, now PhD) and Mr. Xiang Jianwen (PhD, Postdoc student) on the JAIST COE *Digital Rights: Consumers and Producers in a Digital World* project: http://www.ldl.jaist.ac.jp/drcp/. Our joint work is reflected in Chap. 10,

Finally I acknowledge, with pleasure, the inspiration received daily, for 11 months in 2006, from Dr. René Vestergaard (http://www.jaist.ac.jp/~vester/).

[illegible]

Fredsvej 11, DK–2840 Holte, Denmark, February 16, 2009



Contents

Part VI Administrative Appendices

Part I

Technology Management

1

On Domains and On Domain Engineering[1]

Prerequisites for Trustworthy Software

A Necessity for Believable Management

Message to Software Technology Management

- Before software can be designed we must understand its requirements.
- Before requirements can be prescribed we must understand the domain.

With this chapter we wish to make the reader aware of a new dimension to software engineering.

- Automotive engineers have their application sciences include those of mechanics and of thermodynamics and be otherwise based on applied mathematics.
- Mobile phone engineers have their application sciences include those of electromagnetic field theory and of electronics and be otherwise based on applied mathematics.
- Application software engineers, till now, have only had their professionalism be "otherwise" based on computer science. There has, effectively speaking, not been an appropriate set of application sciences, one for each domain of software applications.
- Domain engineering, applied to application areas such as administrative forms processing (i.e., documents), air traffic, financial service systems, health care, manufacturing, "the market" (including digital rights management), transportation, etc., raises the specter of there now emerging proper software application sciences.

In this chapter we outline, in a more pedantic manner, several of the issues stemming from domain engineering.

- We bring small in-line examples illustrating facets of domains and their description.
- We summarise engineering approaches to domain and to requirements modelling.

[1]This is an edited version of [28]. Presented, together with [29], at a number of meetings with Japanese Software and IT industry leaders during the Spring of 2006.

- We show how the latter, requirements engineering, changes and becomes more stable and a more well-founded professional activity.
- And we show, in in Appendices A–E, examples of domain descriptions of (1) a variety of business processes and their reengineering, (2) transportation nets, (3) manufacturing, (4) "the market" and (5) CyberRail.

We hope with this to make you interested in making your group an even more professional engineering enterprise.

End of Message to Software Technology Management

Preface

Warning: This is a Casual Document

This is not a technical scientific document. This is a casual, sort of "advertisement" document. Behind this document lies a major three-volume technical/scientific reference work [31–33]. What may appear as claims in the present document are fully substantiated in the 2416 pages of this referenced major work.

The intended **target audiences for this document** are business, software development and research managers of from small via medium to large scale software houses as well as my peer colleagues in computer and computing science.

The **aim of this document** is to explain the concepts of domain and domain engineering, and motivate why the reader should be interested in understanding what we have to say.

The undoubtedly ambitious **objective of this document** is, on the basis of presentations given by me and my colleagues to the above-mentioned managers — and as based on this document — to convince them that their enterprise ought engage in some form of loose or less-loose collaboration aimed at some form of joint domain engineering ("trial run") activity.

1.1 FAQ: Domains and Domain Models

In this section we bring in some definitions related to domains (Sect. 1.1.1), we briefly characterise what a domain description contains (Sect. 1.1.2), and we overview the triptych dogma of developing software from domain models via requirements to software design (Sect. 1.1.3). In Sect. 1.1.3 we also briefly touch upon such issues as *"domains change slowly"*, *"requirements do not change that often"* and *"domain knowledge representing corporate assets"*. After that we overview the triptych phases of development (domain engineering, requirements engineering and software design) (Sect. 1.1.4). Sect. 1.1.5 briefly touches upon business process engineering. The section ends, Sect. 1.1.6, with a mentioning of the concepts of precise narratives and formal specifications.

1.1.1 Some Definitions

Domains

By a domain we understand a universe of discourse, an area of human activity or an area of science — sufficiently well delineated to justify giving it a name: the *name* domain, and sufficiently well distinguished from "neighbouring" universes or areas to avoid unnecessary overlap and confusion.

Examples of Domains

Examples of domains are: *air traffic*, *financial service industry (banks, insurance companies, portfolio managers, stock brokers, traders and exchanges, etc.)*, *transportation (railways or road nets or airline nets or shipping nets)*, *health care (from private physicians and pharmacies via analytical laboratories and rehabilitation clinics to hospitals, etc.)*, *manufacturing*, etc. These were examples of components of a country's or a region's infrastructure. Another example is *documents* (of any form, shape and medium, being created, modified (edited), copied, moved, etc.). And yet another example is *rights management of digital documents* (usually music recordings, books, movies, photos [images in general], also known as DRM).

Domain Engineering

By domain engineering we understand the modeling of a domain: a careful description of the domain as it is, void of any reference to possibly desired new software, including requirements to such software.

Domain Model and Domain Theory

By a domain theory we understand a formal model of a domain such that properties of the model the domain can be stated and formally verified — claiming that these properties are properties of the domain being modelled.

A domain model is thus a description of a sufficient number of domain entities, domain functions, domain events and domain behaviours — so formulated and detailed that one is able to answer most relevant questions about the domain.

1.1.2 What Is a Domain Description?

A domain description describes the domain: in natural language, for example Japanese or English, and mathematically, in some abstract, formal specification language.

What do the descriptions describe? The short answer is: entities, functions, events and behaviours. A slightly longer answer is given afterwards.

Entities and Types

First one describes the simple entities of the domain: the manifest phenomena — things that you can point to or measure by scientific instruments — and concepts derived from these — things that can be defined in terms of the phenomena.

Example 1.1 *Entities:* We exemplify the notion of simple entities:

Financial Service Industry: Bank accounts, whether demand/deposit, savings & loan, or other are simple entities. So are monies, bank cards, credit cards, securities instruments like bonds and stocks. Etcetera.

The Market: The core entity is merchandise (goods, wares for sale).

Transportation Nets: Road, rail, shipping lane, and air lane segments and corresponding junctions are simple entities. States of junctions (open or closed for movement across a junction from a segment to another segment) are conceptual simple entities.

Documents: A document is a simple entity.

■

The domain stakeholders decide which aggregations of simple entities constitute the dynamic, value-varying state of the domain, which constitute the static, more-or-less value-constant context of the domain.

Functions

Functions apply to entities, some of them "input" to the domain, some being states and/or context values of the domain and yield entities, either "output" from the domain, or new states.

Example 1.2 *Functions:* We exemplify the notion of functions:

Financial Service Industry: Opening and closing bank accounts, buying and selling securities instruments, buying on a credit card, depositing into and withdrawing monies from an account, etc., are functions.

The Market: Buyers inquiring about availability, price and delivery terms of specific merchandise, sellers offering this information, buyers ordering quantities of merchandise, sellers acknowledging such orders (or not) and delivering the goods, buyers accepting the goods — or rejecting them, and sellers invoicing accepted goods — with buyers paying the invoice, etc.

Transportation Nets: Changing the state of a junction (from “red” to “green”) is a function. So is adding a new segment, removing an old one, etc.

Documents: Creating, copying and editing documents are functions.

■

Events

We may label as events some state or context changes.

Example 1.3 *Events:* We exemplify the notion of events:

Financial Service Industry: The event of going below a credit limit when withdrawing monies from an account. The event of a bank failing to meet its obligations. The event of a listed stock company failing to properly report its quarterly dividends.

The Market: The event of running out of stock of some merchandise in some retailer or wholesaler or producer. The event of a buyer failing to pay an overdue invoice.

Transportation Nets: The event of having to turn a junction state into all “red” because of a traffic accident.

Documents: The event of creating the billionth document!

■

Behaviours

Behaviours are sequences of function actions and events.

Example 1.4 *Behaviours:* We exemplify the notion of behaviours:

Financial Service Industry: The opening of a demand/deposit account followed by a sequence of zero, one or more deposits and withdrawals and ending with the closing of the account forms a behaviour.

The Market: There are customer, retailer, wholesaler and producer behaviours, as well as the behaviours of the delivery of the merchandise from producers via wholesalers to retailers and consumers.

Transportation Nets: The movement of transport conveyours (cars, trains, ships and aircraft) along segments, and into and out of junctions, forms a behaviour.

Documents: The sequence of creating a document, editing it, copying it, editing and copying the copy, etc., forms a behaviour.

■

Domain Descriptions are Serious Documents

Examples 1.1–1.4 illustrated tiny aspects of domains. A reasonably comprehensive and fully consistent domain description of even the "tiniest" domain is a serious document. It takes much time and many human resources to establish a trustworthy domain description.

Appendices B–D (Pages 343–395) shows fragments of realistic domain models of (B) transportation nets, (C) manufacturing, (D) documents, (E) "the market" and (D) a futuristic railway service concept CyberRail.

1.1.3 The Triptych Dogma

The triptych dogma is the basis for Vol. 3 of [31–33].

The Triptych Dogma

Before software can be developed the software developers and the clients contracting this software must understand the requirements.

Before requirements can be developed the software developers and the clients contracting these requirements must understand the domain.

Needless to say, this document would not be issued if the concept of domain was widely known and if the concept of first doing domain engineering before requirements engineering was likewise well accepted.

Other Engineering Branches Have Their Domain Theories

Automotive engineers have the physical sciences of mechanics and thermodynamics as part of their well-understood domain. Mobile telephony engineers have the physical sciences of electronics and radio communication (i.e., Maxwell's Equations) as part of their well-understood domain. Aerospace engineers have celestial mechanics and aerodynamics as part of their well-understood domain. Civil engineers have soil physics and structural mechanics as part of their well-understood domain.

Nissan, Mazda and Toyota would only hire such automotive engineers who have the necessary and sufficient scientific and technical skills in their basic science. NTT DoCoMo would only hire such radio and electronics engineers who have the necessary and sufficient scientific and technical skills in their basic science. And so on.

If a software engineer develops software for the financial service industry then, besides the tool science of computing, that software engineer need know "all about" the financial service industry domain theory! Similar for software applications within transportation, health care, manufacturing, air traffic, "the market", etcetera. But they do not! And their companies still get away with it!

It is about time, we think, that application software engineers be given the same opportunity to also conduct their work professionally — by providing them with suitable domain theories.

Domains Change Slowly

Domains change slowly. The majority of phenomena and concepts of the financial service industry remain the same over decades. What you see, today, as a possibly bewildering array of fancy offers is but neat combinations of standard, well-known basic concepts, basic facilities put to new uses. Similar for "all other" domains.

In Future Requirements Will "Never" Change

Some software engineers, and especially some academic software engineering scientists claim that requirements always change — and that therefore "their little gimmick contribution" to requirements engineering offers a solution to the problem. Well, once you have understood the underlying, and, we claim, rather stable domain, then requirements tend not to change "at all"! We shall justify this seemingly "outrageous" claim in this chapter.

Corporate Assets

So, we claim, it is a good thing for a mature, professional software house to focus on its core businesses in terms of one, two or a few more domains. To build up domain knowledge — not just in the heads of its loyal staff — but more importantly, on paper, e.g., electronically "inside the computer": in the form of carefully constructed, carefully maintained, carefully protected and carefully adhered-to domain theories. Once in place, even rudiments of such theories should convince existing and potential clients that their provider is the real "pro".

1.1.4 Proper Software Development

So, for us, software development proceeds in phases:

[1] Domain Engineering

In a project aimed at developing some software application for customers, if one has not already been established for the wider domain of some application, then establish first a domain model — usually with a scope that is far wider than the usually narrow span of the subsequent requirements.

The domain model usually embodies descriptions of the following domain facets:

- the **domain intrinsics:** By domain intrinsics we understand "that in terms of which all other facets are expressed",
- the **supporting technologies** of the domain,
- the **management and organisation** of the domain,
- the **rules and regulations** of the domain,
- the **domain scripts**, and
- the **human behaviour** of the domain.

Most chapters and Appendices A–E of this monograph will touch upon some issues of how to construct a domain model.

[2] Requirements Engineering

From the domain model, in stages of development, and in close interaction with requirements stakeholders, construct the machine, i.e., the hardware + software computing system. There are three parts to requirements:

- The **domain requirements:** By domain requirements we understand those requirements which can be expressed solely using terms of the domain. (Usually domain requirements are called functional requirements.)
- The **interface requirements:** are those requirements which are expressed using terms both of the domain and of the machine — building up around the entities, functions, events and behaviours that are (to be) shared between the domain and the machine.
- The **machine requirements:** are then those requirements which can be expressed sôlely using terms of the machine. (Usually domain requirements are called non-functional requirements.)

Chapter 3 will touch upon some issues of how to construct a requirements model from a domain model.

[3] Software Design

From the requirements model, in stages of development, we design the software.

Chapters 27–28 of Vol. 3, [33], of [31–33] cover a number of techniques for "deriving" trustworthy software from requirements.

1.1.5 Business Process Engineering and Reengineering

Crucial elements in software engineering and in providing services to IT clients is that of identifying the business processes and suggesting the revision of business processes.

With carefully worked-out domain descriptions the pursuit of business process engineering and reengineering takes on a far more professional rôle.

We therefore claim that pursuing serious domain engineering helps consultancy firms better advise their clients.

We refer to Appendix A for more on business process engineering and business process reengineering (BPR).

1.1.6 Precise Narratives and Formal Specifications

Chapters 3–10 and Appendices A–D show both informal and formal domain descriptions.

- The informal, yet precise narratives are directed at domain and requirements stakeholders.
- The formal descriptions — "tuned" carefully, almost line-by-line to the informal narratives — are directed at software engineers representing both the developers and acting as consultants to the domain client.

In this document we show only RSL [31–33, 44, 101, 104, 106] specifications. At JAIST they are developing domain models in CafeOBJ [89, 90, 99, 100].

In domain and in requirements verification the strong, interactive verification features are expected to bring a heretofore unseen high level of trust to bear on domain descriptions and on requirements prescriptions.

1.2 FAQ: Domain Engineering

1.2.1 With Whom to do Domain Models?

Domain models are developed in close collaboration with stakeholders of the domain.

Example 1.5 *Stakeholders:* Typical stakeholders of the financial services domain are:

- The owners of banks, insurance companies, stock broking companies, the stock exchange, credit card companies, etc.
- The executive, divisional, and operational layers of managers of the institutions just mentioned above.
- The clerks, i.e., the "floor" workers of the banks, the insurance companies, the stock broking companies, the stock exchange, the credit card company, etc., institutions.
- The customers of banks, insurance companies, stock broking companies, the stock exchange, the credit card companies, etc.: private citizens as well as commercial forms (businesses, industries, etc.).
- The government regulatory agencies: Federal Savings & Loan Agency, Federal Reserve Bank, Stock Exchanges Commission, etc.
- The ministries of finance, commerce, etc.
- Politicians.

In other words, the stakeholder group is quite large. ■

1.2.2 What Rôle "Business Process Engineering"?

We refer to Appendix A for detailed accounts and examples of the concepts of business processes and business process reengineering.

It is of utmost importance to identify all those business processes that might possibly be affected by requisition of new computing systems. Once a new computing system has been installed then many of the people acting in the domain need change their business processes. Hence — as we shall see in the next section, *FAQ: Business Process Reengineering* — requirements engineering need establish careful reengineering prescriptions (see also Appendix A). That can only be done if the domain engineering work has constructed similarly careful business process descriptions.

1.2.3 Which Are the Facets of a Domain Model?

By a facet of a domain we understand a way of looking at the domain, some view, from some stakeholder or other perspective, of the domain.

We can identify the following domain facets:

- **[1] Intrinsics**
- **[2] Support Technology**
- **[3] Management & Organisation**
- **[4] Rules & Regulations**
- **[5] Scripts**
- **[6] Human Behaviour**

We will briefly touch upon each of these.

[1] Intrinsics

By intrinsics we mean the absolute barebones of a domain: That without which it is not meaningful to talk about anything in the domain.

Example 1.6 *Intrinsics of Transportation:* In order to transport there must be a (i) path, from one location to another, along which to transport (a sequence of one or more road segments, rail lines, shipping lanes, airlanes — called segments connected by junctions); there must something to transport, i.e., a (ii) load (freight, passenger); there must be a (iii) conveyour (that transports, i.e., a vehicle, a car, a train, a ship, an aircraft), and there must be (iv) movement. The terms path (segment, junction), load (freight, passenger), conveyour (car, train, ship, aircraft), and movement are the intrinsics of transportation. ▪

A domain description must describe all the intrinsics.

[2] Support Technology

By support technology we mean the technological or human means for affecting functions and for "carrying" (embodying) entities of the domain.

Example 1.7 *Support Technology of Transportation:* To regulate traffic along a road net, one often deploys signals, for example the red/yellow/-green semaphores of road junctions. To switch trains from one line to another line one deploys switches (point machines) and the switch technology may manifest itself in many ways: hand thrown switches (as in the very old days), mechanically pulled switches (from cabin towers with mechanical pulleys and wires), electromechanical such, or, as today, the solid state interlocking of groups of switches. ▪

A domain description must describe all the relevant support technologies. The descriptions must include descriptions of failure modes, of probabilities of failure, of timing of operations, etc.

[3] Management & Organisation

By management & organisation we mean the structure of layers of management and the issues that are dealt with by management: giving directives, setting codes of conduct (rules & regulations), "back-stopping" (timely responses to) problems arising in lower levels of the worker and manager hierarchy, etc.

Example 1.8 *Management & Organisation: Manufacturing:* In a production plant management must set strategic goals and tactical plans for implementing these goals. Strategies deal with such matters as *when should goodwill in the market be turned into extra borrowing of funds for expansion* — that is conversion of one form of resources to other forms. Tactics deal with such matters as *which allocation and scheduling of serially reusable resources must be implemented in order to achieve smooth production, balanced use of resources, etc.* ▪

A domain description must describe all the management & organisation entities, functions, events and behaviours.

[4] Rules & Regulations

By a rule we mean a directive that states how a human should act in the domain, or how technology in the domain is expected to behave, including be deployed.

By a regulation we mean a directive that states what should occur if a rule is found not to be followed.

Example 1.9 *Rules & Regulations: Banking:* Rule: For normal demand/deposit bank accounts a demand (a withdrawal of money) should not bring the account balance below zero. Regulation: If it does, then the transaction should be rejected, and the account holder notified. ■

A domain description must describe all the rules & regulation entities, functions, events and behaviours.

[5] Scripts

By a script we understand a semi- to fully formal description of rules and of regulations — such that can possibly be computerised and/or which can stand the test of the rule-of-law.

Example 1.10 *Scripts: Digital Rights Management Licenses:* Currently there is a lot of interest in DRM: Digital Rights Management licensing of use of digital works such as music and movie videos. These licenses are usually expressed in a so-called rights expression language. The DRM can then decide whether actual uses of the digital works satisfy, i.e., are in accordance with the licenses. ■

A domain description must describe all the script entities, functions, events and behaviours.

Chapter 10 exemplifies the development of a number of license and contract language designs.

[6] Human Behaviour

By human behaviour we understand the entities, functions, events and behaviours of humans as they go about discharging their work in the domain. Some do it diligently, with care, some with less care, some sloppily, some delinquently, and some in an outright criminal matter.

Example 1.11 *Human Behaviour:* A bank clerk must check and double check customer identification versus account information. Doing so is diligence. Failing occasionally to do so is sloppy. Forgetting outright to even check may be an act of criminal neglect. ■

A domain description must describe all the human behaviour entities, functions, events and behaviours: looseness, non-determinism, vagrancy, and all!

If subsequent software requirements are to cope with human failures within the above outlined spectrum and if one has not properly described that spectrum, then one cannot prescribe proper requirements.

1.2.4 How to Acquire Domain Knowledge?

We see the following — initially tentative — steps in the process of domain acquisition:

- The domain engineer is familiarised with the domain through on-site visits and casual talks with as full a variety of domain stakeholders as can be made available.
- The domain engineer may have access to more or less casual descriptions of the domain from elsewhere. Such documents are also studied in the very initial domain acquisition stage.
- The domain engineer, on the basis of such casual talks, tries out an own, rudimentary domain model — enough for the domain engineer now to formulate an extensive domain questionnaire.
- The domain engineer now present that domain questionnaire to different groups of domain stakeholders.
- For each such group the questionnaire asks questions that cover:
  - ⋆ the entities,
  - ⋆ the functions,
  - ⋆ the events, and
  - ⋆ the behaviours

  of the domain, and from each of the full variety of domain facets:
  - ⋆ the intrinsics,
  - ⋆ the support technologies,
  - ⋆ the management & organisation,
  - ⋆ the rules & regulations,
  - ⋆ the script, and
  - ⋆ the human behaviour facets.
- Individuals and groups of individuals within the diverse stakeholder communities are now, possibly guided by the domain engineers, to answer the questionnaire. Usually the answers can be expressed in one or two line statements. We refer to these statements as domain description units.
- The domain description units are now indexed, classified, categorised, analysed, and possibly revised — with stakeholders.
- A “final” such, usually very large, database registered set of domain description units — free from inconsistencies and otherwise relative complete — form the basis for the domain engineer’s domain description, informal and formal.
- Thus ends the domain acquisition stage when the domain description stage has begun.

1.2.5 How to Validate a Domain Model?

To Get the Right Domain

Domain validation is about getting the right domain. Not a domain description which describes entities, functions, events and behaviours that were not intended, but intrinsics, support technologies, management & organisation, rules & regulations, scripts and human behaviours which indeed characterise the domain in question.

After the resource-consuming domain description stage comes the stage where the proposed domain model is put forward for validation. Domain validation is a stage in which domain stakeholders, typically a subset of those who took part in the domain acquisition stage, collaborate with the domain engineers.

Line-by-line the two "parties" go through the informal description of the domain: its entities, functions, events and behaviours; its business processes; and its intrinsics, support technologies, management & organisation, rules & regulations, scripts and human behaviours. Agreement has to be reached on each and every item of description. Disagreements lead to revisions of the domain description. Sooner or later the domain modeling process stabilises.

The domain validation process can be supported technologically by reasonably sophisticated hyper-link cross-referenced domain description documents.

1.2.6 How to Verify a Domain Model?

To Get the Domain Right

Domain verifications is about getting th domain right. Not a domain description with mistakes, errors and inconsistencies, but a domain description that is consistent and relative complete.

During the domain description process many questions arise as to the interpretation of stakeholder expressed domain description units. To resolve many of these the domain engineer may have to express lemmas (propositions, theorems) that may or may not hold of the (formalised) domain description being worked out.

Domain verification is the process of analysing domain description units, stating hypotheses and of proving lemmas about, testing, or model checking the emerging domain description. Domain analysis and verification may thus involved a variety of support tools: Proof checkers, theorem provers, testing tools and model checkers.

1.3 FAQ: Requirements from Domain Models?

The aim of requirements is to prescribe a machine. The machine is the combination of hardware and software to be acquired and/or developed.

Whereas a domain description describes “the domain out there”, as it is, a requirements prescription prescribes “the machine in there”, as you would like it to be!

The domain engineer “looks at the world” and carves out some segment for description. The requirements engineer knows what is feasible with machines and tries to map a suitable segment of the domain onto a machine.

1.3.1 With Whom to do Requirements Models?

Same answer as given in Sect. 1.2.1 on page 11. See Example 1.5 on page 11.

1.3.2 What Role “Business Process Re-engineering”?

We refer to Appendix A for detailed accounts and examples of the concepts of business processes and business process re-engineering (BPR). We mandate that domain engineering be the new phase of software engineering, one which has been fully carefully carried out before requirements engineering is seriously commenced. With domain engineering and its view of business processes both requirements and business process reengineering takes on a whole new dimension — and becomes a both easier and more meaningful stage of requirements engineering.

We repeat from Sect. 1.2.2:

> *It is of utmost importance to identify all those business processes that might possibly be affected by requisition of new computing systems. Once a new computing system has been installed then many of the people acting in the domain need change their business processes. Hence requirements engineering need establish careful re-engineering prescriptions (see also Appendix A). That can only be done if the domain engineering work has constructed similar careful business process descriptions.*

1.3.3 Which Are the Facets of a Requirements Model?

There are three main parts to a requirements prescription:

- **Domain Requirements:** These are the requirements that can be expressed solely by using terms from the domain.
- **Interface Requirements:** These are the requirements that are expressed using terms both from the domain and from the machine.
- **Machine Requirements:** These are the requirements that can be expressed solely by using terms from the machine.

We will now briefly survey each of these.

Which Are the Facets of Domain Requirements?

The domain requirements are the requirements that can be expressed solely by using terms from the domain. The domain requirements describes that part of an idealised view of the domain which is to "reside" in the machine.

In addition to the domain facets (intrinsics, support technology, management & organisation, rules & regulations, scripts, human behaviour) there are an orthogonal number of domain requirements facets. They are:

- **Projection:** Not all of a domain description need be "implemented". Projection serves to identify those parts of a domain description which shall remain in the requirements prescription.

  Example 1.12 *A Projected Transportation Domain:* In Appendix B we present a domain description of a multi-modal transportation net. That description is intended to also cover dynamic aspects of such nets: traffic, the flow of vehicles across the various modalities of the net (road, rail, sea and air) including the transfer of goods between different modality conveyours, etcetera. An example projection would be to leave out all of the dynamics and focus just on rail line maintenance. ▪

- **Determination:** Those parts of a domain description which are projected onto the requirements prescription may express undesired nondeterminism, that is, a kind of "looseness" with respect to entity value, functionality, event variability and behaviour. Determination serves to endow the projected parts with a necessary and sufficient, desirable level of determinism.

  Example 1.13 *A Determined Market Domain:* In Appendix E we present a domain description of "the market". That description covers arbitrary buying and selling amongst pairs of buyers and sellers (consumers and retailers, retailers and wholesalers, and wholesalers and producers). To limit the description to only allow such orders which are covered by seller catalogues represents a determination. ▪

- **Instantiation:** Oftentimes a domain description describes a domain in its fullest generality. And very often a required application shall reside in a domain which is far less general.

  Example 1.14 *An Instantiated Transport Domain:* A domain description may have covered all possibly conceivable railway nets. But if the application is only for the Japanese Shinkansen, then that domain description can be considerably instantiated (that is, abbreviated, shortened). ▪

- **Extension:** Sometimes certain operations are not feasible in the domain. For humans to carry them out would require inordinate access to resources, including time. With computing and communication such hitherto infeasible operations may now become feasible. We say that the description of previously infeasible operations in the domain extends that domain.

The need to extend the domain has arisen in the context of prescribing requirements.

Example 1.15 *An Extended Travel Planning Domain:* In the olde days, with telephone directory thick airline guides on flights anywhere in the world, it was not feasible to think of a geographically illiterate clerk to find combinations of typically 5–6 consecutive flights that would bring a passenger from some small locality in western China to some other small locality in northern Brasil. ∎

- **Fitting:** Oftentimes two or more groups of requirements engineers are working on sufficiently distinct applications albeit within relatable domains (either the same or two or more domain that do indeed share some phenomena and concepts). Requirements may then, naturally arise, requirements that imply that the otherwise distinct set of requirements being (otherwise) worked out, from respective domain description (parts), be fitted to one another.

  Example 1.16 *Two Timetable-Fitted Transport Domains:* Two groups of requirements engineers are each working on their transport requirements, one for train traffic, its monitoring and control, and one for bus traffic, its monitoring and control. During this work it is decided to fit the two traffic timetables such that queries can involve both train and bus timetables. ∎

Which Are the Facets of Interface Requirements?

The interface requirements are the requirements that are expressed using terms both from the domain and from the machine. The interface requirements prescribe software which builds a bridge between the real domain outside the machine and its idealised counterpart inside the machine.

The interface is defined by all those entities, functions, events and behaviours that can be said to be shared between the domain and the machine. In the domain these entities, functions, events and behaviours "occur for real". In the machine they serve to "mimic" a perceived real domain.

We consider the following facets of interface requirements:

- **Shared Phenomena and Concept Identification:** A line by line inspection of the domain requirements shall reveal and result in a list of all shared phenomena and concepts: simple entities, functions, events and behaviours.

- **Shared Data Initialisation Interface:** Usually, before a new computing system, i.e., the machine, can be put to use, it must be initialised. Typically a database must be established. The software resulting from initialisation requirements is often substantial.

Example 1.17 *Supply Chain State Initialisation:* In requirements for a supply chain system of consumers, retailers and delivery services it is necessary to establish initial sales catalogues and delivery service tables. ■

- **Shared Data Refreshment Interface:** Initial information may have to be updated.

  Example 1.18 *Supply Chain State Refreshment:* In requirements for a supply chain system of consumers, retailers and delivery services it is also necessary to have to more or less irregularly to update sales catalogues and delivery service tables. ■

- **Computational Interface:** The main computations of a machine may occasionally need prompts from the domain: guidelines as whether to perform a computation in one way or in another.
  The domain operations cannot be fully implemented by the machine but its computational "path" need be supported by human interaction, that is, the operation is 'shared'.

  Example 1.19 *Consistency of Transport Net Segments:* We refer to the next interface item: MM dialogues, and to the example, Example 1.20 of that item. While successively providing input for segment after segment, and for junction after junction (not mentioned below) the input provider may decide, occasionally, to request the input vetting system to check for consistency of input data. Such requests and the possible need for the update of previously input data, amount to a computational interface. ■

- **Man-Machine Dialogue Interface:** The bulk, or mass, of interaction between the machine and the domain are guided by dialogues. MM dialogues prescribe desirable sequences of interactions and how these sequences are presented over the usually graphic interface (GUI).

  Example 1.20 *GUI for Transport Net State Initialisation:* Every segment of a transport net contains the following information: Segment identifier, segment name, segment length, identifiers of the two junctions between which the segment is connected, a set of four Bezier coordinates that approximate the segment curvature, etc. A graphic user interface "window" has names paired with blank fields for providing this kind of information. Etcetera. ■

- **Man-Machine Physiological Interface:** The MM dialogue, besides GUI, is usually "carried" by tactile instruments (keyboard, mouse, "pointing to the screen, depressing icons (buttons)"), by sound (microphones and loudspeakers), video and fingerprint recognition, etc. A close fit to the domain is often desirable.
- **Machine-Machine Dialogue:** Not all initialisation or update of information takes place over the man-machine interface. Remaining such refreshment and update occurs between machines. Typically involving data

migration from legacy systems (i.e., machine for which no proper domain engineering exists).

Which Are the Facets of Machine Requirements?

The machine requirements are the requirements that can be expressed solely by using terms from the machine, that is, the hardware and the software with which to build the machine.

- **Performance:** Performance is measured in terms of computation (response) time, storage consumption, and usage of other (equipment) resources.
- **Dependability:** To properly define the "ilities" of accessability, availability, integrity, reliability, robustness, safety, security,etcetera, one must first define the concepts of failure, error and fault. Fault tree analysis can be used to determine suitable dependability requirements.
  All this is carefully outlined in [33, Chap. 19, Sect. 19.6].
- **Maintainability:** There are a number of maintainability issues: adaptive, corrective, perfective, preventive and extensional maintenance: to fit new hardware and software, to remove bugs, to tune performance, to safeguard against failures due to other forms of maintenance, and to implement additional, new requirements.
- **Platform:** Software is developed on specific computing platforms, to be executed on specific computing platforms, to be maintained/serviced from specific computing platforms, and to be demonstrated on specific computing platforms. These platforms must be precisely prescribed.

  Example 1.21 *A Satellite System:* Software for a satellite is to be developed on a Sun Microsystems Linux platform, to be demonstrated on a Apple Computers OS X platform, to be maintained, on ground, on a IBM PC-compatible Microsoft Vista platform, but to run on a space-borne military computer platform. ▪

- **Documentation:** Software development documentation can be extensive, and encompass documents covering all phases, stages and steps of development, from domain engineering via requirements engineering to software design, including, of course, the "executable" code. Or software documentation may just be a user's guide. In-between there are installation manuals, maintenance manuals, usage logbooks, test suite manuals with test outcomes, etc.

1.3.4 How to Acquire Requirements?

Principles and techniques very much similar to those covered in Sect. 1.2.4 apply here. So we just refer the reader to study Page 15 — reading 'requirements' wherever Sect. 1.2.4 wrote 'domain'.

1.3.5 How to Validate a Requirements Model?

Principles and techniques very much similar to those covered in Sect. 1.2.5 apply here. So we first refer the reader to study Page 16 — reading 'requirements' wherever Sect. 1.2.5 wrote 'domain'. Then validation must be performed not just on the requirements prescription documents but also the underlying domain description documents.

1.3.6 How to Verify a Requirements Model?

Principles and techniques very much similar to those covered in Sect. 1.2.6 apply here. So we first refer the reader to study Page 16 — reading 'requirements' wherever Sect. 1.2.6 wrote 'domain'. Then verification must be performed not just on the requirements prescription documents but also with respect to the underlying domain description documents.

1.3.7 What About Satisfiability and Feasibility?

Satisfiability

For a requirements prescription to be satisfactory the following must hold:

- *the document must be correct* (i.e., verified),
- *the document must be unambiguous*,
- *the document must be complete*,
- *the document must be consistent*,
- *the document must be stable*,
- *the document must be verifiable*,
- *the document must be modifiable*,
- *the document must be traceable*, and
- *the document must be faithful.*

Section 23.2 of Vol. 3 [33] provides details.

Feasibility

For a requirements prescription to be feasible the following must hold:

- *the requirements prescription must be technically feasible*,
- *the requirements prescription must be economically feasible*, and
- *the requirements prescription must somehow imply implicit/derivative goals of the project.*

Sections 23.3–5 of Vol. 3 [33] provides a few more details.

Technical Feasibility

Technical feasibility amounts to:

- *Feasibility of business process re-engineering:* Can the prescribed business process re-engineering requirements be implemented? We refer to Appendix A.
- *Feasibility of hardware:* Can the required hardware be implemented?
- *Feasibility of software:* Can the required software be implemented?

Section 23.3 of Vol. 3 [33] provides details.

Economic Feasibility

Economic feasibility amounts to:

- *Are the development costs feasible?* Are they realistic and can they be funded?
- *Are the write-off costs feasible?* Is it economic to use the required system?
- *Do gains outweigh costs?*

Section 23.4 of Vol. 3 [33] provides a few more details.

Compliance with Implicit/Derivative Goals

By implicit/derivative goals we mean those goals that cannot be expressed in terms of computable functions but which are expected to be fulfilled once the required software has been in sue for some time. Classical examples are: The corporation using this software becomes more competitive, or its staff are now more satisfied with their working conditions, etc.

Compliance is hard to measure, let alone predict. But attempts should be made to assess compliance up-front.

Section 23.5 of Vol. 3 [33] provides a few more details.

1.4 Conclusion

1.4.1 Myths and Commandments of Formal Methods

As of the year 2009 some, even respectable software engineers and academics, have problems with what they refer to as formal methods, which we prefer to call formal techniques.[2] One often finds that these sceptics voice various concerns. Some in the form of myths or claims. Other concerns reflect hesitancy

[2]Recall that a [good] method is a set of principles for selecting and applying a number of principles, techniques and tools in order to [efficiently] analyse a problem and provide (i.e., construct, synthesize) an [efficient] solution to that problem. Given that the principles cannot be formalised in that they most often relate to pragmatic issues — which also cannot be formalised — it does not seem wise to refer to a method as a formal method. So instead we prefer to speak about formal techniques and formally based tools.

with respect to how such formal techniques can be inserted into university curricula and into industry.

In this section we shall discuss these and related topics.

The **aim** of this section is to cover some — perhaps, let's hope — historical objections made against formal techniques and related issues. The **objective** of this section is to prepare you with counterarguments should you become engaged in discussions centered around these topics.

First Seven Myths

Anthony Hall [116] lists and dispels the following myths (claims) about formal techniques:[3]

1. *Using formal techniques can* ***guarantee that software is perfect.***

   Of course, use of formal techniques cannot guarantee perfect software. It can, when properly followed, and in most cases indeed does, lead to far more appropriate software.
2. *Formal Techniques are* ***all about program proving.***

   Well, at least in [31–33] it is not. In those three volumes we have emphasised abstract modelling.
3. *Formal techniques are* ***only useful for safety-critical systems.***

   Formal techniques are useful for any kind of software system, whether a translator (compiler, interpreter), a database information management system, a reactive system, a workpiece (spreadsheet, text processor) system, etc.
4. *Formal techniques* ***require highly trained mathematicians.***

   No, they do not. But they do require software engineers who are willing and able to think abstractly, and here mathematics is a wonderful carrier. To do proofs requires, not highly trained logician mathematicians, but software engineers with a sense of logic, with analytic minds and the ability to reason.
5. *Using formal techniques* ***increases the cost of development.***

   No. In numerous projects (some conducted under the auspices of the European Union's IT research programmes, in the 1980s and the 1990s) it has been demonstrated that using formal techniques did not increase cost of development, and in several cases it decreased the cost. For example, consider DDC's, Dansk Datamatik Center's, very successful development of a full `Ada` compiler [58, 59, 77]. DDC spent around 44 man years to develop, on time, a United States Department of Defense validated compiler — while another European and several US companies spent at

[3]We list the "myths" and claims as enumerated in [116], but the subsequent indented comments represent our own views.

least three–five times the manpower on rather late delivery compilers. The DDCI company appears to be the only surviving Ada compiler provider.

6. *Formal techniques are* ***unacceptable to users.***

   Who says users should read formal specifications? In [33] we have stressed the importance of concurrently developing and maintaining informal as well as formal domain descriptions, requirements prescriptions and software design specifications.
7. *Formal techniques are* ***not used on real, large-scale software.***

   Of course they are. And, where they are not, they should be! Doing otherwise is basically outright criminal, and is cheating the customer — since formal techniques can be used.

We encourage the readers to study Anthony Hall's delightful [116].

Seven More Myths

Jonathan P. Bowen and Michael G. Hinchey [66] builds upon Anthony Hall's analysis [116], and add a further seven "myths" and claims:[4]

8. *Using formal techniques* ***delays the development process.***

   Like item 5 above, using formal techniques does not, in general, delay the development process. It may, and usually will demand that far more time is spent on domain and on requirements modelling, and on early stages of software design. But, also in industrial projects, use of formal techniques has shown to then decrease rather significantly the length and manpower needs of coding.
9. *Formal techniques are* ***not supported by tools.***

   Most formal techniques today come with industry-scale tool sets.
10. *Formal techniques mean* ***forsaking traditional engineering design methods.***

    No. Many traditional engineering methods still apply. Some need to be revised a little.
11. *Formal techniques* ***only apply to software.***

    No. Formal techniques are, interestingly enough, today far more widespread in hardware development than in software development. It seems hardware producers are more responsible, since the costs of having to withdraw a chip from the market can easily run into US $ 300 million.
12. *Formal techniques are* ***not required.***

[4]We list the "myths" and claims as enumerated in [66], but the subsequent indented comments represent our own views.

Yes, they are. In particular, software for military applications, in the UK, now demands the use of formal techniques.

13. *Formal techniques are* ***not supported.***

    There are now many software houses, especially in Europe, which offer consultancy advice on the use of formal techniques in other software houses' formal developments. *The Japanese company* **CSK Holdings Corporation**[5] *is a world leader in supporting their clients*[6] *in using the VDM formal method.*

14. ***"Formal methods" people always use formal methods.***

    Well, we really cannot speak on behalf of all "formal methods people". So, let's leave this one uncommented.

Despite the seeming "outdatedness" of some of the seven more myths, we still encourage the readers to study Bowen and Hinchey's delightful [66].

Ten Formal Methods Commandments

Given that the myths and claims have been disposed of in a trustworthy, believable manner, we can then go on and reiterate what has been said again and again in this monograph: When using formal techniques, please consider carefully the following sound advice from Jonathan P. Bowen and Michael G. Hinchey [67]:[7]

15. **Choose an appropriate notation.**

    Certainly.

16. **Formalise, but not overformalise.**

    What is probably meant here is: Choose an appropriate abstraction level.

17. **Estimate costs.**

    Always.

18. **Have a formal methods guru "on call".**

    See answer to item 13 above.

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

[6]Felica Networks, Japan, http://www.felicanetworks.co.jp/index.html, is one such company: The NTT/DoCoMo mobile phone, *FOMA 905i* (http://www.nttdocomo.com/features/foma905igallery/), and the *Mobile Felica* smart chip was designed using VDM extensively. In a paper at the Formal Methods 2008 Industry Day (http://www.fm2008.abo.fi/industry_day.php), Mr. Taro Kurita, FeliCa Networks Inc. and Messrs. Miki Chiba and Yasumasa Nakatsugawa Sony Corporation gave a presentation: *Application of a Formal Specification Language (VDM) in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone.*

[7]We list the Ten Commandments as listed in [67], but the subsequent indented comments represent our own views.

19. **Do not abandon thy traditional development methods.**

    See answer to item 10 above.
20. **Document sufficiently.**

    This monograph repeatedly stresses this point and [31–33] does it almost to the extreme. In most other engineering practices documentation is far more extensive than what we witness today (year 2009) in software development. So follow the advice of the present volume: *Document, document, document.*
21. **Do not compromise thy quality standards.**

    In fact, tighten your quality standards.
22. **Do not be dogmatic.**

    Creating abstract models and making design decisions as to software data structures and algorithms requires exceedingly open minds. Developing software, in general, requires the effort of at least two, and usually 5–8, people in tight collaboration. Dogmatism, sticking to early development (modelling and design) decisions, simply "is out". Your colleagues will not have it.
23. **Test, test and test again.**

    We stress formal testing, i.e., testing based on formal, abstract interpretation of formal domain descriptions, formal requirements prescriptions and formal software specifications. Besides verification and model checking, it is indeed necessary to test.
24. **Reuse.** Re-use of modules is what is primarily referred to here. Preparation for re-use is supported if the formal specification language possesses a module (class, scheme, object) concept. Then domain 'modules' can be re-use edited into requirements 'modules' and these again into software design 'modules'. Then, in "related" domains, or requirements or software designs these 'modules' can then be further "re-use edited". Otherwise re-use seems to this author that "reuse" is sort of a "white elephant," a desideratum that few can live up to.

    When, for example, a compiler is first developed, its development, all stages, from domain (i.e., language semantics) description via requirements prescription to software design, is being reused. We hope. That is, the company, the group that develops the first compiler, "survives" to make several subsequent generations of that "same" compiler, but now for slightly, or less slightly changed, requirements, for new language features, etc.

    That is reuse at the level we know and are familiar with. For us to think of reusing a module, or even a component, from some problem frame, i.e., from some domain-specific architecture development in an entirely different domain-specific architecture development makes less sense.

    It does, however, make sense in the meaning of that of the Object Management Group's (OMG) guidelines for, say, dictionary components.

The kind of dictionaries referred to have a base part which can be reused across compiler, operating system, database, and several application system developments.

So, really, all we can say is: The jury is still out, and the verdict can be expected in the next decade!

As for re-use of domain descriptions and requirements prescriptions: The very purpose of developing domain descriptions is that they be re-used whenever requirements for software within the application area are being developed.

And even the requirements, for some applications, can be partially reused, i.e., fitted to, requirements for a "neighbouring" area of the same domain.

1.4.2 FAQs: Frequently Asked Questions

General

25. *Should/can/must stakeholders understand formal specifications?*

    No, not necessarily. As we have advocated, proper developments should contain complementary informal and formal descriptions, prescriptions and specifications.

    For a number of software developments, customers may engage consultants to also check that the formal descriptions, prescriptions and specifications are up to standard.

    For a number of software products, insurance companies require that a certified company, like Lloyd's [165] (or such similar companies as Norwegian Veritas [192], Bureau Veritas [68] or TÜV [237]), regularly and irregularly, unannounced, inspect and, in various ways, check the development. Such insurance and "verification" companies are increasingly turning to formal techniques so their staff can understand and professionally evaluate the use of formal descriptions, prescriptions and specifications.
26. *What should/could be the languages of informal descriptions?*

    For domain descriptions it should be the national, i.e., natural language of the client plus the professional language of the domain. No IT jargon is basically needed — unless, of course, IT plays a nontrivial rôle in the already existing domain.

    For requirements prescriptions the answer is the same as for domain descriptions, except that now one is allowed to use, in appropriate areas of typically interface and machine requirements, an appropriate, generally established sublanguage of IT.

    For software designs — for which we have not dealt with informal annotations to any serious extent — it is, of course, necessary to use the language also of IT (software).

As for domain-specific languages, make also sure that proper terminologies are established for the IT (software) sublanguages that are used.

27. *What should/could be the languages of formal descriptions?*

    Whichever is most appropriate and at hand. For most developments that we know of, i.e., for most problem frames, the `RAISE` Specification `Language, RSL,` is adequate. You can then, when and as needed, augment `RSL` descriptions, prescriptions or specifications with Petri nets [148, 199, 210–212], message sequence charts [142–144], live sequence charts [80, 128, 153], statecharts [123,124,126,127,129] or duration calculus [247,248] descriptions, prescriptions or specifications — or several of these. These "augmentations" were covered, to some nontrivial depth, in [32, Vol. 2, Chaps. 12–15].

    Or you can use CafeOBJ [89, 90, 99, 100]. The CafeOBJ software support system allows stepwise development, execution and verification of CafeOBJ specifications.

    Or you can use B [1, 71], Event B, VDM-SL [55, 56, 95, 96] or Z [132,133,229,230,242] — all come, or will soon come, with suitable Petri net, message or live sequence chart, statechart, duration calculus or TLA+ [156, 175] augmentations.

    `RSL` variants of `UML`'s Class Diagrams may also be advisable [32, Vol. 2, Chap. 10].

28. *When have we specified enough — minimum/maximum?*

    You have specified enough, both informally and formally, when what is left to describe are such things as identifier formats. That is, when you have specified everything but possibly that, then you have specified the necessary and sufficient amounts. The trivial things left unspecified are those things that one can safely trust the software designer to make final and trustworthy design decisions about. Also, certain aspects of graphical user interfaces, specific handling of tactile input, etc., seem to belong to this class of initially unspecified things.

Domains

29. *Why domain engineering by computing scientists and software engineers?*

    Because computing science has the tools, namely the specification languages, and because computing science has the principles and techniques of abstract modelling. Mathematicians — in some sense — could be claimed to have similar such tools, but they really do not. Their abstractions go well beyond those that are needed for domain modelling. They are not interested in proof systems, for example, for formal specifications — but in the more general notions of power of such proof systems, etc. Finally, the computing scientists interface, daily, with software engineers — and, in the hard realities of the day, domain theories are the first to be demanded by software engineers.

Most formal specification languages can handle systems that evolve, that is, whose components grow and shrink. Mathematics, in todays' conventional sense, cannot handle system evolution.

30. *Should one use normative and/or instantiated domain descriptions?*

    This is a contentious issue. For a specific requirements development one may be tricked into developing only an instantiated domain description, that is, a domain description that is already instantiated to the specific domain.

    Some authors seem, in their writing, to assume instantiated domain descriptions. The author of this volume advises normative, i.e., generic, domain descriptions.

31. *Who should research and develop domain theories?*

    There are basically three possibilities, listed in causal order:

    - initially university and academic research centre **computing science** departments, i.e., their staff,
    - eventually **domain-specific** university and academic research centre departments, and
    - finally, domain-specific **commercial companies.**

    Initially it is advised that university and academic research centre **computing science** scientists research and develop domain models. As mentioned above, in item 29, initially the computing scientists have the basic methods needed to do domain theory research, and are also interested in the engineering of large-scale documentation, etc.

    But eventually, within years, say 3–5 years after the initial start of computing science R&D in domain theories, it should also be undertaken by **domain-specific** research groups: transportation, in healthcare, in financial services, in marketing and sales (e-marketing), etc. Just as such university departments are, today, using (applied) mathematics, we can foresee that they will also be able, soon, to use even fairly sophisticated computing science ideas.

    And, finally, private, **commercial companies,** for example, software houses strong in a particular application domain, will embark on such domain theory R&D, as will suppliers of any form of technology to companies within the domain.

32. *What is the timeframe for the R&D of domain theories?*

    It is strongly believed that the timeframe for the R&D of domain theories is of the order of 10 to 20 years, or in cases up to 30 years, before one can safely say that a domain theory has been established.

In other words: Patience is called for. Conviction that establishing such theories is of utmost importance is called for.

To do research and development on domain theories seems to belong to the category of "Grand Challenge" endeavours (Sect. 1.4.3).

Requirements

To us, there are basically only two questions concerning requirements development:

33. *Requirements always change, so why formalise?*

    No! It may be true that people conceive of requirements "always changing". But we venture to claim that such "changes" are really not so much "changing requirements" as they are, or reflect, increased, and hence better, understanding of the domain.

    In other words: Given that one had an established, i.e., a reasonably comprehensive, domain theory, we will then claim that requirements do not change "so much" (as before conceived)!
34. *Must we formulate requirements strictly before software design?* This question could also appear in the previous section as: *Must we determine domain descriptions strictly before requirements prescriptions?*

    In both cases the answer is: Yes, for the time being. Till such a time when we indeed have (i) reasonably firmly established domain theories, and (ii) a sufficient body of knowledge, i.e., experience with requirements "strictly derived" from domain theories, until such a time we are, due to commercial, i.e., competitive, pressures, more or less forced to develop domain descriptions hand-in-hand with requirements prescriptions, and the latter hand-in-hand with early stages of software design. The special approach to software development shows a way in which to develop domain descriptions "staggered" with the development of requirements prescriptions, and these again "staggered" with the development of software architecture design — where, by "staggering", we mean that one phase follows almost right "on the heels" of the preceding phase.

1.4.3 Research and Tool Development

Evolving Principles, Techniques and Tools

As programming methodology and computing science (i.e., foundational) research progresses, the present development principles and techniques will evolve, and more elegant forms of these can be expected. New, formal specification languages will emerge. And tools for their use, including verification, model checking and testing tools will be constructed. One thing seems, however, to be an assurance: these new principles techniques and tools (the latter including the new languages), will not deviate radically from what this monograph shows.

Grand Challenges

To *put a man on the moon* was a technological as well as a scientific grand challenge. To embark upon, conduct and complete the *human genome project* was likewise a grand challenge.

Three Dimensions of Grand Challenges

Related to this monograph we can formulate three sets of grand challenges: (i) *integration of formal techniques,* (ii) *trustworthy evolutionary systems development*, and (iii) *domain theories.* We will briefly remark on these.

Integration of Formal Techniques

In [32, Chaps. 10, 12–15] we introduced UML class diagrams, Petri nets, message and live sequence charts, statecharts and the duration calculus. The chapters suggested that, when appropriate, these other notational, mostly diagrammatic systems be used in conjunction with, for example, RSL. The formal issue is: How does the semantics of RSL fit with the semantics of UML class diagrams, Petri nets, message and live sequence charts, statecharts and the duration calculus?

The referenced chapters gave some strong hints. But "the final jury is still out!"

Much research and much experimental development still has to be done before we deploy these combinations or integrations in common industrial practice. For now they can be used in carefully monitored and integrated formal techniques guru-tutored industrial developments. We refer to a series of conferences on *IFM: (Integrated Formal Methods)*, which are held annually, for references to ongoing R&D [7,65,69,111].

- *We consider it a 'Grand Challenge' to achieve a set of formal techniques and formally based tools which together cover software development for all of today's and the immediately foreseeable applications.*

Trustworthy Evolutionary Systems Development

Software systems evolve. From when they are first delivered till they are finally disposed of they usually undergo many, many changes, that is, they are maintained: Corrected (for bugs), perfected (new functionalities are added, old functionalities are, resource-consumption-wise, made more efficient), and adapted (to new platforms). Software systems evolution, the proper handling of legacy systems, i.e., systems that have been in use, say, for decades, is a major problem. The use of formal techniques in the initial development of these is no hindrance, but, we strongly believe, the non-use of formal techniques and/or the absence of proper, fully comprehensive documentation, is an obstacle to smooth, problem-free evolution.

- *We consider it a grand challenge to achieve a set of development principles and techniques as well as a set of management practices which together cover all of today's and the immediately foreseeable applications and which by careful use — and reuse — can ensure software systems whose evolution, from initial development, via repeated adaptive and perfective maintenance, to final disposition, perhaps decades later, ensure as near bug-free software as is humanly conceivable.*

Domain Theories

We repeat our adage: software cannot be designed before we have a reasonable grasp of its requirements; requirements cannot be prescribed before we have a reasonable grasp of the domain of the software; and hence it is of utmost importance, as this monograph attests, to (somehow) build requirements development on domain theories. The somehow hedge makes room for the developers to codevelop the domain description and the requirements prescription.

- *We consider the following to be examples of grand challenges: to achieve domain theories for such domains as railways, transportation in general, the market (buyers and sellers: consumers, retailers, wholesalers, producers, brokers, distributors, etc.), healthcare, financial services (banks, insurance companies, securities instrument brokers and traders, stock (etc.) exchanges, portfolio management, etc.), and production (i.e., manufacturing), etc.*

On the Nature of "Grand Challenges"

Tony Hoare has formulated 17 criteria for a research topic to be a grand challenge. We borrow the topic lines from [135], but edit, i.e., shorten Hoare's, as usual, poignant, discussion. In other words, we strongly encourage the reader to study Hoare's paper.

The "it" below refers to "a grand challenge".

1. **Fundamental:** It relates strongly to foundations, and the nature and limits of a discipline.
2. **Astonishing:** It implies constructing something ambitious, heretofore not imagined.
3. **Testable:** It must be objectively decidable whether a grand challenge project endeavour is succeeding or failing.
4. **Revolutionary:** It must imply radical paradigm shifts.
5. **Research-oriented:** It can be achieved by methods of academic research — and is not likely to be met sôlely by commercial interests.
6. **Inspiring:** Almost the entire research community must support it, enthusiastically, even while not all may be engaged in the endeavour.
7. **Understandable:** Comprehensible by — and captures the imagination of — the general public.
8. **Challenging:** Goes beyond what is initially possible and requires insight, techniques and tools not available at the start of the project.
9. **Useful:** Results in scientific or other rewards — even if the project as a whole may fail.
10. **International:** It has international scope: Participation would increase the research profile of a nation.
11. **Historical:** It will eventually be said: It was formulated years ago, and will stand for years to come.
12. **Feasible:** Reasons for previous failures are now understood and can now be overcome.
13. **Incremental:** Decomposes into identified individual research goals.
14. **Cooperative:** Calls for loosely planned cooperation between research teams.
15. **Competitive:** Encourages and benefits from competition among individuals and teams — with clear criteria on who is winning, or who has won.
16. **Effective:** General awareness and spread of results changes attitudes and activities of scientists and engineers.
17. **Risk-Managed:** Risks of failure are identified and means to meet will be applied.

1.4.4 Application Areas

With this section we shall try to give some more examples. But the examples will only be dealt with in a discursive manner. For each of a number of such examples, we will briefly outline the application area and then refer to a monograph, a book, in which the example is covered to some non-trivial depth.

We mention the following books:

- I. Hayes (ed.): *Specification Case Studies* (Prentice Hall, 1987), [130].
- C. Jones, R. Shaw (eds.): *Case Studies in Systematic Software Development* (Prentice Hall, 1990), [151].

- H.D. Van, C. George, T. Janowski, R. Moore (eds.): *Specification Case Studies in RAISE* (Springer, April 2002), [238].

Needless to say: they (should) all belong in the reference library of the professional software engineer.

In the list below chapter references are to chapters in the above mentioned and below repeated first references. Second, separately bracketed references are to individual papers (chapters).

1. **The UNIX Filing System:** Chap. 4 [130] [183]
   Title explains the application.
2. **CAVIAR: Visitor Information System:** Chap. 5 [130] [98]
   A reasonably sophisticated company visitor and meeting (room reservation) system is developed.
3. **The IBM CICS Transaction System:** Chaps. 14–17 [130] [131]
   A number of papers outline the major legacy system reengineering of the IBM Customer Information and Control System (CICS).
4. **A Proof Assistant:** Chap. 4 [151] [182]
   The design of a proof assistant system, with theorem store, proof verification, etc., is carefully argued.
5. **Unification:** Chaps. 5, 6 [151] [97]
   Two chapters outline fundamental aspects of unification, a technique used extensively in proof systems, and in rewrite systems, including interpreters for, for example, logic programming languages.
6. **Storage:** Chaps. 7, 8 [151] [108]
   Two papers investigate heap storage and garbage collection.
7. **Graphics:** Chap. 13 [151] [170]
   Paper investigates and formalises line representations on graphics devices.
8. **A University Library System:** Chap. 3 [238] [194]
   A reasonably sophisticated library system is developed.
9. **A Radio Communications-Based Telephone Switching System:** Chap. 4 [238] [91]
   In a fascinating development, a system for radio communication-based telephony for The Philippines is developed. It involved a centralised station and some 40 (Philippine island remote) stations, time-division multiplexing (TDM), and many other technology-based hardware equipment factors. This careful, stepwise development unfolds towards an implementable system.
10. **A Ministry of Finance Information System:** Chap. 5 [238] [164]
    Developed for the Vietnam Ministry of Finance, this system involves the Taxation, the Budget and the Treasury Departments as well as all the actions within and between them: from assessment of tax bases, via the budgeting for all ministries, to the collection of taxes.
11. **Multilingual Document Processing:** Chap. 6 [238] [92]

A system is developed for processing (creating, editing, communicating and displaying) documents containing any number of scripts for any combination of the four script directions: horizontal left-to-right (say English), horizontal right-to-left (say Arabic), vertical left-to-right (say Mongol) and vertical right-to-left (say old Chinese and Japanese).

12. **Production Processes:** Chap. 7 [238] [193]
    A manufacturing system is developed, one which involves production cells, stock handling and all the related processes.
13. **Travel Planning:** Chap. 8 [238] [223]
    A reasonably sophisticated travel planning system is developed.
14. **Authentication:** Chap. 9 [238] [235]
    Some safety properties of authentication protocols are formulated and proven.
15. **Spatial Graphics:** Chap. 10 [238] [190]
    A model of (what is called) the Realm data structure and its operations is given. The Realm data structure is used in representing three-dimensional spatial data and operations on these.

1.4.5 Closing Remarks

On Programming, Engineering and Management

Most, if not all, software engineering texts and handbooks concentrate on the management aspects and on the informal, human-centered facets of programming and engineering.

We shall, in this monograph, focus on abstraction and both informal and formal modelling of systems and languages; focus on the development principles and techniques of a new kind of engineering: domain engineering; bring an altogether new focus to bear on the phases of requirements engineering and software design. We have only briefly, in Chap. 2 and Chap. 5, covered some aspects of software management. The new focus, to remind the reader, is based on all software development being initially based on extensive and serious domain engineering. The stage of domain requirements, and the stages of software design from either and all of the three domain, interface and machine requirements, highlight the new focus — as does the insistence on codeveloping both informal and formal specifications.

From this triptych view of software engineering springs a new awareness of software development management. Such management is predicated on the systematic ("light") or rigorous or even formal use of the principles and techniques of these three volumes. Traditional engineering management is predicated by laws of natural science and must consider human factors. Software engineering management, in contrast, is predicated on the more mathematical theories of computing science and the application domains, and then must consider human factors. Software development management is a fascinating area. But it is not one that we feel competent to "preach" about.

Current Software Engineering Edifices

In today's software engineering there are usually no two similar software engineering solutions to identical or near-identical problems. Software systems, from different suppliers, but for near-identical application problems usually offer significantly different user interfaces; and oftentimes vastly different ("hidden") implementations. Each such software system usually requires significant training. Users switching from one product to what ought be a similar product most often require significant retraining. Such "reusers" typically do not recognize that these distinct software products are providing near-identical solutions. As a result users become "religious" about software systems that they are using. Companies, for fear of retraining costs, when seeking new staff usually advertise that they are using "such-and-such" software products and that applicants must have the proverbial "two and a half years" prior experience with this product. I consider this a disgrace to our industry. An airline pilot with Airbus airplane flight experience can with predictable and acceptable costs be retrained for Boeing airplanes. And conversely. Many application solutions require that their users learn a whole vocabulary of concepts. Typically these vocabularies are not (theory of) domain-oriented; sometimes they are somewhat requirements-oriented; and usually they are strongly implementation-oriented. In any case such vocabularies are detrimental to the intellect of their (forced) users.

Current Software Engineering Jargon

Not all software is end-user software — in the sense of these users being people who have not been trained in IT in general. Two categories of software (packages) that can be characterised as not being end-user software (packages) are computing systems base software like database systems, compilers, multiple-user operating systems, and so on, and so-called middleware. The current jargon defines middleware software as *software that allows "front ends" like web browsers/servers or other end-user software packages to communicate with "back end" base software like database management systems (i.e., databases).*

A New View on Software Engineering

The main messages of this monograph is: The diligent reader will gain a view of software engineering which is rather different from the view we think is usually propagated by traditional textbooks. The message here is that software engineering is a highly intellectual activity. In addition to good engineering analysis, software engineering emphasises writing beautiful documents. The view is also characterised by reasoning over texts, and calculation, in the form of transformation (refinement and reification), verification, model checking and tests, calculations that take formal texts and yield formal texts. Software engineering is only to a small extent based on the natural sciences. Software engineering is primarily based on computing science and hence on the mathematical disciplines of logic, recursive function theory and modern algebra. The above view applies also when the principles and techniques of these three volumes are applied in their informal version. When applied in the formal version the message covers a spectrum of "formality": from systematic ("lightweight"), via rigorous, to (fully) formal uses of formal techniques. This view and this message is carried most forcefully by [31–33, Vols. 1 and 2]. If the reader only follows the first message (and therefore hardly deploys even the lightweight formal techniques approach), or follows either of the systematic to formal approaches and then finds, after having studied these volumes, that her view on software engineering has changed accordingly, then the author has achieved a main objective.

2

Possible Collaborative Domain Projects[1]

A Management Brief

- JAIST/DEDR[2] has some interesting things to offer.
- It is in the DOMAIN ENGINEERING area of software engineering:
  - ⋆ the orderly,
  - ⋆ manageable, and
  - ⋆ believable

  development of
  - ⋆ trustworthy,
  - ⋆ dependable
  - ⋆ software

  — on time, at cost — the right software:
  - ⋆ verifiably correct and
  - ⋆ and customer-validated.
- This chapter outlines some ideas on
  - ⋆ possible joint collaboration
  - ⋆ centered around the first phase of a triptych of software engineering
    - ◇ DOMAIN ENGINEERING,
    - ◇ requirements engineering and
    - ◇ software design;
  - ⋆ and based on using formal approaches
    - ◇ that allow careful interactive analysis
    - ◇ validation
    - ◇ and verification

    of designs.
- There is another JAIST/DEDR supporting document. Chapter 1:
  - ⋆ *On Domains and Domain Engineering*
    Prerequisites for Trustworthy Software
    A Necessity for Believable Project Management

  It provides substantial evidence.

Abstract

In this note we suggest a number of alternative, collaborative projects within the broader research and engineering area of domain theory and domain engineering.

The common denominators are: The collaborators (i.e., partners), including the proposing JAIST/DEDR Group, each have their own vested interest in one or another facet of the proposed joint topic of R&D; the

[1]This is an edited version of [29]. Presented, together with [28], at a number of meetings with Japanese Software and IT industry leaders during the Spring of 2006.

[2]DEDR: Domain Engineering and Digital Rights, a JAIST COE Project: http://www.ldl.jaist.ac.jp/drcp/

goal of the project can thus be defined both as the sum of the goals of each partner as well as the sum-total of their working together: namely an increased, mutually beneficial awareness of engineering by academia, and an increased awareness of academic research approaches and results by industry.

Expected outcome of the joint R&D are: (i) Sizable precise descriptions (in English and Japanese) of selected (i.e., chosen) domains, their formalisation and analysis; (ii) example software package or sub-system development for selected applications and related to the chosen domain descriptions; (iii) and increased awareness in the Japanese IT community of the benefits of strict domain engineering, related requirements engineering and trustworthy software.

2.1 Background

The background for this proposal is the emergence of a new technically sound and scientifically fascinating approach to software development: From domain models via requirements to software design. This approach is illustrated especially by Item 3, [33], in the series:

1. *Software Engineering, Vol. 1: Abstraction and Modelling.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
2. *Software Engineering, Vol. 2: Specification of Systems and Languages.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
3. *Software Engineering, Vol. 3: Domains, Requirements and Software Design.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

We also refer to Chap. 1:

4. *Domains and Domain Engineering. Prerequisites for Trustworthy Software. A Necessity for Believable Project Management.*

2.2 Prior Evidence

Needless to say, we would not put forward this document unless we represent some strong previous and successful projects of the kind mentioned in the next section. Our reference list includes:

2.2.1 Ministry of Finance, Vietnam

MoFIT/UNU-IIST

Over a three year period in the mid-1990s UNU-IIST, the UN University's International Institute for Software Technology, www.iist.unu.edu, collaborated with the Vietnamese Ministry of Finance on domain engineering, requirements capturing and suggesting a computing system architecture for the problem described in the next section. That problem is formulated generically.

General

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

Now the main functions of a ministry of finance wrt. its own taxation department, the budget departments (of various ministries and of the ministry of finance) and that ministry's treasury department are as follows:

- **Assessment:** Annually an order is issued — by the ministry of finance taxation department — whereby the corresponding taxation departments of each province (of the state), each district within each province (of the state), and each commune within each district (etc., etc.) are to assemble, gather, obtain, by census or otherwise, statistical data, that is "the assessment data". These data represent "best guesses" of the basis for tax revenue (such as personal income, sales (for sales tax purposes), fees (for services rendered by province, district or commune authorities), etc.). From the communes this kind of data is communicated (perhaps in simplified, summary form) to the district of that commune, and likewise from district to province, and to state. These communications must take place before certain dates ($D_{a_{c\to d}}, D_{a_{d\to p}}, D_{a_{p\to s}}$).
- **Budgeting:** More or less simultaneously an order is issued — by the budget department of the ministry of finance — whereby each ministry (m_μ, incl. the ministry of finance, m_f) is to set up a budget, B_{m_μ}, for next year's activities (i.e., expenditures) E_{m_μ}. The ministry of finance sets an initial ceiling I_{m_μ} (of so many millions of, say, dollars) for respective ministries' expected incomes.

  The various ministries submit their (possibly negotiated) budgets for next year to the ministry of finance by a certain date $D_{\to m}$. A twist to this budgeting process may occur if the ministry of finance judges, well before $D_{\to m}$, but after $D_{a_{p\to s}}$, that the assessment data warrants either a downward (pessimistic), or an upward (optimistic), adjustment of the income I_{m_μ}. The submitted budget B_{m_μ} must balance within the possibly adjusted income ceiling I_{m_μ}. The various ministries also have "shadow" budget departments in each province, district and commune.

- **National Budget Enactment:** The parliament then negotiates and eventually, in time for the next year, passes the national budget, B_s, as assembled from all ministries' individual budgets B_{m_μ}.

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

References

- Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. *Developing a Financial Information System.* Technical Report 81, UNU-IIST, P.O.Box 3058, Macau, September 1996. `http://www.iist.unu.edu/newrh/III/1/docs/techreports/report81.pdf`

  Abstract: This document describes the work done in the UNU/IIST MoFIT project during the period April–September 1996 by five Fellows from Vietnam (four from the Ministry of Finance, one from the Institute of Information Technology). The eventual aim of the project is to describe a complete financial information system. The first part of the project concentrated on the taxation system, the Vietnamese Government's main revenue collecting system. It includes a domain analysis in two parts, an informal narrative and a formal model; a prototype of part of that system developed from the formal specification and used to test it; a description of the security aspects of the system; an extension of the formal model describing the security aspects; and a description of taxation policies, particularly those likely to change in the immediate future. The formal components are written in the RAISE specification language, RSL using the RAISE development method.
- Do Tien Dung, Chris George, Hoang Xuan Huan, and Phung Phuong Nam. *A Financial Information System.* Technical Report 115, UNU-IIST, P.O.Box 3058, Macau, July 1997. `http://www.iist.unu.edu/newrh/III/1/docs/techreports/report115.pdf`[3]

  Abstract: In this document we continue the work started in "Developing

[3]Partly published in *Requirements Targeting Software and Systems Engineering*, LNCS 1526, Springer-Verlag, 1998.

a Financial Information System", UNU/IIST Research Report 81. That document concentrated on the taxation system and on its detailed development. Here we take a broader view, sketching the taxation system and also the budget, treasury and external aid and loan systems. Then we show how these may all be combined, allowing not only "vertical" communication within each system but also "horizontal" communication between components of different systems. We thus provide a top-level specification of a national financial information system.

2.2.2 Railway Computing Systems, China

With the Chinese Ministry of Railways UNU-IIST, over a four year period, 1993–1996, carried out a domain–requirements–software design study for a train running map system for the ZhengZhou–WuHan north–south artery. A train running map is a two dimensional diagram. Horizontal lines designate train stations. The horizontal, say the t, dimension denote time. The vertical dimension denote lines between stations such that if two adjacent horizontal lines designate stations s_i and s_j, then the space between these horizontal lines denotes the rail line(s) between s_i and s_j. Let us assume s_i above s_j. A train traveling from s_i and s_j is now denoted by a specific slanting, i.e., a diagonal line, "the train", proceeding from "north–west" to "south-east". "The train" traveling from s_j and s_i is therefore denoted by a specific slanting, i.e., a diagonal line proceeding from "south-west" to "north-east". Steeper diagonal lines denote faster trains. Usually a running map depicts many trains. In the period t_0 (leftmost edge) to t_N (rightmost edge) there might typically be 30-40 trains in either direction. The map is "crowded". A delayed (too fast) train shows up skewed, with a slower (steeper) gradient than its scheduled diagonal. If there are n crossing train diagonal lines between adjacent stations then there must be n parallel tracks (somewhere) between these stations. Rescheduling a train means to both move the scheduled diagonal to the right and to change its gradient. When moving "the train" its gradient may cross other train gradients on the line — which may not be feasible if it is a single line. Hence a running map system is a rescheduling system for quickly moving "trains" around subject to rules & regulations.

We refer to a small collection of mostly UNU-IIST-related documents and publications [13, 15, 17, 20–24, 49, 51, 53, 57, 93, 102, 103, 162, 163, 181, 191, 204, 208, 209, 227, 234, 244, 246, 249].

2.2.3 Radio Communications, The Philippines

With the Philippine government's Advanced Science and Technology Institute (ASTI), UNU-IIST carried out a joint R&D project on a radio communications based telephone system 1994–1996. It was subsequently completed by ASTI in Manila.

Simplifying — but see the reference below — the system was built up around a central radio station and a number of (40) local, inexpensive radio stations (placed in the vast island country of The Philippines). Think of the 1+40 stations organised as a simple one level tree, the root is the central station, the immediate subtree leaves are the 40 local stations. Order these 1–40. Number the root 0. Now communication is like a conveyour belt that winds its way from the root to the first station and back, then from the root to the second station, and back, and so on, from the root to the 40th station and back, and then all over again: $0-1-0-2-0-3-0\cdots\ 0-39-0-40-0-1-0-2-0\cdots$. Telephone calls are now digitally time-multiplexed so that a call from any subscriber (attached to some station i (0..40)) to some other subscribed (attached to some station j (0..40)) is chopped into "zillions" of small packages and placed on the conveyour belt suitably marked with sender and receiver information. The project was then to specify this domain: the equipment, the arrangement of telephone messages, their dial-up and connection, etc., and then to establish requirements to a dependable software control system and its design.

- Roderick Durmiendo and Chris George. *Formal Development of a Digital Mutiplexed Radio-Telephone System.* Research Report 67, UNU-IIST, P.O.Box 3058, Macau, Feb 1996. `http://www.iist.unu.edu/newrh/-III/1/docs/techreports/report67.ps.gz`
  Abstract: This paper presents a formal development of a Radio Telephone System by a sequence of correctness-preserving refinements. We follow several steps of refinement from an abstract applicative specification which is validated against the properties and behavior of a basic telephone service, to a specification involving a central station and a number of remote stations communicating synchronously by means of radio channels. Particular features of the development are the decomposition of the basic telephone service into separate layers for the phones and the communication network, and the introduction of finite communication resources. We verify that the decompositions preserve correctness and that the resources are allocated and released correctly. The work was carried out with the RAISE specification language and its associated method, using the RAISE tools.

2.3 Possible Project Topics

As can also be gleaned from Item 4 on page 40 we suggest joint R&D projects around domains, requirements and advanced software for either one or more of the following areas, with one or more selected areas giving rise typically one collaborative project:

2.3.1 Administrative Forms Processing

Public and private institutions, enterprises, businesses, industries, are becoming increasingly dependent on semi-automated document handling systems. These systems have grown piecemeal, from simple document text processing to "office" products, etc. Many issues of generality, of safe and secure transfer of documents, etc., are given short shrift[4]. It is about time to reconsider the issues from basic principles.

Topics: Documents of all kinds and forms dominate the public and private administration. Templates serve as the basis for filling-in applications, surveys, questionnaires, invoices, budgets, accounts, and other forms of simple documents. Meta-templates serve to aggregate simple documents into aggregate documents (transacted invoices into accounts, merging accounts and budgets into audits, etc.). Document handling further involves such issues as who maintains (keeps) masters, who is allowed to edit and copy documents, documents only being accessible to properly authorised persons. Documents may also belong to the so-called digital rights domain: video clips, photos, music, movies, books, etc.

Of interest to: Public and private administrators and administrations, hospitals, banks, insurance companies, taxes and excise departments of the ministry of finance, etc., etc., and the providers of IT services, equipment and software for this infrastructure segment of society.

Goals: There are several concurrent goals: (i) To develop a common theory of documents, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of document handling facet (creation, editing, copying, distribution, shredding) and also protection (copyrights (digital rights management), access authorisation (security), and secure availability). (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for document handling.

We refer to Chap. 7.

2.3.2 Air Traffic

Aircrafts cruise the air lanes. Air traffic becomes increasingly denser. Air traffic control (ATC) becomes increasingly more serious.

Topics: We propose to domain model the aircraft airspace of airports, air lanes, and air traffic control centers (terminal towers, approach towers, regional and continental control centers) — i.e., of their entities, functions,

[4]Merriam-Webster defines 'short shrift' by: barely adequate time for confession before execution, little or no attention or consideration, quick work – usually used in the phrase make short shrift of

events and behaviours. Included in this modelling is also the modelling of the supporting technologies (radar, tactical collision avoidance systems, etc.), the management and operation of air space, the (ICAO and local aviation authority) rules and regulations, their scripts, and the spectrum of from diligent via sloppy and delinquent to outright criminal behaviour of humans acting in the air traffic domain (pilots, controllers, etc.).

Of interest to: Civil aviation authorities, airlines, air traffic controllers, airports, pilots, passengers, etc. — as well as to the aircraft and the IT + software industry providers of technology.

Goals: There are several concurrent goals: (i) To develop a common theory of air traffic, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of air traffic control. (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for air traffic control.

We refer to:

5. See Appendix Sect. A.1.1 on page 332.
6. Dines Bjørner: *Software Systems Engineering — From Domain Analysis to Requirements Capture: An Air Traffic Control Example. 2nd Asia-Pacific Software Engineering Conference (APSEC '95)*, Brisbane, Queensland, Australia (IEEE Computer Society, 1995)
7. Kristian Kalsing: *Specification of Air Traffic Control*[5]. Software Verification Research Centre, School of Information Technology, The University of Queensland, Australia, October 3, 1999.

2.3.3 Airports

Increasingly, around the world, air traffic grows by "leaps and bounds". Airports are becoming congested. New airports are being rapidly built. China expects, according to industry sources, to open one new airport every month for the next 10 years! Airport management, including the training of airport service and support staff becomes acute.

Topics: An airport can be seen as a work flow system where people (passengers, aircraft crews, gate staff, etc.), aircraft, aircraft supplies (luggage, catering, gasoline), aircraft cleaning, aircraft mechanical etc. check, information (tickets, boarding cards, luggage tabs, passenger lists, revised timetables, etc.), etc., flow and interact. We propose to domain model airports, including the entities of an airport, the functions, the events and the behaviours. Included in this modelling is also the modelling of the supporting technologies (check-in counters, boarding card machines, luggage

[5]http://www.ldl.jaist.ac.jp/dedr/airtrafficcontrol.ps

tab machines, baggage conveyor belts, etc.), the management and operation of airports, the spectrum of from diligent via sloppy and delinquent to outright criminal behaviour of humans acting in the airport domain (passengers, gate staff, etc.).

Of interest to: Passengers, airlines, airports, civil aviation authorities, aircraft crews, suppliers (catering services, cleaning services, etc.), the software houses providing airport software packages, etc.

Goals: There are several concurrent goals: (i) To develop a common theory of airports, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of airport operation. (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for airports.

8. Anders Dinesen and Ibrahim Alameddine: *Towards Domains, Requirements, Software Design. Descriptions for Airport Management Applications*[6]. This is an undergraduate student term project (1/5'th weekly load) Dept. of IT, Technical University of Denmark, August 8, 2000

2.3.4 Financial Service Industry

E-Banking, the handling of insurance payments and claims processing and buying and selling securities instruments (stocks, bonds, etc.) over the Internet, etc., calls for an overhaul of our understanding of the whole financial service industry.

Topics: We propose to comprehensively domain model not only the individual transactions within banks, insurance companies, stock (etc.) brokers and traders, the stock exchanges, portfolio management, credit card companies, etc., but more importantly, on the basis of sufficiently detailed models of the former, to domain model, on one hand, the interactions between these "players", between banks, between insurance companies, etc., and between banks and insurance companies, between banks and securities instrument brokers and traders, between banks and credit card companies, etc., and, on the other hand, between "the market": consumers, retailers, wholesalers, producers, distributors and banks, etc., etc.

Of interest to: (i) Each and everyone of all the commercial players of the financial service industry: banks, insurance companies, stock (etc.) brokers and traders, the stock exchanges, portfolio management, credit card companies, etc., (ii) private citizens (the users, the clients, customers, of these services), (iii) "the market" (retailers, wholesalers, producers, distributors), (iv) public and private regulatory agencies (state and federal savings & loan regulatory agencies, federal and state exchange commissions,

[6]http://www.ldl.jaist.ac.jp/dedr/airport.ps

etc., etc.), involved ministries (finance, trade, industry, citizens protection, etc.), "the public at larger", and politicians (eager to profile themselves as champions of either industry or consumers, "or both"!), and the software houses providing financial software packages, etc.

Goals: There are several concurrent goals: (i) To develop a common theory of the financial service industry, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of financial service transactions (or other). (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for the financial service industry.

2.3.5 Health Care

Health costs are soaring. Freedom of choice with respect to selecting private physician, clinic and hospital is spreading, specialisation of treatment and treatment places (locations), etc., and all this within a framework of increased differentiation of and collaboration between private and public health insurance.

Topics: We propose to comprehensively domain model the flow of people, information, material, and monitoring & control within the health care sector at large: from citizens (healthy or sick), via private physicians, treatment clinics, hospitals, pharmacies, the pharmaceutical industry, providers of health care equipment, etc., to the national boards of health, the ministries of health, etc. Special models, embedded within the larger model, might focus on (1) patient medical records (in preparation for electronic patient journals, EPJ)[7], (2) hospitalisation plans, (3) the interaction between analytical instruments (X-Ray machines, CTM (computer tomography machines) and MRS (magnetic resonance scanners), etc.) and patient medical records (cum EPJ), etc.

Of interest to: Patients, medical professionals, pharmacists, clinics, hospitals, national boards of health, etc., the pharmaceutical industry, ministry of health, the software houses providing health care software packages, etc.

Goals: There are several concurrent goals: (i) To develop a common theory of the health care, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of health care transactions (or other). (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for the health care sector.

[7]See also project proposal 1: Administrative Forms Processing.

9. Dines Bjørner: *Domain Modelling some Healthcare Sector Concepts*[8]. Vastly incomplete internal draft report. Dept. of IT, Technical University of Denmark, September 14, 2000

2.3.6 Manufacturing

Agile manufacturing, the ability to "turn around" and respond quickly to new or changed production orders, including the production of systems involving many co-ordinated producers, is becoming an everyday issue.

Topics: We propose to comprehensively domain model the flow of people, information, material, and monitoring & control within the manufacturing companies and between these, as well as between these and suppliers of product parts (incl. raw materials) and consumers of products, and also the related supply chain of delivery services. More specifically we propose to model manufacturing floors (of loosely or tightly coordinated machines, conveyour belts or delivery fork lifts, etc., and their interfaces to the supply and end-product warehouses), order processing departments, etc., etc. As part of requirements for agile manufacturing we propose to model the coordination (by agents and brokers) of how orders for agile production of complex systems are resolvable through collaboration between otherwise competing manufacturers.

Of interest to: The manufacturing industry in terms of individual manufacturers and the industry as a whole (Keidanren[9] and METI[10]), the distribution (trucking) companies, industry research centres in industry, at universities, and at government level, and the software houses providing manufacturing software packages.

Goals: There are several concurrent goals: (i) To develop a common theory of the manufacturing industry, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of manufacturing company and/or industry transactions (or other). (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for the manufacturing industry.

We refer to Appendix C.

2.3.7 "The Market"

The concept of e-market is alluring. We all transact simple purchases over the Internet: buying airline tickets, books, records, movie DVDs, etc. We are also

[8]http://www.ldl.jaist.ac.jp/dedr/healthcare.ps

[9]http://www.keidanren.or.jp/

[10]http://www.meti.go.jp/english/

beginning to acquire, rent, or otherwise, music and movies: paying for their rendering on suitable devices in our possession. A sizable variety of software packages are offered. But do also these packages together constitute or reflect a proper understanding of the market?

Topics: We propose to comprehensively domain model the market in terms of consumers, retailers, wholesalers, producers, distribution services and the interface to credit and bank card payment services. Included in such a comprehensive model is the modelling of functions like inquiring as to what is available, offering "deals", submitting and accepting orders, sending, accepting, invoicing, paying, rejecting, and "repairing" purchased merchandise (between buyers [consumers, retailers, wholesalers] and sellers [retailers, wholesalers, producers]). Included is also the modelling of agents acting on behalf of potential buyers or sellers, and brokers acting on behalf of potential buyers and sellers. Auctioning and the management of digital rights licenses and their use are yet further matters that, together with the previous functionalities illustrate the depth and breadth of "the market".

Of interest to: Consumers, retailers, wholesalers, producers, distribution services, credit card companies, banks, market (fair trade) associations, consumer protection organisations, ministry of trade, the software houses providing e-market systems, etc.

Goals: There are several concurrent goals: (i) To develop a common theory of "the market", a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of market transactions (or other). (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for "the e-market".

We refer to Appendix E and to

10. Dines Bjørner: *Domain Models of "The Market" — in Preparation for E-Transaction Systems*[11]. In: Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski) Kluwer Academic Press, The Netherlands, December 2002

2.3.8 Transportation

Transportation, as financial services and health care, count as the prime infrastructure components whose quality strongly influence a country's or a region's welfare. Transportation takes many forms, i.e., there are many sub-infrastructures that can each be tackled more or less separately — and some of these will be covered in Sect. 2.3.8[1] (Page 51), Sect. 2.3.8[3] (Page 52) and

[11] http://www.ldl.jaist.ac.jp/dedr/themarket.ps

Appendix B. But we can also speak of the generic domain of transportation: transportation nets and traffic.

Topics: We propose to model the entities, functions, events and behaviours of transportation nets and traffic, that is, of multi-modal segments (roads, rail lines, air lanes, shipping lanes) and multi-modal junctions (street intersections, railway stations, airports, harbours), their composition into multi-modal nets, the projection of multi-modal nets onto single modality nets (road nets, rail nets, air lane nets, shipping lane nets), the functions of enlarging, reducing or "repairing" (incl., maintaining) segments, junctions and sub-nets, the events of segments, junctions and sub-nets (impassable, closed, disconnected, etc.), and the behaviours of nets and traffic. Included in this overall model is the modelling of support technologies (such as traffic monitoring and control [junction semaphores, railway line signals, etc.], etc.), management & organisation (of segment and junction maintenance, traffic, etc.), and rules & regulations (and their scripts, for net maintenance, traffic, etc.).

Of interest to: Any of the stake holders in any of the road, rail, air traffic or shipping domains.

Goals: There are several concurrent goals: (i) To develop a common, generic theory of transportation, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ; (ii) to serve as a basis for developing requirements for software that may be common to, i.e., shared by the domain models covered in Chap. 7; and (iii) to otherwise serve as a common reference point for the domain models covered in Sect. 2.3.8[1] (Page 51), Sect. 2.3.8[3] (Page 52) and Appendix B

We refer to Appendix B.

[1] Container Shipping

Topics: We propose to model the entities, functions, events and behaviours of container shipping: containers, container ships, container (harbour) terminals, the stowage of containers aboard ships and in (harbour or port) terminal pool areas, the processing of shipping requests (bills of lading, way bills),

Of interest to: Container shipping lines, container terminal ports, removal companies arranging for the transport of goods, as well as software houses and operations research companies providing IT and logistics consultancy and software support.

Goals: There are several concurrent goals: (i) To develop a common theory of container logistics, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of containers transactions (order processing, stowage, etc.). (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for container shipping.

See

11. Dines Bjørner: A Container Line Industry Domain, a 90 page report: http://www2.imm.dtu.dk/~db/container-paper.pdf

[2] Road Nets and Traffic

See Appendix B.

[3] Railways

Train traffic, in Europe, China, Japan and the United States is considered a main provider of overland freight transport and, still in many places, passenger transport. Road congestion, on free and toll ways are such that, with the above, the demands on moving certain forms of transport away from roads and onto rails is socially increasing.

Topics: We propose to model the entities, functions, events and behaviours of railway systems: lines with their signalling and stations with their intricate rail arrangements (leading to issues of interlocking), the monitoring and control of train traffic, the setting of signals and switches, and so on. Included in these models are models of supporting technologies reflecting real time embedded systems (signals, interlocking, etc.), management & organisation, rules & regulations, etcetera.

Of interest to: Railway infrastructure owners and operators, train operators, passengers, freightors, regulatory agencies, etc., as well as software houses and operations research companies providing IT and logistics consultancy and software support.

Goals: To develop a common theory of railways, a theory manifested in the form of well-written narrative descriptions in Japanese (and English) as well as formalised in for example CafeOBJ. (ii) To develop portable software modules that handle one or another kind of railway planning and operations. (iii) Possible industry standardisation proposals. (iv) New electronic (incl. mechatronics) "gadgets" for railways.

See Appendix B, [13,15,17,20–24,49,51,53,57,93,102,103,162,163,181,191, 204,208,209,227,234,244,246,249] and

12. Dines Bjørner: Towards a TRain Book, This document suggests a number of domain models for a variety of railway facets. It is part of an ongoing domain theory "Grand Challenge" effort: TRain: `http://www.railway-domain.org/thetrainbook.ps.`

2.4 Project Modalities

How do we foresee these projects being

- formulated,
- funded,
- carried out, and
- propagated?

Well, that's what this chapter "is all about"! Let us just suggest a few ideas concerning how possible joint projects may be carried out:

- Group(s) at the industrial (or institutional) partner(s) work[s] on their application domain specific requirements and software designs — all the while regularly communicating requirements and assumptions about the domain to the other partners.
- The DEDR Group at JAIST works on the domain model:
  - ⋆ carefully expressing
    - ◇ narratives and
    - ◇ terminologies (including ontologies)
    - ◇ in both Japanese and English;
  - ⋆ carefully formalising this domain model in both
    - ◇ CafeOBJ and
    - ◇ RAISE and/or VDM;

    and
  - ⋆ carefully
    - ◇ analysing and
    - ◇ verifying properties, i.e.,
    - ◇ propositions, lemmas and theorems of the formalisations.
- The industry (institutional) and JAIST partners meet regularly, it is suggested, typically once every 12 weeks[12], for three–four days, presenting their work to each other, discussing shortcomings, improvements and possibly revise project plans.
  - ⋆ This mode of operation was and is typical of the European Community ESPRIT and Framework Programme projects.
- Each project period is suggested to be 18 months.
- The joint project is suggested reviewed yearly, 15 months into each period.
- A full project is suggested to last for 2–3 periods.
- The project is suggested to yearly *propagate* its work at open, three day workshops.

[12]Thus there are 11 full working weeks between the weeks where the groups meet.

Part II

A Science & Engineering of Domain Models

3

The Rôle of Domain Engineering in Software Development[1]

Abstract

We outline the concept of domain engineering and explain the main stages of developing domain models. Requirements engineering is then seen as an intermediate stage where domain models are "transformed" into requirements prescriptions. Software Design concludes development — and we comment on software correctness with respect to both requirements prescriptions and domain descriptions. We finally overview this new phase of development: domain engineering and argues its engineering virtues while relating them to object-orientedness, UML, component-based SE, aspect-orientedness and intentional software development.

3.1 Introduction

3.1.1 Triptych Dogma

Traditionally, today, software development starts with expressing requirements and then goes on to design software from the requirements. In this paper we shall explain why this is not good enough. First we express the triptych dogma: *Before software can be developed we must understand its requirements. Before requirements can be expressed we must understand the domain in which the software (plus the hardware) is to reside. Therefore we must first develop an understanding of that domain.*

3.1.2 Triptych of Software Development

We therefore develop software as follows: First we develop a domain description. Then, from the domain description, we develop a requirements prescrip-

[1]This is an edited version of [37]. Presented at the October 2006 meeting of the IPSJ (Information Processing Society of Japan) Software Engineering Symposium 2006, Tokyo.

tion. And, finally, from the requirements prescription we develop a software design. While developing we verify and validate the domain description, verify and validate the requirements prescription with respect to the domain description and requirements stakeholder statements, and verify the software design with respect to the requirements and domain specifications.

3.2 An Example: Railway Nets

Before we delve into too much "talking about" domain descriptions let us show a tiny example. The example covers a description of just a small part of a domain: the net of rails of a railway system. There are only two parts to the description: A systematic, "tight", precise English narrative, and "its" corresponding formalsation in the specification language, RSL of RAISE [31–33,44,101,104,106]. We do not show "all the work" that precedes establishing this description.

3.2.1 Narrative

1. A railway net is a net of mode railway.
2. Its segments are lines of mode railway.
3. Its junctions are stations of mode railway.
4. A railway net consists of one or more lines and two or more stations.
5. A railway net consists of rail units.
6. A line is a linear sequence of one or more linear rail units.
7. The rail units of a line must be rail units of the railway net of the line.
8. A station is a set of one or more rail units.
9. The rail units of a station must be rail units of the railway net of the station.
10. No two distinct lines and/or stations of a railway net share rail units.
11. A station consists of one or more tracks.
12. A track is a linear sequence of one or more linear rail units.
13. No two distinct tracks share rail units.
14. The rail units of a track must be rail units of the station (of that track).
15. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.
16. A rail unit has one or more connectors.
17. A linear rail unit has two distinct connectors. A switch (a point) rail unit has three distinct connectors. Crossover rail units have four distinct connectors (whether simple or switchable), etc.
18. For every connector there are at most two rail units which have that connector in common.
19. Every line of a railway net is connected to exactly two distinct stations of that railway net.
20. A linear sequence of (linear) rail units is an acyclic sequence of linear units such that neighbouring units share connectors.

3.2.2 Formalisation

```
type
1.  RN  = {| n:smN • obs_M(n)=railway |}
2.  LI  = {| s:S • obs_M(s)=railway |}
3.  ST  = {| c:C • obs_M(c)=railway |}
    Tr, U, K

value
    4.   obs_LIs: RN → LI-set
    4.   obs_STs: RN → ST-set
    5.   obs_Us: RN → U-set
    6.   obs_Us: LI → U-set
    8.   obs_Us: ST → U-set
    11.  obs_Trs: ST → Tr-set
    15.  is_Linear: U → Bool
    15.  is_Switch: U → Bool
    15.  is_Simple_Crossover: U → Bool
    15.  is_Switchable_Crossover: U → Bool
    16.  obs_Ks: U → K-set

    20.  lin_seq: U-set → Bool
     lin_seq(us) ≡
        ∀ u:U • u ∈ us ⇒ is_Linear(u) ∧
        ∃ q:U* • len q = card us ∧ elems q = us ∧
           ∀ i:Nat • {i,i+1} ⊆ inds q ⇒ ∃ k:K •
              obs_Ks(q(i)) ∩ obs_Ks(q(i+1)) = {k} ∧
           len q > 1 ⇒ obs_Ks(q(i)) ∩ obs_Ks(q(len q)) = {}
axiom
4. ∀ n:RN • card obs_LIs(n) ≥ 1 ∧ card obs_STs(n) ≥ 2

6. ∀ n:RN, l:LI • l ∈ obs_LIs(n) ⇒ lin_seq(l)

7. ∀ n:RN, l:LI • l ∈ obs_LIs(n) ⇒ obs_Us(l) ⊆ obs_Us(n)

8. ∀ n:RN, s:ST • s ∈ obs_STs(n) ⇒ card obs_Us(s) ≥ 1

9. ∀ n:RN, s:ST • s ∈ obs_LIs(n) ⇒ obs_Us(s) ⊆ obs_Us(n)

10.  ∀ n:RN,l,l′:LI • {l,l′}⊆obs_LIs(n)∧l≠l′⇒obs_Us(l)∩ obs_Us(l′)={}

10.  ∀ n:RN,l:LI,s:ST •
           l ∈ obs_LIs(n)∧ s ∈ obs_STs(n)⇒obs_Us(l)∩ obs_Us(s)={}

10.  ∀ n:RN,s,s′:ST • {s,s′}⊆obs_STs(n)∧s≠s′ ⇒
```

```
          obs_Us(s)∩ obs_Us(s′)={}

11.  ∀ s:ST•card obs_Trs(s)≥1

12.  ∀ n:RN,s:ST,t:Tr •
          s ∈ obs_STs(n)∧t ∈ obs_Trs(s)⇒lin_seq(t)

13.  ∀ n:RN,s:ST,t,t′:Tr •
          s ∈ obs_STs(n)∧{t,t′}⊆obs_Trs(s)∧t≠t′ ⇒
             obs_Us(t) ∩ obs_Us(t′) = {}

18. ∀ n:RN • ∀ k:K •
         k ∈ ∪{obs_Ks(u)|u:U•u ∈ obs_Us(n)} ⇒
            card{u|u:U•u ∈ obs_Us(n)∧k ∈ obs_Ks(u)}≤2

19. ∀ n:RN,l:LI •  l ∈ obs_LIs(n) ⇒
     ∃ s,s′:ST • {s,s′} ⊆ obs_STs(n) ∧ s≠s′ ⇒
       let sus=obs_Us(s),sus′=obs_Us(s′),lus=obs_Us(l) in
          ∃ u,u′,u″,u‴:U • u ∈ sus ∧
             u′ ∈ sus′ ∧ {u″,u‴} ⊆ lus ⇒
             let sks=obs_Ks(u),sks′=obs_Ks(u′),
                 lks=obs_Ks(u″),lks′=obs_Ks(u‴) in
        ∃!k,k′:K•k≠k′∧sks ∩ lks={k}∧sks′ ∩ lks′={k′}
          end end
```

3.2.3 References

We can refer to more complete descriptions of railway domains: www.railway-domain.org. There are some publications: [18, 20, 52, 53] and a "book": "The TRain Book": http://www.railwaydomain.org/book.ps. See also Appendix B.

3.3 Domains

3.3.1 Examples of Domains

There are basically three kinds of domains, sometimes called application domains or business domains. These are: base systems software such as compilers, operating systems, database management systems, data communication systems, etc.; "middleware" software packages: Web servers, word/text processing systems, etc.; and the real end-user applications. That is, software for airlines and airports; banks and insurance companies; hospitals and healthcare in general; manufacturing; the market: consumers, retailers, wholesaler, the distribution chain; railways; securities trading: exchanges, traders and brokers; and so forth.

3.3.2 Domain Description

What Is a Domain Description

What do we mean by a domain description? By a domain description we mean a document, or a set of documents which describe a domain as it is, with no references to, i.e., with no implicit requirements to, software. The informal language part of a domain description is such that a stakeholder of that domain recognizes that it is a faithful description of the domain. So, a domain description describes something real, something existing. Usually a domain description describes not just a specific instance of a domain, but a set of such, not just one bank but a set of "all" banks!

How Is a Domain Description Expressed?

How is a domain description expressed? By a domain description we mean any text that clearly designates an phenomena, an entity, or a function (which when applied to some entities become an action), or an event, or a behaviour (i.e., a sequence of actions and events) of the domain, or a concept defined, i.e., abstracted from other domain descriptions.

Domain Descriptions Are Indicative

Domain descriptions described what there is, the domain as it is, not as the stakeholder would like it to be.

Informal and Formal Domain Descriptions

Domain descriptions come in four, mutually supportive forms, three informal texts and one formal: (i) rough sketches are informal, incomplete and perhaps not very well structured descriptions; (ii) terminologies — explaining all terms: names of phenomena or concepts of the domain; (iii) narratives — "tell the story", in careful national/natural and professional language; and (iv) formal specification — formalising in mathematics the narrative and provides the ultimate answer to questions of interpretation of the informal texts. Initial descriptions necessarily are rough sketches. They help us structure our thinking and generate entries for the terminology. Terminologies, narratives and formalisations are deliverables.

Existing Descriptions

Are there accessible examples of domain descriptions? Yes, there are descriptions now of railway systems, transportation nets, financial service industries, hospital healthcare, airports, air traffic, and many other domains. Some are in the form of MSc theses, some are part of PhD theses. Some fragment domain descriptions are published in journal papers, some in conference papers. And several are proprietary — having been developed in software houses. For all the cases implied above the descriptions include formal descriptions.

3.3.3 Domain Engineering

How to Construct a Domain Description?

In the following we will briefly outline the steps — domain stakeholder identification, domain acquisition, domain analysis, domain modelling, domain verification and domain validation — that it takes to construct a domain description.

Domain Stakeholders

All relevant stakeholders must be identified. For, say a railway domain, typical stakeholder groups are: *the owners of a railway; the executive, strategic, tactical and operational management — that is several groups; the railway ("blue collar") workers — station staff, train staff, line staff, maintenance staff, etc.; potential and actual passengers and relatives of these; suppliers of goods and services to the railway; railway regulatory authorities; the ministry of transport; and politicians "at large".* Liaison with representatives of these stakeholder groups must be regular — as some, later, become requirements stakeholders.

Domain Acquisition

The domain engineer need acquire information ("knowledge") about the domain. This should be done pursuing many different approaches. The two most important seems to be: (i) reading literature, books, pamphlets, Internet information, about the domain, and (ii) eliciting hopefully commensurate information from stakeholders. From the former the domain engineer is (hopefully) able to formulate a reasonable questionnaire. Elicitation is then based on distributing and the domain engineers personally "negotiating" the questionnaire with all relevant stakeholders. The result of the latter is a set of possibly thousands of domain description units.

Domain Analysis

Description Unit Attributes

The domain description units are then subjected to an analysis. First they must be annotated with attribute designators, (i) ontological such as entity, function, event, behaviour; (ii) facets such as intrinsics, support technology, management & organisation, rules & regulation, script, human behaviour; and (iii) administrative such as source of information, date, time, locations, who acquired, etc.: etcetera.

Problems

Analysis of the description units involve looking for and resolving incompleteness, inconsistency and conflicts.

Concepts

Analysis of the description units primarily aims at discovering concepts, that is, notions that generalise a class of phenomena, and for discovering meta-concepts, that is “high level” abstractions that together might help develop as generic and hence, it is believed, applicable, reusable domain model as possible.

Domain Modelling Proper

Domain modelling is then based on the most likely database-handled domain description units. The domain model, that is, the meaning of the domain description must capture: (i) intrinsics: that which is at the basis of, or common to all facets, (ii) technologies which support phenomena of the domain, (iii) management & organisation: who does what, who reports to whom, etc., (iv) rules & regulations — governing human behaviour and use of technologies — (v) sometimes manifested in scripts, and (vi) human behaviour — diligent, sloppy, delinquent or outright criminal. It must all be described!

Domain Verification

Verification — only feasible when a formal description is available — proves properties of the domain model not explicitly expressed, and serves to ensure that we got the model right.

Domain Validation

Validation is the human process of “clearing” with all relevant stakeholders that we got the right model [64].

Discussion

Thus domain engineering is a highly professional discipline. It requires many talents: interacting with stakeholders, ability to write beautifully and concisely, ability to formalise and analyse formal specifications, etc. Domain engineers are also researchers: physicists of human made universes.

3.3.4 Professionalism of SE

Mechanical engineers are fully versant in the laws of the domain for which they create artifacts (Newton's Laws, etc.). Radio engineers, when hired, a fully versant in Maxwell's Equations — laws governing their application domain. And so it goes for all other professional engineers than SEs. Sometimes their basis in theoretical computer science is rather shaky. And always they know little or nothing about the business domain for which they develop software: financial services, transportation, healthcare. It is not becoming of a professional. Domain engineering brings professionalism into SE.

3.4 "Deriving" Requirements

3.4.1 "The Machine"

By "the machine" we understand that computing system, hardware and software, which is to be inserted in the domain in order to support some activities of the domain.

3.4.2 Three Kinds of Requirements

There are basically three kinds of requirements: (i) domain requirements — those which can be expressed sôlely using terms of the domain; (ii) machine requirements — those which can be expressed without using terms of the domain (in the vernacular: sôlely using terms of the machine); and (iii) interface requirements — those which must be expressed using terms both of the domain and the machine. We treat these in a slightly changed order.

Domain Requirements

One can rather simply, that is very easily, develop the domain requirements from the domain description. Here is how it is done: "Go through" the domain description, with the various requirements stakeholders, while seeking answers to the following sequentially order questions: (a) Projection: should this "line" (being read) be part of the requirements? (b) Instantiation: if so, should what is described be instantiated from its usually generic form? (c) Determination: and — if it is expressed in a loose or non-deterministic, i.e., under-specified manner — should it be be made more determinate? (d) Extension: Are there potential phenomena or concepts of the domain which were not described because they were infeasible in the domain — if so can a machine make it feasible? (e) Fitting: Are there other requirements development, elsewhere, with which the present one could be "interfaced"?

The result of a domain requirements development phase is a (sizable) document that is expected to (functional-) requirements-specify that which can be computed (and communicated electronically).

Interface Requirements

The interface requirements development stage now starts by identifying all the phenomena that are to be shared between the domain ("out there") and "the machine" ("in here")!

The shared phenomena and (now, to be, machine) concepts are either simple entities, functions, events or behaviours. Each such shared phenomenon leads, respectively, to interface requirements concerning entities: bulk data (database) initialisation and refreshment; functions: man/machine dialogue concerning computational progress; events: handling of interrupts, unforeseen or rare situations, and the like; and behaviours: logging and replay monitoring and control. For each of the four classes due consideration is paid wrt. use of visual displays, tactile instruments ("mouse", keyboard, stylos, sensitive screens or pads, etc.), audio equipment: sound recognition and production, smell, taste, and physics measurements.

The result of an interface requirements development phase is a (sizable) document, adjoint to or interleaved with[2] the domain requirements document, that is expected to (user-) requirements-specify that which can be interchanged (input/output between man or machine and machine).

Machine Requirements

Machine (or systems) requirements deal with such matters as performance: dealing with concerns of storage and response times (hence equipment "numbers"); dependability: accessibility, availability, reliability, fault tolerance; security, etc.; maintainability: adaptive, perfective, corrective and preventive maintenance; portability: development, demonstration, execution, and maintenance platform issues; documentation: installation, training, user, maintenance and development documents.

The machine requirements are developed "against" a check-list of all these requirements possibilities and focusing on each line of the domain requirements and interface requirements documents.

The result of a machine requirements development phase is yet a (sizable) document — adjoint to the domain and interface requirements documents — that is expected to (system-) requirements-specify that which can be also implemented.

3.4.3 Further SE Professionalism

Requirements engineering has been made easy. The domain is relatively stable. There is now a clear, well-defined path from domain models to requirements models. The adage, i.e., the common observation, *"requirements always*

[2]Whether adjoint or interleaved is a determined by style and by the problem-at-hand.

change" need no longer be true. For a software engineer to command the process of creating domain models and comfortably transforming them into requirements with input from requirements stakeholders signifies professional SE.

3.5 Software Design

To round off the triptych approach to software development, such as advocated here, we briefly mention that the requirements specifications (which prescribe what), are now the basis for refinement into software designs (the how). We shall not go into these aspects in this paper but refer the reader to the rather fully comprehensive [31–33] other than recalling

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

Correctness of $\mathcal{S}$ with respect to $\mathcal{R}$ can be proven using recorded assumptions about the $\mathcal{D}$.

3.6 Rôle of Domain Descriptions

3.6.1 A Science Motivation

One rôle of domain modelling is that of obtaining and recording understanding. The domain engineer is a researcher: studies "new territory". Just as physicists for centuries have studied "mother" nature, so it is high time we study the universes of man-made structures.

3.6.2 A Engineering Motivation

Another rôle of domain modelling is the engineering one. We present an elegantly formulated summary of the rôle of domain descriptions in software engineering. It was expressed by Sir Tony Hoare — in an exchange of e-mails in July 2006.

Tony Hoare's Assessment

"There are many unique contributions that can be made by domain modelling.

1. The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.
2. They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.

3. They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.
4. They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.
5. They enumerate and analyse the decisions that must be taken earlier or later any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

Structuring of Rôles

We rephrase our and Tony's formulation as follows: Domain models represent theories of human organisations — and as such they are interesting in-and-by themselves. Domain models also represent a first major result in a software development: In proving correctness of $\mathcal{S}$oftware with respect to $\mathcal{R}$equirements assumptions are repeatedly made about the $\mathcal{D}$omain. We can summarise the engineering rôle of domain engineering as follows:

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

3.6.3 Conventional SE Paradigms

We are presenting a novel theory-based approach to software development. This, the triptych approach has to compete with conventional software development methods, that are currently *en vogue* in the industry. Let us try relate some of these conventional methods to what we advocate: Examples are: object-oriented programming (OO), unified modeling language (UML), component-based programming (CBSE), aspect-oriented software engineering (AOS), and int∃ntion∀l software development (∃∀).

I believe that you will find that some of the strengths of OO, CBSE and especially ∃∀ are occurring "naturally" in both the domain engineering and in the "derived" requirements engineering. That is, we wish to point out how we can understand basic traits of OO, CBSE, AOP and especially ∃∀ and thus explain why these approaches have won adherents.

OO Programming

We assume that the audience is well familiar with the OO programming paradigm. We shall briefly list some of the OO "quarks"[3]: class, object, method, message passing, inheritance, encapsulation, abstraction and polymorphism. Several of these "quarks" are specifically oriented at programming rather than specifying.

[3]Term used by Ms Deborah J. Armstrong in naming fundamental OO concepts (http://en.wikipedia.org/wiki/Object-oriented_programming)

The foremost feature of OO that are of interest in the context of domain and requirements engineering is the concept of objects. OO objects are also present in the triptych approach to formalisation: the encapsulation of related entities, functions and events in a module, with that module now denoting possible behaviours. Many object modelling techniques, essentially all discovered by the Simula 67 originators [12], carry over to domain and requirements specifications.

So first we remark that OO primarily addresses programming assuming given requirements, whereas the triptych approach advocated here address the entire span of software development from, and significantly focused on domain modelling, and on deriving requirements from domain descriptions. The triptych approach advocated here does not prescribe which combination of coding paradigms you may wish to use: OO, AOP, CBSE, XP (Expert Programming), etc.

The RSL scheme, class and object constructs and their parameterisation provide for straightforward expression of object-orientedness.

UML

UML, to us, is a confused approach to software development. Yet, however, it has some appealing features. It mixes informal textual specifications with several graphical techniques (Petri nets, MSCs, Statecharts). It supposedly has a "powerful" data schema concept (class diagrams). In UML you are programming more than you are specifying.

But UML, to us, has problems: It has no notion of abstraction. It thus has no notion of stepwise development — necessary to conquer complexity with a phase, and necessary to separate the necessarily separated phases of domain modelling (the why), requirements modelling (the what), and software design (the how). One cannot reason logically over UML specifications. It is placed somewhere between requirements and software design specifications. For these reasons we cannot take UML serious[4] and we wonder why professional software engineers do?

CBSE: Component-based SE

A basic concern of CBSE is building software systems from reusable components. There seems to be many different "schools" of CBSE. Being in Japan it is reasonable to follow that of the CBSE Group, Fukazawa Laboratory, Waseda University[5]: In a narrow sense, *"a software component is defined as*

[4]Well, we do, of course, and that is why, in [32, Chaps. 12-14], we "UML-ise" formal techniques in that they cover Petri Nets, Message and Lice Sequence Charts and Statecharts, respectively, and "integrate" them with, in our case, RAISE/RSL.

There really is no way in which we can 'formalise' UML.

[5]http://www.fuka.info.waseda.ac.jp/Project/CBSE

a unit of composition, and can be independently exchanged in the form of an object code without source codes. The internal structure of the component is not available to the public. The characteristics of the component-based development are the following:

- *Black-box reuse*
- *Reactive-control and component's granularity*
- *Using RAD (rapid application development) tools*
- *Contractually specified interfaces*
- *Introspection mechanism provided by the component systems*
- *Software component market (CALS)*

It is natural to model and implement components in an object-oriented paradigm/language. Therefore, when understanding the component, the traditional techniques in the OO paradigm such like OO framework, design patterns, architecture patterns and meta-patterns are very important"

We shall now try evaluate basic tenets of CBSE in light of the triptych approach. We strongly advice that search for components start at the level of domain modelling.

- Start with a simple entity.
- Which are the functions that "create", inspect and change variables primarily involving that (or those) simple entity (entities) ?
- Which are the events ?
- Choose such entities that basically form states that are subject to, i.e.., "exhibit" behaviours.

Once identified in the domain, requirements may project, instantiate, determinate, extend and fit them in a variety of ways. Such domain-to-requirements operations may be expressed in the form of adjusting suitable parameters to a schema/module/object like abstract formalisation of the domain component. From here on many of the intriguing issues of CBSE can be better understood, either as basic abstraction-to-concretisation refinements or as simple coding tricks. CBSE certainly has a rôle in making the triptych approach even more viable.

AOP: Aspect-oriented Programming

A basic concern of AOP is that some code is scattered or tangled, making it harder to understand and maintain. It is scattered when one concern (like logging) is spread over a number of modules (e.g., classes and methods). That means to change logging can require modifying all affected modules. Modules end up tangled with multiple concerns (e.g., account processing, logging, and security). That means that to change one module entails understanding all the tangled concerns.

AOP attempts to aid programmers in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization. AOP does so using primarily language changes, while AOSD (aspect-oriented software development) uses a combination of language, environment, and methodology.

Separation of concerns entails breaking down a program into distinct parts that overlap in functionality as little as possible. All programming methodologies — including procedural programming and object-oriented programming — support some separation and encapsulation of concerns (or any area of interest or focus) into single entities. For example, procedures, packages, classes, and methods all help programmers encapsulate concerns into single entities. But some concerns defy these forms of encapsulation. Software engineers call these cross cutting concerns, because they cut across many modules in a program.

So, really, AOS, is primarily a coding discipline. So why do we bring it up here, in a presentation which is primarily not about coding, but about domains and requirements. Some software engineers (may) ask: *What is relation between the triptych approach and AOS?*Our answer is: The cross cutting concerns appear not to be caused by domain requirements, nor by interface requirements, but by machine requirements. Thus problems of cross-cutting concern appears to be introduced in a serious, but not really user-oriented stage of requirements development. This "discovery" might enlighten researchers in the AOS community.

∃∀: Intentional Software Development

The intentional software development paradigm is the creation of Charles Simonyi[6].

It appears that little if any literature is readily accessible [2, 224–226]. So we shall resort to quoting from *Intentional Software*'s Web page (http://-intentsoft.com/technology/glossary.html). The quotes are in *slanted text.*

Domain

A domain is an area of business, engineering or society for which a body of knowledge exists. Examples include health care administration, telecommunications, banking, accounting, avionics, computer games and software engineering.

Domain Code

Domain code is the structured code to represent the intentions contributed by subject matter experts for the problem being solved. Domain code includes contributions from all domains relevant to the software problem. Domain code is not executable (as traditional source code is — by compilation or interpretation), but it can be transformed into an implementation solution when it is input to a generator that has been programmed to perform that transformation process.

[6]Intentional Software, Bellevue, Washington, USA; http://intentsoft.com

Domain-Oriented Development

Domain-oriented development is the process of separating the contributions of subject matter experts and programmers to the maximum extent so that generative programming can be applied to structured domain code. This greatly simplifies improvements to the domain and implementation solutions.

Domain Schema

A domain schema is a schema for a specific domain. The domain schema defines the domain terminology and any other information that is needed — for the intentional editor and generator to work — such as parameters, help text, default values, applicable notations and other structure of the domain code. Domain schemas are created by the subject matter experts and programmers working together, and are expressed in a schema language.

Domain Terminology

Domain terminology means the terms of art (words with a special meaning) in a domain, for example "claim payment" in health care administration. Domain terminology is important because it is the usual way to express intentions. Broadly speaking, terminology includes notations normally used by a subject matter expert, such as tables, flowcharts and other symbols. The meaning of the terms is part of the domain knowledge that is shared between subject matter experts and programmers to the extent necessary and ultimately designed into domain schemas and the generator.

Discussion

Intentional software development, it should be clear from the above, builds on a number of software development tools which are provided with domain description-like information and which can then automate code generation. Other than that we shall neither comment nor speculate on Charles Simonyi's characterisations.[7] We believe that the reader can easily see the close relations to the triptych phases of development. We find them fascinating and will try communicate our own observations to Charles Simonyi before commenting in depth. ∃∀ certainly has a rôle in making the triptych approach even more viable.

[7]Well, I cannot, of course, refrain from saying that seven groups of my students have founded a number of Danish software companies whose corporate asset it is that they generate code for application of the domain-specific area which is their company's hallmark, their corporate asset.

3.7 Conclusion

3.7.1 What Have We Achieved?

We have outlined two of the major phases of a new approach to software development: domain engineering — primarily — and requirements engineering — as it relates to domain engineering. We have not really covered the relation of requirements engineering to software design, i.e., programming — other than now saying: software design is then a further refinement of the requirements, and, well, read next! We have then related this, the triptych approach to some current programming and software development paradigms: OO, CBSE, AOS — as mostly programming cum coding paradigms, and Intentional Software Development which, to us, have a much clearer and cleaner understanding of the domain, with the domain intentions, when being edited, probably having the editing stage amount to, or being based on some form of requirements development.

3.7.2 What More Need be Achieved?

Well, on the basis of three volumes,

- D. Bjørner: *Software Engineering, Vol. 1: Abstraction and Modelling* (Springer, 2006)
- D. Bjørner: *Software Engineering, Vol. 2: Specification of Systems and Languages* (Springer, 2006)
- D. Bjørner: *Software Engineering, Vol. 3: Domains, Requirements and Software Design* (Springer, 2006)

supported by almost 6,000 lecture slides and supported by extensive RAISE tools, what more could one wish?[8]

The answer is: more tools, tools to support documentation: creation, editing, versioning, etc.; tools to support domain and requirements acquisition and analysis; tools to extend the use of RAISE; as well as tools to integrate the formal use of RAISE with the formal use of Petri nets [148,199,210–212], MSCs [142–144], LSCs [80,128,153], Statecharts [123,124,126,127,129], Duration Calculus and TLA+ [247,248], [155,156,175,176], etc. [7,65,69,111,216], further theorem proving, proof checking, model checking and testing tools.

[8]Well, this monograph should be added to the above three items as well as a forthcoming textbook: Domain Engineering – The Triptych Approach, http://www.imm.dtu.dk/~db/de-p.pdf (or /tseb.pdf)

4

Verified Software for Ubiquitous Computing

A VSTTE/Ubiquitous Computing Project Proposal

Abstract

We sketch a project that is to be pursued over the next 15 years and, hopefully, by partners on five continents: (i) the Pacific: Australia, New Zealand; (ii) Asia: China, India, Japan, Korea, Macau, Singapore; (iii) Europe (Austria, Denmark, England, France, Germany, Ireland, Italy, Portugal, Sweden, Switzerland) and (iv-v) North and South America (Argentina, Brasil, Canada, USA).

Sets of commensurate descriptions of transportation domains shall be developed and analysed. A domain theory of, it is proposed, transportation shall thus emerge. Based on these descriptions (formalised using a set of integrated formal notations) a set of commensurate prescriptions of requirements for a ubiquitous computing system for automated highways shall be developed. The "commensurateness" of the different requirements prescriptions and their relations to the commensurate domain descriptions shall likewise be proved. Eventually these requirements shall be programmed with program annotations relating program statements to respective requirements and domain models.

In this report we outline and discuss implications of the above.

There are two versions of this document.

A short version is the invited paper for the First Asian Working Conference on Verified Software, UNU-IIST, Macau, 29–31 October, 2006[1].

A long version aims at informing potential project partners in, we hope, sufficient detail so as to help them decide to join the project.

This is the long version.

4.1 The Backgrounds

4.1.1 "A Gleam in the Eye"

Three strands appear to come together in this project proposal.

[0]This is an edited version of [38]. Presented at the 29 October 2006 1AWCVS (First Asian Working Conference on Verified Systems), Macau SAR.

[1]http://www.iist.unu.edu/www/workshop/AWCVS2006/

One initiation point for this research and advanced development proposal is the author's fascination with domain engineering as applied to transportation. Initially to the railway sub-domain. More generally, and later, with the transportation domain. And, in this proposal, the instantiation of this transportation domain to the concept of "the automated highway".

Two other initiation points were Tony Hoare's similar obsession with VSTTE, "a million lines of verified code", and Robin Milner's "lifting" of the concerns of ubiquitous computing to elegant heights.

More than 30 years of quest for provably correct software, since the days of the birth of VDM at the IBM Laboratory, Vienna, Austria, is embodied in the project.

4.1.2 Grand Challenges of Informatics

Ref. 1 enumerates 13 criteria for a project to attain "grand challenge" status;

1. *Criteria for a Grand Challenge*
   Tony Hoare, Robin Milner, Martyn Thomas, and Alan Bundy;
   Revised by Tony Hoare, 30 May 2002.
   http://www.cra.org/Activities/grand.challenges/hoare.pdf
   The primary purpose of the formulation and promulgation of a grand challenge is to accelerate the advancement of science. The only purpose of this document is to clarify the concept of grandness as applied to a scientific challenge. The suggested criteria concentrate on those aspects of grandness that contribute towards the primary scientific goal of the challenge. They are the criteria that distinguish a grand challenge from the many other worthy kinds of challenge that are formulated to contribute to economic, political or other societal goals. Each criterion is intended to describe some property relevant for the comparison and evaluation of proposed grand challenges solely according to their degree of grandeur. No challenge, however grand or otherwise desirable, should be expected to meet all the criteria. The order of the criteria is not significant.
   (a) *It arises from scientific curiosity about the foundation, the nature or the limits of the scientific discipline.*
   (b) *It has enthusiastic support from (almost) the entire research community, even those who do not participate.*
   (c) *It has international scope: participation would increase the research profile of a nation.*
   (d) *It is generally comprehensible, and captures the imagination of the general public, as well as the esteem of scientists in other disciplines.*
   (e) *It was formulated long ago, and still stands.*
   (f) *It goes beyond what is initially possible, and requires development of techniques and tools unknown at the start of the project.*
   (g) *It calls for planned co-operation among identified research teams and schools.*

(h) *It encourages and benefits from competition among identified individuals and teams, with clear criteria on who is winning, or has won.*
(i) *It necessitates collaboration of several identified research specialties, theoretical and/or practical.*
(j) *It decomposes into identified intermediate research goals, whose achievement brings scientific or economic benefit, even if the project as a whole fails.*
(k) *It should be rather obvious how far and when the challenge has been met (or not).*
(l) *It should lead to radical paradigm shift, breaking free from the dead hand of legacy.*
(m) *It is not likely to be met simply from commercially motivated evolutionary advance.*

The word "challenge" is these days commonly applied to a survey of relevant topics in some general research area that promises some short-term social or economic benefit, often just to a single nation or region (provided, of course,that sufficient research funds are allocated to it immediately). The criteria listed above are not intended to apply to such challenges.

The tradition of Grand Challenges is common in many branches of Science. If you want to know whether a challenge is grand enough, compare it with:

- *Put a man on the moon*
- *Cure cancer (in ten years)*
- *Prove Fermat's last theorem*
- *Map the Human Genome*
- *Map the Human Proteome*
- *Find the Higgs boson*
- *Find Gravity waves*
- *Unify the four forces of Physics*
- *Complete Hilbert's programme for the foundations of mathematics*

Some of these have succeeded, some of them have failed, and some of them are still open. In computing, the following are listed, not because we recommend them, but because they are familiar to Computer Scientists. The reader is invited to evaluate any of the examples according to the criteria given later. Obviously, no challenge will meet all of them.

- *Prove that P is not equal to NP*
- *The Turing test*
- *The Verifying Compiler*
- *A championship chess program*
- *A championship GO program.*
- *Automatic translation of scientific literature from Russian to English.*
- *A mathematical model of the evolution of the web.*
- *A wearable computer serving as a guide dog for the blind.*

Some of these have succeeded, some of them have failed; some of them have lost the interest of the scientific community, and some are reliably *conjectured to be impossible. They are quoted here only as illustrations of the property of grandeur. There is no implication that any of these will be found worthy of general support.*

4.1.3 VSTTE: Verified Software: Theories, Tools and Experiments

The idea at the root of the VSTTE grand challenge is that of a verifying compiler. That compiler accepts annotated code. The annotations are, for example, like pre- and post-conditions (say in a Hoare-style logic). The annotations are meant to express requirements and assumptions about the machine and the environment of the machine on which the code is executed. The verifying compiler is then expected to verify that the annotated code indeed satisfies the annotations.

Reference 2 is an original source for the VSTTE effort; Ref. 3 is a position statement prepared for the conference otherwise announced in Ref. 4 while Ref. 5 elaborates on Ref. 3; Ref. 6 provides further elaboration in the form of a FAQ; Ref. 7 announces one kind of follow-up on the VSTTE conference in October 2005.

2. The Verifying Compiler: A Grand Challenge for Computing Research
Tony Hoare, Journal of the ACM, Vol. 50, No. 1, January 2003
http://www.csl.sri.com/ shankar/GC04/hoare-compiler.pdf
3. VSTTE: Verified Software: Theories, Tools, Experiments
Conference home page, Oct. 10th-14th, 2005
http://vstte.inf.ethz.ch/ or http://vstte.ethz.ch/
4. The IFIP Working Conference on Verified Software: Theories, Tools, Experiments; conference report
Tony Hoare, Jayadev Misra, and N. Shankar, October 20, 2005
http://vstte.ethz.ch/report.html
5. Verified Software: Theories, Tools, Experiments — Vision of a Grand Challenge project
Tony Hoare and Jayadev Misra, July 2005.
http://vstte.ethz.ch/pdfs/vstte-hoare-misra.pdf
6. Verified Software: Frequently Asked Questions
Tony Hoare and Jayadev Misra, 19 September 2005
http://wiki.se.inf.ethz.ch/vstte/index.php/Verified_Software:_Frequently-_Asked_Questions
7. FLoC'2006 Workshop on Verified Software: Theories, Tools, and Experiments
K. Rustan M. Leino, Summer 2006
http://research.microsoft.com/~leino/vstte2006/
8. Verified Software Roadmap 2006
A list of 16 references (in the form of URLs) are given to position statements on the topic of VSTTE. http://qpq.csl.sri.com/vsr/private/verified-software-roadmap-2006/, http://qpq.csl.sri.com/vsr/private/verified-

-software-roadmap-2006/dines.pdf/view refers to an early version of the current proposal.

4.1.4 Ubiquitous Computing: The Automated Highway

By ‘ubiquitous computing’ we shall here understand the presence of computation (and communication) “everywhere”. And by ‘the automated highway’ we shall specifically understand the use of computation and communication in every possible aspect of automating the orderly (safe and efficient) movement of cars along a net of highways. This latter implies the availability of sensors and actuators along and possibly above and/or “outside” the highway net and in all vehicles moving along this net of highways.

Ubiquitous Computing

9. *Ubiquitous Computing Grand Challenge: Introduction*
   http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/
   - *There is burgeoning population of ’effectively invisible’ computers around us, embedded in the fabric of our homes, shops, vehicles, farms and some even in our bodies. They are invisible in that they are part of the environment and we can interact with them as we go about our normal activities. However they can range in size from large Plasma displays on the walls of buildings to microchips implanted in the human body. They help us command, control, communicate, do business, travel and entertain ourselves, and these ’invisible’ computers are far more numerous than their desktop cousins. How many computers will you be using, wearing, or have installed in your body, in 2020? How many other computers will they be talking to? What will they be saying about you, doing for you, or to you? By that time computers will be ubiquitous and globally connected. Shall we be able to manage such large-scale systems, or even understand them? How do people interact with them and how does this new pervasive technology affect society? How can non-computing people configure and control them? What tools are needed for design and analysis of these constantly adapting and evolving systems? What theories will help us to understand their behaviour?*
   - *These are the sort of issues which make Ubiquitous Computing a Grand Challenge; join us in addressing them.*
   - *The Ubiquitous Computing Grand Challenge (UbicompGC) is one of the 6 UKCRC Grand Challenges. It was formed by merging two of the original Grand Challenges GC2 ”Science for global ubiquitous computing” which focused on theory and GC4 ”Scalable ubiquitous computing systems” which focused on engineering aspects. UbicompGC is formulating a research manifesto which postulates the need for combined*

Science (theory) as well as addressing the Engineering and Social issues related to building Ubiquitous Systems. So far, most research in the UK and elsewhere has focussed on the Engineering with very little attention on the theory required to underpin the design and analysis of ubiquitous systems which are intrinsically large-scale and complex. Some of the work in the Equator project has addressed social aspects and how people will interact with Ubiquitous Systems. The overview page summarises the goals of the challenge.
Background information on the Grand Challenges together with a report describing each of the challenges can be found on the UKCRC Grand Challenges site. We intend to hold a mini-workshop as part of the Grand Challenges Conference in Glasgow 22-24 March 2006.

- *UbicompGC activities are currently being promoted by the EPSRC funded UK-UbiNet Network grant which predated UbiCompGC and runs from April 2003 - March 2006. The Ubicomp steering committee are all members of the UK-UbiNet management committees. See the UK-UbiNet for information on UK and worldwide activities on ubiquitous computing, workshops organised which addressed the issues raised by the UbiCompGC research manifesto, future workshops and information on events and conferences.*
- *We welcome discussion of all aspects of UbicompGC. For this purpose, please subscribe to the UbicompGC email list. We especially invite discussion on the final draft of the UbicompGC manifesto, which we aim to finalise by 1 September 2005.*
- *There will be a sequence of evolving Annexes to the manifesto, including descriptions of ”foothill projects”, also published for discussion with the manifesto. Two threads of discussion, called 'Manifesto' and 'Foothills', have been created for these discussions.*
- *More generally, wide-ranging discussion will greatly help the steering committee in building a community around the Grand Challenge, and in coordinating its activity.*
- *UbicompGC Steering Committee*
  (a) *Prof. Morris Sloman, Department of Computing, Imperial College London (Chairman)*
  (b) *Dr. Dan Chalmers, Informatics, University of Sussex*
  (c) *Prof. Jon Crowcroft, Computer Laboratory University of Cambridge*
  (d) *Prof. Marta Kwiatkowska, School of Computer Science, University of Birmingham*
  (e) *Prof. Robin Milner, Computer Laboratory University of Cambridge*
  (f) *Prof. Tom Rodden, Computer Science and IT, University of Nottingham*
  (g) *Prof. Vladimiro Sassone, Informatics, University of Sussex*

10. *Ubiquitous Computing Grand Challenge: Overview*
http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/overview.html

- *By 2020 how many computers will you be using, wearing, have in your home, or even in your body? Computers are ubiquitous and will soon be globally connected. Shall we be in control of the complex emerging behaviour arising from their aggregation in a "ubiquitous" global network, or even understand it? As these devices become smaller, more numerous, more independent from users and more deeply embedded in the world around us, they raise formidable scientific and engineering challenges.*
- *We propose to develop scientific theory and the design principles of Global Ubiquitous Computing together, in a tight experimental loop.*
- *At least since Leonardo, engineers have relied upon science when building their artifacts, whilst real-world challenges have often set the goal-post for scientists. Nevertheless, often in the past the two have developed independently from each other. Why are we advocating here for their integration as the key of our research method? The reason is in the very scale and nature of ubiquitous computers, and the way they are growing intimately embedded in our lives: we can hardly afford "experimental" failures, as we allow them to control transport, monitor our health, regulate our bank accounts as well as our most delicate global financial mechanisms.*
- *The final goals of the research proposed here can be described succinctly as:*
  - ⋆ *To define a set of design principles that pertain to all aspects of ubiquitous computing and are accepted, taught, and used in practice;*
  - ⋆ *To develop a coherent information science that allows descriptive and predictive analysis of ubiquitous computing at many levels of abstractions, and to use it to derive, analyse, justify its constructions.*
- *Qualities of Global Computers*
  In order to allow a ubiquitous computer to blend in the environment and act "invisibly" whilst being dependable and remain controllable, we will have to guarantee ("by design and construction") that it is:
  - ⋆ *Fluid: it structure will vary frequently, and evolve in the long term;*
  - ⋆ *Purposive: each of its components has a purpose, which explains its actions;*
  - ⋆ *Autonomous: it makes autonomous decisions based on its context and its "experience" of it;*
  - ⋆ *Reflective: it can take actions (on itself) based on self-analysis;*
  - ⋆ *Trustworthy: it can be trusted not to misuse information and resources;*
  - ⋆ *Tactful: its impact on its surroundings is effective, but minimal*
  - ⋆ *Scalable: the size of its subsystems can vary greatly, yet the same principles apply to them all; and*
  - ⋆ *Efficient.*

- *Challenges of Global Computing*
  We are already developing new theories, programming languages and experimental systems to help engineer ubiquitous computers. The challenges ahead (arising from the qualities required) are many, and of considerable depth. Amongst the engineering ones:
  - ⋆ *design devices to work from solar power, are aware of their location and what other devices are nearby, and form cheap, efficient, secure, complex, changing groupings and interconnections with other devices;*
  - ⋆ *engineer systems that are self-configuring and manage their own exceptions;*
  - ⋆ *devise methods to filter and aggregate information so as to cope with large volumes of data, and to certify its provenience.*
  - ⋆ *business model for ubiquitous computing, and other human-level interactions.*
- *Amongst the science challenges we list:*
  - ⋆ *discover mathematical models for space and mobility, and develop their theories; devise mathematical tools for the analysis of dynamic networks;*
  - ⋆ *develop model checking, as well as techniques to analyse stochastic aspects of systems, as these are pervasive in ubiquitous computing;*
  - ⋆ *devise models of trust and its dynamics;*
  - ⋆ *design programming languages for ubiquitous computing.*
- *These and other ideas are developed in detail in the Manifesto of the UKCRC Grand Challenges proposal:*
  - ⋆ *Global Ubiquitous Computing: Design and Science*
  - ⋆ *http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/manifesto.html*

11. *Ubiquitous Computing, The Computer Journal.2006; 49: 390-399*
    http://www.bcs.org/BCS/Awards/Events/cjlectures/ubiquitous.htm
    http://comjnl.oxfordjournals.org/cgi/content/full/49/4/390
    - *The Computer Journal presents the first Computer Journal Lecture: "Ubiquitous Computing" by Robin Milner.*
    - *23rd February 2006, Central London, 5:30 pm for a 6:15 pm start.*
    - *Lecture by Robin Milner - Computer Laboratory, University of Cambridge. The lecture will be followed by a debate on the lecture. The lecture itself, and the discussion following it, will be edited for the Computer Journal.*

    The vision of ubiquitous computing (ubicomp) is that computing entities become an effective part of our environment, supporting our lives without our continual direction, so that we can be largely unaware of them. One of the UK Grand Challenges for Computing Research addresses not only the ubicomp vision, but also the design principles and theories that will support it. Ubicomp will entail hardware/software systems that exceed those that we know by orders of magnitude in size. There is little *chance of extrapolating existing methods of software production to cope with them. Ubicomp offers an opportunity to develop a deeper science of computing that interweaves three ingredients - vision, design and theory - more intimately than ever before. This is the Grand Challenge; the lecture will explore how we may approach it.*

12. *Ubiquitous Computing: Shall we understand it?*
    Follow-up discussion on the above: http://www.bcs.org/server.php?show-=ConWebDoc.4708&setPaginate=No
13. *Wikipedia's URL on Ubiquitous Computing: http://en.wikipedia.org/wi-ki/Ubiquitous_computing*
    Ubiquitous computing (ubicomp, or sometimes ubiqcomp) integrates computation into the environment, rather than having computers which are distinct objects. Other terms for ubiquitous computing include pervasive computing, calm technology, and things that think. Promoters of this idea hope that embedding computation into the environment and everyday objects would enable people to move around and interact with information and computing more naturally and casually than they currently do. One of the goals of ubiquitous computing is to enable devices to sense changes in their environment and to automatically adapt and act based on these changes based on user needs and preferences.
14. *Ubiquitous Computing Evaluating Consortium:*
    http://ubiqcomputing.org/ — An effort led by SRI Intl.
15. *Ubicomp: Conference announcements:*
    http://ubicomp.org/ubicomp2005, http://ubicomp.org/ubicomp2006
16. *IEEE Journal on Pervasive Computing*
    http://www.computer.org/portal/site/pervasive/

The Automated Highway

Just very briefly: an automated highway system or Smart Road, is an advanced intelligent transportation system technology designed to provide for driverless cars on specific rights-of-way. It is most often touted as a means of traffic congestion relief, since it drastically reduces following distances and thus allow more cars to occupy a given stretch of road.

17. *Wikipedia: Automated Highway System http://en.wikipedia.org/wiki/-Automated_highway_system*
    An automated highway system (AHS) or Smart Road, is an advanced Intelligent transportation system technology designed to provide for driverless cars on specific rights-of-way. It is most often touted as a means of traffic congestion relief, since it drastically reduces following distances and thus allow more cars to occupy a given stretch of road.
    - *How it works*
      The roadway has magnetized stainless-steel spikes driven one meter apart in its center. The car senses the spikes to measure its speed

and locate the center of the lane. Further the spikes can have either magnetic north or magnetic south facing up. The roadway thus has small amounts of digital data describing interchanges, recommended speeds, etc.
The cars have power steering, and automatic speed controls, but these are controlled by the computer.
The cars organize themselves into platoons of eight to twenty-five cars. The platoons drive themselves a meter apart, so that air resistance is minimized. The distance between platoons is the conventional braking distance. If anything goes wrong, the maximum number of harmed cars should be one platoon.

- *Deployments*
  A prototype automated highway system was tested in San Diego County, California in 1991 along Interstate 15. However, despite the technical success of the program, investment has moved more toward autonomous intelligent vehicles rather than building specialized infrastructure. The AHS system places sensory technology in cars that can read passive road markings, and use radar and inter-car communications to make the cars organize themselves without the intervention of drivers.

18. *Foothill Project: Automating the Highway: http://www-dse.doc.ic.ac.uk/-Projects/UbiNet/GC/Manifesto/fp-automatinghighway.html*
    UKCRC Grand Challenges for Computing Research. Ubiquitous Computing: Science and Design
    Since this project is one of the bases for the present proposal we quote in extenso.
    - *Jon Crowcroft: http://www.cl.cam.ac.uk/users/jac22/*
    - *Monitoring and control of private vehicles on the public highway is high on the political agenda; this is because it is becoming feasible, and may be desirable for at least two reasons: first, from the economic perspective, it may achieve more efficient use of road resources; second, from the safety perspective, it may achieve a significant drop in injury and death on the roads. Various prototypes exist, and various projects are current. Many technologies interact, and there are numerous legal, political and economic stakeholders. We propose a foothill project to study monitoring and control with particular concern for efficiency and safety, in the context of ubiquitous systems for transport. For efficiency (of road use) the monitoring and control may be either distributed or centralised, or a combination of the two. In a distributed system the car receives information from navigation systems and roadside monitors concerning routes, conditions and prices; it (or its driver) then makes a decision and pays. on the other hand a centralised system, such as the London congestion-charging scheme, depends entirely on a network of roadside monitors, recording data about vehicles, drivers and journeys on a central database used as the basis for billing.*

- *To improve safety, there a spectrum of possible solutions from distributed to centralised systems. At the centralised extreme, 'car-trains' have been proposed; vehicles joining trunk routes would be logically clumped, and controlled by a single aggregate unit. At the distributed extreme, each vehicle always chooses its own velocity, using data from on-board and remote sensors. There are many research problems; for example:*
  - ⋆ *What are the design spaces for distributed and/or centralised systems in the two cases? Can they me mixed, e.g. distributed for efficiency of road-use but centralised for safety?*
  - ⋆ *By what measures can each solution in the space be assessed for its contribution to both efficiency and safety?*
  - ⋆ *In each possible design, what threats arise from neglect or malevolence? These threats may attack endanger correct technical function, or they may endanger privacy (for example, centralised records may be illegally mined to deduce driver habits).*
  - ⋆ *Success in addressing these problems will involve a variety of theoretical or simulational models of distributed and mobile processes; and will prompt the further development of such models.*

19. *A longer paper addressing these issues of item 18 on the facing page is also available:*
http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/Manifesto/road.pdf
20. *Carnegie Mellon AHS group http://www.cs.cmu.edu/Groups/ahs/*
21. *Automated Highway Systems: http://scholar.lib.vt.edu/theses/available/etd-5414132139711101/unrestricted/ch2.pdf, http://72.14.203.104/search?q=cache:-ctx_HZkkonwJ:scholar.lib.vt.edu/theses/available/etd - 5414132139711101/unrestricted/ch2.pdf+Automated+Highway&hl=en&ct=clnk&cd=3*

4.1.5 Domain Engineering

Section 4.3.2 should probably be read first. By a domain model we understand an abstraction of some reality, something which exists, and as it is conceived to be. A domain model is typically expressed by both a precise, and as here English narrative and a formalisation. If the formalisation is in logic then one of the set of models which satisfy the logic is what we mean by "the domain model". If the formalisation is in some model-oriented specification language, such as `B` [1, 71], `RAISE` [31–33, 44, 101, 104, 106], `VDM-SL` [55, 56, 95, 96] or `Z` [132, 133, 229, 230, 242], then the semantics of these languages explicitly designates the model(s). To create such a model, or, typically, set of models, one needs to acquire domain knowledge, analyse it, and then bring the acquisition and the analysis together in a description such as for example given in Sect. 4.3.2 and according, for example, to the principles, techniques and tools as outlined to some depth in Chap. 11 of [33].

22. Domain Engineering: Part IV, Chaps. 8–16 of [33].

These chapters cover:

8. Overview of Domain Engineering
9. Domain Stakeholders
10. Domain Attributes: Continuity, Discreteness and Chaos; Statics and Dynamics; Tangibility; One, Two, ... Dimensionality
11. Domain Facets: Domain Facilitators (Business Processes), Intrinsics, Support technologies, Management and Organisation, Rules and Regulations, Scripts, Human Behaviour
12. Domain Acquisition
13. Domain Analysis and Concept Formation: Incompleteness, Inconsistency, Conflicts; Concepts
14. Domain Verification and Validation
15. Towards Domain Theories
16. The Domain Engineering Process Model

23. “Domain Engineering”, Chapter in forthcoming EATCS Series textbook covering a number of BCS FACS Evening Seminars, 2003-2005, Eds. Jonathan Bowen and Paul Boca, Springer, 2007. http://www.jaist.ac.jp/~bjorner/domain.ps
24. Chapter 1 [28] “On Domains and Domain Engineering, Prerequisites for Trustworthy Software, A Necessity for Believable Project Management”, http://www.jaist.ac.jp/~bjorner/jaist-dom.ps, 182 pages

    This is a somewhat lengthy document which summaries item 22 on the previous page, lists and answers a number of FAQ, and brings, in a number of appendices, reasonably sized examples of domain descriptions or references to such: Transportation, Manufacturing, Documents, “The Market”, “Cyberrail” (a Japanese conception of a future railway system). In addition the document also surveys the concept of Business Processes (referred to in Chap. 11, see item 22 on the preceding page, as well as a summary of the RSL language.
25. Chapter 2 [29] “Possible Collaborative ‘Domain’ Projects: Japanese Private and/or Public Institutions + JAIST/DEDR”, The JAIST School of Information Sciences’ Domain Engineering and Digital Rights Group, http://www.jaist.ac.jp/~bjorner/jaist-dom-projs.ps

    Based on the lengthy document referenced in item 24, this short document outlines, for Japanese public and private institutions (government and industry) how possible collaborative domain engineering projects might be organised.

4.2 The Triptych Dogma

4.2.1 The Dogma

Before software can be designed its requirements must be understood. Before requirements can be prescribed the application domain must be understood.

Hence the "idealistic" triptych dogma:

- $\mathcal{D}$**:** First one must establish a proper, comprehensive domain theory for the application area — here transportation in general.
- $\mathcal{R}$**:** From such a domain theory one can, as we show in Chaps. 18–25 of Vol 3 of [31–33], i.e., [33], "derive" domain (i.e, functional) requirements.
- $\mathcal{S}$**:** From these again, as we show in Chaps. 26–29 [33], we can decide major structures of the desired software.

4.2.2 Verification

In proving software, i.e., program code, $\mathcal{S}$, correct with respect to requirements, $\mathcal{R}$, such a proof is always relative to assumptions about the domain, $\mathcal{D}$:

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

To do proper proofs we postulate that proper theories need first be established of the domain and of the requirements.

4.2.3 Decomposition of Project Effort

The present proposal hinges to some extent on the triptych dogma. But the proposal allows for research groups to focus their contributions "down stream" from domain descriptions and requirements prescriptions, or only on one of these.

Some "Verified Software for Ubiquitous Computing" project groups will wish to focus on one or more of:

- Domain Theories (descriptions and analysis) using one or another approach, including combinations of approaches, other groups on
- Requirements Prescription development from Domain Descriptions and proofs of development "transformations", other groups on
- Requirements Theories (descriptions and analysis), or on
- Software Design development from Requirements Prescriptions and proofs of development "transformations", or on
- Program Code Annotation and Verification — from Domain Descriptions and Requirements Prescriptions

Project management is then needed to secure that it all "hangs together", forms a whole.

4.3 The Project Proposal

4.3.1 Summary

Section 4.2.3 on the preceding page suggests a number of project components:

1. Sets of commensurate correlated informal and formal descriptions of transportation domains shall be developed and analysed. We refer to Sect. 4.3.2.
2. A theory of transportation shall thus emerge. That theory shall explain and verify rove what precisely is meant by commensurateness.
3. Based on these descriptions (formalised using a set of integrated formal notations) a set of commensurate prescriptions of requirements for a ubiquitous computing system for automated highways shall be developed.
4. The "commensurateness" of the different requirements prescriptions and their relations to the commensurate domain descriptions shall likewise be verified.
5. Eventually these requirements shall be programmed with program annotations relating program statements to respective requirements and domain models.
6. The programs with annotations shall be processed by a verifying compiler and the program code shall be automatically verified wrt. the program annotations.

4.3.2 Domain Theories

To get a grasp on what a domain description might entail we present an example domain description. From that the reader is expected to draw a number of conclusions, including whether this project proposal is of interest or not. We refer to Item 1 Sect. 4.3.1.

A Domain Model

Let us take a reasonably comprehensive look at an albeit simple domain description, namely that of a simple transportation net and its traffic. We refer to

- The BCS FACS Journal, December 1005
  http://www.bcs-facs.org/newsletter/facts200512.pdf

for a more detailed exposition of the below description.

(1.) A transportation net (N) consists of segments (S) and junctions (J). (2., 3.) That is, from a net one can observe the set of all segments and the set of all junctions. (4.) There is at least one segment and two (distinct) junctions in any net. (5.) Segments and junctions have unique identifications (Si, Ji)[2]. (6.,7.) One can observe segment and junction identifiers from segments, respectively junctions. (8., 9.) To avoid "junk" being models of our specification we must insist that the number of segments [junctions] of a net equals the number of segment [junction] identifiers of that net. (10., 11.) Segments are delimited by a pair of junctions, and from a segment one can (thus) observe the pair of junction identifiers of the delimiting junctions. (12., 13.) Junctions connect to one or more distinct segments, and from a junction one can (thus) observe the set of one or more respective segment identifiers.

```
type
1.  N, S, J
value
2.  obs_Ss: N → S-set
3.  obs_Js: N → J-set
axiom
4.  ∀ n:N • card obs_Ss(n) ≥ 1 ∧ card obs_Js(n) ≥ 2
type
5.  Si, Ji
value
6.  obs_Si: S → Si
7.  obs_Ji: J → Ji
axiom
8.  ∀ n:N • card obs_Ss(n) ≡ card {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
9.  ∀ n:N • card obs_Js(n) ≡ card {obs_Ji(c)|j:J • j ∈ obs_Js(n)}
type
10. Jip = {|{ji,ji′}:Ji-set • ji≠ji′|}
value
11. obs_Jip: S → Jip
type
12. Si1 = {|sis:Si-set•card sis ≥1|}
value
13. obs_Sis: J → Si1
axiom
14. ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒
15.     let {ji,ji′} = obs_Jis(s) in
16.     ∃! j,j′:J • {j,j′}⊆obs_Js(n) ∧ j≠j′ ∧
17.       obs_Si(s) ∈ obs_Sis(c) ∩ obs_Sis(c′) end
18. ∀ n:N, j:J • j ∈ obs_Js(n) ⇒
19.     let sis = obs_Sis(c), ji = obs_Ji(j) in
20.     ∃! ss:S-set • ss⊆obs_Ss(n) ∧ card ss=card sis ∧
21.     sis = {|obs_Si(s)|s:S•s ∈ ss|} end
```

[2]Segment and junction identifiers can be thought of as summarising the distinct spatial location of that which they identify.

(14.–17.) For every segment of a net there exists exactly a set of distinct junctions of that net whose identifiers are those observable from that segment, and (18.–21.) – vice versa – for every junction of a net there exists exactly a set of distinct segments of that net whose identifiers are those observable from that junction.

The above description covers just the net of its junction-connected segments. Nets, Segments and Junctions are phenomena. Identifiers are concepts — as are the next many notions.

(22.) Paths are triples of (junction,segment,junction) identifiers, and routes are sequences of paths. (23.–25.) 'paths(n)' expresses all paths of a net 'n'. (27.–30.) A route is a finite sequence of paths such that adjacent sequence triples designate adjacent paths, that is, paths sharing a junction identifier (well-formedness). (32.–35.) 'routes(n)' expresses all routes of a net, 'n', that is, a possibly infinite set since routes my be cyclic. Any path is a route. If r and r' are routes such that the last path of r shares last junction identifier with the first junction identifier of the first path of r', then their concatenation is a route.

```
type
22. P = Ji × Si × Ji
value
23. paths: N → P-set
24. paths(n) ≡
25.    {(ji,si,ji′)|s:S,ji,ji′:Ji,si:Si• s ∈ obs_Ss(n) ∧
                               {ji,ji′} ∈ obs_Jis(s)∧si=obs_Si(s)}
type
26. R = {|r:P*•wf_R(r)|}
value
27. wf_R: P* → Bool
28. wf_R(r) ≡
29.    ∀ i:Nat • {i,i+1}⊆inds(r) ⇒
30.      let (_,_,ji)=r(i), (ji′,_,_)=r(i+1) in ji = ji′ end
31. routes: N → R-infset
32. routes(n) ≡
33.    let rs = {⟨p⟩|p:P•p ∈ paths(n)}
34.           ∪ {r^r′|r,r′:R•{r,r′}⊆rs∧wf_R(r^r′)} in
35.    rs end
```

(36.) The state of a segment is a either an empty set, or a singleton set, or a set of two "reversed" pairs of the identifiers of the two junctions to which the segment is connected. An empty set designates that the segment is closed for traffic. A single pair $(j_i, j_{i'})$ designates that the segment is open for traffic from the junction identified by j_i to junction identified by $j_{i'}$. A double pair designates that the segment is open for traffic in both directions. (37.) The state of a junction is a set of pairs of identifiers of the segments that are connected to that junction. If (s_{i_j}, s_{i_k}) are in the state then it means that traffic can traverse the junction from the segment identified by s_{i_j} to the segment identified by s_{i_k}. (38.–39.) From a segment [junction] one can observe the state of the segment [junction]. (40.–43.) One can speak of and observe the state space of a segment [junction]. (44.–45.) The current state of a segment [junction] lies in the state space of that segment [junction].

```
type
36. SΣ = {|jip:(Ji×Ji)-set •
              card jip≤2∧card jip=2⇒(ji,ji′)∈ jip⇒(ji′,jip)∈ jip∧ji≠ji′|}
37. JΣ = (Si×Si)-set
value
38. obs_SΣ: S → SΣ
39. obs_JΣ: J → JΣ
type
40. SΩ = SΣ-set
41. JΩ = JΣ-set
value
42. obs_SΩ: S → SΩ
43. obs_JΩ: J → JΩ
axiom
44. ∀ s:S • obs_SΣ(s) ⊆ obs_SΩ(s),
45. ∀ j:J • obs_JΣ(j) ⊆ obs_JΩ(j)
```

(46.) Time, vehicles and lengths are introduced. Time is considered a dense, ordered set. (47.) Fractions are reals in the inclusive range 0 to 1. (48.) A (vehicle) position is either at a junction or some fraction a segment (down, from fj_i to tj_i, implicitly) identified by this pair of identifiers of the junctions to which the segment is connected. (49.) Real [observed] traffic, rTF [oTF], is a continuous function [discrete map] from time to pairs of nets and positions of vehicles. (50.) Segments have lengths. (51.–58.) The lengthy axiom expresses one aspect of traffic: that positions are indeed positions of the net.

```
type
46. T, V, L
47. F = {|f:Real•0≤f≤1|}
48. P == mkP_at_J(ji:Ji) | mkP_along_S(fji:Ji,f:F,tji:Ji)
49. rTF = T → (N × (V →m P))
50. oTF = T →m (N × (V →m P))
value
51. obs_L: S → L
axiom
52. ∀ tf:(rTF|oTF) •
53.    ∀ t ∈ dom tf •
54.       let (n,vps) = tf(t) in
55.       ∀ p:P • p ∈ rng vps ⇒
56.       case p of
```

```
57.       mkP_at_J(ji) → ji ∈ obs_Jis(n),
58.       mkP_along_S(jf,_,jt) → {jf,jt}⊆obs_Jis(n)
        end end
```

The formalisation of other aspects of traffic are left as an exercise: (i) that real [observed] traffic is truly continuous [monotonic] (including: moving from open segments through "open" junctions onto open segments and: vehicles do not "jump around"), (ii) that there are no "ghost" vehicles (that is, that a vehicle which is in the traffic at two distinct times are in the traffic at all times in between these, etcetera.

Real Highway Transportation Nets

The model above has been simplified to fit this chapter. A more realistic model, one that can serve as a domain model for the requirements of an automated highway, would model (i) multi-lane highway segments and junctions, (ii) vehicle lane positions, (iii) vehicles changing lanes, etc.

4.3.3 Requirements

General: Co-ordination and Function Requirements

Now, depending on requirements for ubiquitous automated highway computing, a number of co-ordinated requirements prescriptions need be studied, be developed, and the resulting formal prescriptions analysed — including establishing a theory of the requirements. By "co-ordinated requirements prescriptions" we mean a "set and a co-ordination" which, as a whole, prescribe requirements to an automated highway computing system, but such that individual members of the set ("function requirements") each solve a well-prescribed problem in the context of the co-ordination software (prescribed by the "co-ordination").

We think of the co-ordination requirements and the corresponding software to represent the essence of ubiquity.

Ubiquitous Requirements: Some Speculations

In this section we shall sketch some rather preliminary ideas on some aspects of requirements to automated highways — such aspects, it seem, which may cast light on what we may be an issue of, an "ingredient" in requirements to ubiquitous computing systems.

A first set of requirements could be those for the computerise support of "platooning", that is, of cars joining, driving as part of, and leaving platoons, that is, sequences of cars going, say at some high speed, from one place to another — with cars possibly leaving one platoon to join another, etc. We refer, for example, to *Safe Platooning in Automated Highway Systems* http://-www.path.berkeley.edu/PATH/Publications/PDF/PRR/97/PRR-97-46.pdf.

Let us, more generally, perform an experiment. Assume traffic “rolling” down a highway. Traffic consists of a number of vehicles passing down a number of opposite lanes, possibly at different velocities, possibly changing lane, overtaking, possibly slowing or speeding up.

Now what might “automation” mean in this case? Well it could mean either of many different things. All vehicles could be coerced into travelling at “same” velocity, or “as much as possible” close to (from below) a desired mean velocity, or be “free to change velocity”, and/or be constrained to “near same” (mean) acceleration and deceleration. Etcetera.

Whichever of these are chosen, it seems that each vehicle is expected to register, as is expected done by their drivers today, some composite “picture”, a “local state”, of the traffic “around” that vehicle, to “some depth”, that is, to some distance from that vehicle. Based on such a local state the driver today makes driving (steering, acceleration, deceleration, etc.) decisions. Today drivers do that — thus basically without consulting other drivers’ plans (including their local states). In an automated highway computing system one might expect individual as well as communal driving decisions and hence actions to be based on summaries of local states “around” each vehicle, or even global such summaries. Classically one might be able to express such decision outcomes as the result of solving a large number of for example differential equations, or, alternatively, some similar sets of fuzzy control expressions.

One is here reminded of:

- Victor I. Varshavski and Dmitrii Aleksandrovich Pospelov: *Puppets without strings: Reflections on the evolution and control of some man-made systems*, Mir Publishers, Moscow, 1984 (English edition: Mir Publishers, 1988, 294 pages).

An analysis of what is thought to be needed to express (i.e., to monitor and, later, control) local (and “less local”) states in domain descriptions and, subsequently in requirements prescriptions should give a rather precise idea about the need for computing system sensors and actuators.

Everyone can speculate: mention this and that kind of sensors and actuators, but only a thorough domain analysis and a “therefrom derived” requirements analysis can tell us exactly the properties we need of those sensors and actuators.

Further on Requirements

In this chapter we shall not further attempt to formulate further requirements. To do this properly requires, in our mind, very close collaboration with the Cambridge University group of Prof. Jon Crowcroft. We refer to:

- [18] http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/Manifesto/fp-automatinghighway.html

- [19] http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/Manifesto/road.pdf

It is hoped that some of the groups involved in Domain Theories will also want to become involved in the Requirements-related projects.

And it is expected that this, the Requirements effort, will be closely collaborating with the Ubiquitous Computing Automated Highway project as referred to above.

4.3.4 Software

We postulate, as we did in Sect. 4.3.3 on page 90, that the resulting computing system software consists of the co-ordination (the ubiquitous) software and its many "isolated" packages. We speculate that these "isolated" software packages can each be developed in isolation — provided that a stable interface to the co-ordination software can be established — and that they can each be verified in, i.e., assuming this context.

We shall not speculate further on the software possibly emanating from this proposed project other than the above single paragraph.

4.3.5 Overview of Project Components

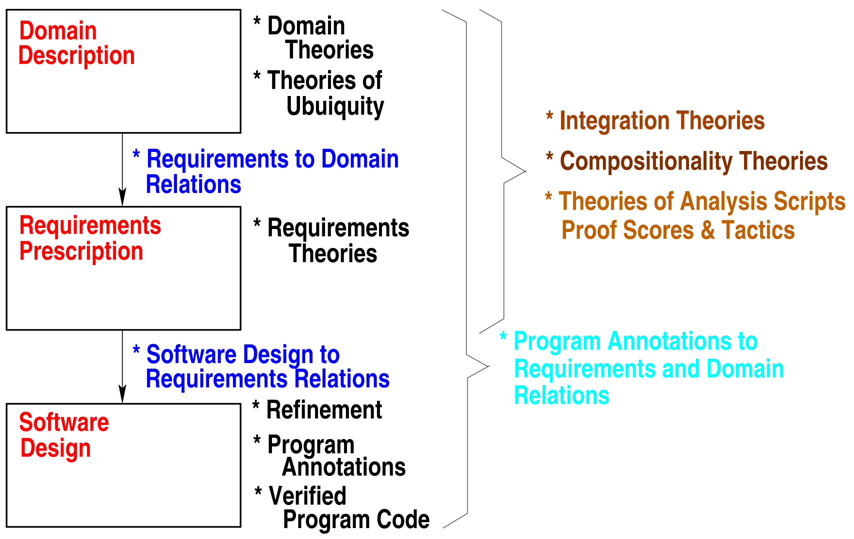

Domain Description
* Domain Theories
* Theories of Ubuiquity
* Requirements to Domain Relations
Requirements Prescription
* Requirements Theories
* Software Design to Requirements Relations
Software Design
* Refinement
* Program Annotations
* Verified Program Code
* Integration Theories
* Compositionality Theories
* Theories of Analysis Scripts Proof Scores & Tactics
* Program Annotations to Requirements and Domain Relations

Fig. 4.1. Project Components

4.4 Challenges

There is an overall challenge:

> **Challenge # 0. Verified Software for Ubiquitous Computing:** The overall challenge is to both understand the concept of ubiquitous computing and contribute to the verified software concept.

And there are a number of constituent challenges. The latter contribute to the former. We formulate some of the latter as we expect different research groups around the world to more readily "catch on" to the latter challenges than to the former.

4.4.1 Commensurate Specifications

The example given in Sect. 4.3.2 on page 86 was expressed in the RSL formal specification language [31–33, 44, 101, 104, 106]. There are other formal specification languages. To wit: Alloy [146], B [1, 71], CafeOBJ [89, 90, 99, 100], Casl [11, 78, 184, 185], CSP [137, 138, 218, 222], VDM-SL [55, 56, 95, 96], RSL [31–33, 44, 101, 104, 106], or Z [132, 133, 229, 230, 242] are some of the better known.

Some (verified software for ubiquitous computing project groups) will specify their domains or requirements in one of these languages. Others in another of these languages.

> **Challenge # 1. Commensurate Specifications:** It is therefore a challenge to show relations between some pairs of specifications purporting to describe "the same thing", that is, to embody the comparable theories.

4.4.2 Integrated Specifications

The example given in Sect. 4.3.2 on page 86 captures only some basic entities of transportation nets and some basic characteristics of traffic (a behaviour).

To capture such phenomena as the individual behaviour of a vehicle, the monitoring and control of traffic and of traffic signals (i.e., the changing of segment and, in particular junction states, would probably be a mistake. More suitable formal notations should be used. They should then be used in conjunction with some formal textual notation, such as those mentioned in Sect. 4.4.1.

Which other such possible, "more suitable" notations exists, today, Summer 2006? We suggest that perhaps either one or more of the following may be used: `DC` (for duration calculus, a temporal logic specification language) [247, 248], message or live sequence charts (`MSC`s, [142–144], or `LSC`s, [80, 128, 153]), `Petri nets` [148, 199, 210–212], `Statecharts` [123, 124, 126, 127, 129], `TLA+` [155, 156, 175, 176], or other. Some of these other may "be in the works" as of now: being researched and put forward as means for modelling, say, statistical, continuous, chaotic or other phenomena.

Whenever the textual notations of `Alloy` [146], `Event B` [1,71], `CafeOBJ` [89,90,99,100], CASL [11,78,184,185], `VDM-SL` [55,56,95,96], `RSL` [31–33, 44,101,104,106], `Z` [132,133,229,230,242], or other, are combined with those of, for example, `DC` [247,248], `MSC` [142–144], `LSC` [80,128,153], `Statecharts` [123,124,126,127,129], `TLA+` [155,156,175,176], or other, we say that the result is a set of two or more interfacing specifications. The interface may be manifest in for example the following way: In specification $S_A^{L_i}$ some identifiers $id_1, id_2, \ldots, id_n$ are claimed, semantically, to be somehow 'equivalent' with identical identifiers in specification $S_B^{L_j}$. When this is the case, an obligation arises, namely to show that a suitable equivalence (or satisfaction) relation can be established. Reasoning in one notation "must carry over" into "interfacing" parts expressed in another specification.

Substantial research is done in this are as witnessed by, for example, [7, 65,69,111,216].

A real core issue for this project to study is the relationship between the kind of textual specification languages extensively listed and referenced above and the calculus of classical mathematics — as for example used in automotive engineering: differential equations and the like. It is a challenge to provide as seamless transition from specification in the form of classical applied mathematics, including operations research, and the newer formal specification languages.

Challenge # 2. Integrated Specifications: For specific such interfacing (and thus commensurate) specifications or for general uses of their specification languages, suitable equivalence (or satisfaction) relations must be established (formulated and formally verified).

4.4.3 Domain Theories

By a domain theory we understand a domain description and a number of theorems that can be proved to hold of what the description models. These theorems are (obviously) proved in the proof system of the notations in which the domain description is expressed.

What could such "interesting" theorems be about? Well, let's try some possible, or potential examples:

- *Kirchoff's Law for Railways:* Assuming that train timetables are modulo 24 hours and that trains run on time, then for any 24 hour contiguous period (i.e., interval) and for every train station it is the case that the number of trains arriving at a station minus the number of trains ending their journey at that station plus the number of trains starting their journey at that station equals the number of trains leaving that station.

- *"No Ghost Trains":* In a domain description of highway traffic along a segment, then for any consecutive times t and t'' and for any vehicle which is in the traffic at those times it is the case that the vehicle is also in the traffic at all times between t and t''.
- *"God does'nt Play Dice":* We assume that the model for traffic is continuous, or at least monotonic over time (either a dense set of points, or, respectively, a discrete set of points with clearly defined "next" time successors). We assume a single lane, one traffic direction segment. If at time t vehicle v_i is "after" vehicle v_j in the traffic along the segment, then at a "next" (i.e., the successor) time t' vehicles v_i and v_j have not reversed their position.

This leads us to formulate:

> **Challenge # 3. Domain Theories:** So, for the domain of transportation (incl. highway transportation) the challenge is to identify and prove "interesting" theorems.

A source of inspiration as to what kinds of theorems to look for in any domain theory seems to be [94]. It is hard for me to think that laws of man-made domains are that fundamentally different from those of nature.

4.4.4 Analysis Scripts: Proof Scores and Tactics

In connection with research around CafeOBJ some insight is gained with respect to "standard" scripts according to which CafeOBJ specifications can, or should be analysed. These are called proof scores. Proof scores are first class CafeOBJ objects. In that sense they are different proof tactics pursued as early as the early days of LCF [110, 179, 180, 198, 202] and is also a core subject of PVS [195, 196] proof work [197].

> **Challenge # 4. Analysis Scripts: Proof Scores and Tactics:** So, for any domain or requirements formalisation the challenge is to find a suitable, reusable set of analysis scripts.

4.4.5 Ubiquity

"How are the computers in the automated highway system configured and controlled? What tools are needed for design and analysis of constantly adapting and evolving computing system? What theories will help us to understand its behaviour? How do we manage such systems, or even understand them? How do people interact with them and how, in which ways, does this new pervasive technology affect us?"

These are some of the many questions that the Ubiquitous Computing — Science and Design — Grand Challenge initiative is raising.

Challenge # 5. Ubiquity: The proposed Verified Software for Ubiquitous Computing project is expected to contribute to at least the tools are needed for design and analysis of constantly adapting and evolving computing system and to their underlying theory.

4.4.6 Compositionality

We have made some vague, non-committed references to 'decomposition' of the domain description, the requirements prescription and the software design tasks (the latter in terms of 'isolated' software packages composed with 'co-ordination software'.

Challenge # 6. Compositionality: Does the established computer science work on compositionality, for example as illustrated by [83–88], apply here? Or must new concepts be developed?

This version of the compositionality concept apply specifically to components and objects.

We refer to a recent paper on compositionality: [48].

4.4.7 Formalisation of Machine Requirements

Machine requirements such as performance, dependability, maintainability, platforms, etc. are not easily formalisable, and, when formalisable (with some difficulty) it is not yet well understood how one "derives" implementations correct wrt. the machine requirements.

Challenge # 7. Formalisation of Machine Requirements: Ubiquitous computing has very much to do with machine requirements so a special challenge, in the small, and, it appears, as many distinct challenges, is to discover, study and make practically useful principles, techniques and tools of machine requirements and their implementation.

4.4.8 Requirements to Domain Relations

We here claim, and show in of Vol. 3 [33, Chap. 19] and in [41], that core issues of requirements, what we term the domain requirements and the interface requirements, can be systematically deduced, in interaction between the domain description, the requirements engineers and the client.

Challenge # 8. Requirements to Domain Relations: Although the engineering seems clear it also seems desirable to have a more formal understanding of the relations between requirements prescriptions and domain descriptions, and for a suitable variety of cases — for example such as appears to be offered in the context of software for ubiquitous computing.

4.4.9 Software Design to Requirements Relations

Many of the relations that are implied here are those that show up, it appears — and to this author certainly — in for example Michael Jackson's seminal work on 'Problem Frames' [147].

> **Challenge # 9. Software Design to Requirements Relations:** So we pose as a challenge to more precisely understand these relations — including formally.

4.4.10 Annotation to Requirements and Domain Relations

A major cornerstone of the VSTTE ideal of program correctness is that of being able to have a machine prove correctness of program code with respect to program code annotations.

We assume that programs are annotated, and that the annotations contain elements

(1) some of which "clearly" relate to $\mathcal{R}$equirements prescriptions,
(2) others (therefrom possibly distinct (?)) of which "clearly" relate to $\mathcal{D}$omain descriptions, and
(3) yet others of which (perhaps the three kinds of annotation elements "overlap") "clearly" relate to the programming language.

The three, as we postulate, "clearly" identifiable parts appears (and this is yet a postulate) to express properties that then appears to only be provable in relation to respective semantics: (1′) The domain theory, (2′) the requirements theory and (3′) the programming language semantics or proof system.

In view of our suggestion:

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

we pose the challenge:

> **Challenge # 10. Program Annotations:** Do the postulates above (1,1′–2,2′–3,3′) hold. How can we express the annotations so as to [most?] clearly designate the three relations? And how is the verifying compiler going to handle all this?

4.5 Some Observations

Rather unusual for a paper of this kind we bring some extensive comments on an earlier version.

4.5.1 Michael Jackson's 22 March, 2006 Observations

I would be inclined to go further, in a direction that may offer some insight but would surely be fruitless for influencing the GC. A major difficulty of developing a software-intensive system that interacts with a non-formal physical or human domain is that formalisation is always only approximate. If the system is critical it becomes vital to obtain a very good approximation indeed, and this cannot, in general, be achieved by elaborating the description of domain properties. Instead it becomes necessary to structure a set of domain descriptions, with their attendant requirements and subproblem machines, so that they form an assemblage that in total is a good enough approximation. A simple case is a cascading structure in which the highest element in the cascade describes an idealised domain in which everything works 'as it should' and other elements describe a less idealised domain in which various 'faults' invalidate parts of the assumptions on which the highest element is based.

More generally still, a realistic problem is an assemblage of many subproblems, each addressing a partial requirement on the basis of a partial view of the problem world. The composition of these subproblems raises many concerns such as conflict, interference, incommensurability, and so on, which must be addressed if the subproblems and their solutions are to be combined into the desired system. This complexity of composition inevitably leads to a complexity of software structure. Without addressing this aspect of software structure the developer cannot know whether or not the behaviour of the programmed machine has a good likelihood of ensuring satisfaction of the problem requirements.

One manifestation of this difficulty is that the 'verifying compiler' version of the GC can be seen to depend on the program having an appropriate structure. If the specification is to be embedded in the program the form of pre- and post-conditions and invariants, the program must possess a structure that can accommodate the specification without introducing so many specification variables that the specification itself becomes unintelligible. To take a trivial and grossly simplified example: if a word processor specification includes an invariant on the document that is to be processed, there ought to be a program component corresponding to the whole document. If there are, instead, only components corresponding to operations on the document there will be no place in the program where the document invariant can be written.

4.5.2 Tony Hoare's July 31, 2006 Observations

The first subsection below is from "random paragraphs" drafted by Tony Hoare, July 2006: Theory for Verified Software, received in an e-mail.

The second subsection below is from an e-mail of It predates "random paragraphs". The bulleted items of the first subsection were part of the June e-mail from which we only bring the advice.

Random Paragraphs

In any practical software project, theory begins to play a role long before the project starts. Its initial role is to construct a mathematical model of the real world domain in which the projected software will be deployed, without making any commitment to a particular choice of functionality or of implementation technology.

An academic example of a useful domain model is one that describes a range of procedures for the organisation of different kinds of scientific meeting [Fisler]. An industrial example is the domain of scheduling algorithms [Doug Smith]. Domain models (often called ontologies) are of wide application in various e-sciences, to standardise terminology and logical structure of observable data, and to help solve consistency problems between experimental data bases. A domain model may make a contribution towards the analysis of systems composed of other systems (including commercial off-the shelf software). A domain model can provide a framework for classification of a range of software and hardware standards, relating them to each other and to their common purposes. A good example is the ISO Reference Model for Open Distributed Processing. Other serious applications (perhaps too serious for experimentation!) are in the service industries, including retail sales, transport, the health service, and even parts of e-government.

- *A well-structured domain model will describe all aspects of the real world that are relevant for any good software design in the area. It describes possible places to define the system boundary for any particular project.*
- *A domain model makes explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.*
- *A domain model describes the whole range of possible designs for the software, and the whole range of technologies available for its realisation. It is a suitable basis for the Æproduct lineÇ approach to software design and construction.*
- *A domain model provides a logical framework for formalisation and full analysis of a range of possible customer requirements; it reduces the risk of contamination of requirements by the choice of a particular technology of implementation.*
- *A domain model enumerates and analyses the decisions that must be taken earlier or later any design project; it identifies those decisions that are independent and those that conflict, those decisions that must be taken early, and those that can be postponed. Late discovery of feature interactions can be forestalled.*

Research on domain modelling dates back at least as far as the development of VDM, Z, RAISE, and there are now many mathematical formalisms available for describing both the whole and various aspects of a domain model. For example, there are formalisms based on set theory (Z), process calculus (Pi), *abstract algebra (ASM), graphical notations (UML), and mixtures of these. Each of these will be suitable for describing different aspects of the domain. It will be a challenge to use them in an appropriate combination, with formal checks of consistency between them. Work on combined use of formalisms may motivate and guide research toward unifying theories of programming.*

An even more important challenge will be to validate a domain model. This may be done by a broad survey of existing implementations, showing that they can all be classified as examples (at least, in approximation) of the model. The model may suggest useful experiments on the real world. The extra effort devoted to the generality of a domain model will be very worth while, because it can be done long in advance of any particular project, and because domain models are much more re-usable than any of their specific implementations.

Comments, I of III

You will find the above issues covered extensively in [33].

Advice[3]

I suggest you give more prominence to these (ed.: the above bulleted) advantages.

A large part of the serious work of domain modelling is a check of the correspondence between the model and the real world. I think your plans should include careful design of the earlier experiments that make this check. This would increase the value of your project far more than actual implementation of any particular one of the possible designs. This particular example project would have to be conducted in close (maybe too close) association with the automobile industry, who are already developing the technology and conducting the experiments – and even delivering the products!

I think your project will have more impact and attract more researchers if you put forward a small range of example projects. They should include a domain taken from the home, one taken from the law (perhaps election law), and perhaps a fragment of an application in (local) government. Different researchers will find different domains more attractive.

The whole point of your example projects is that each of them will deliver results that are much more general than any particular implementation (whether verified or not). Don't spoil this by implementing just one of the possibilities. If you are going to implement anything it should be a program development tool or generator that can be applied to the whole class of implementations, including perhaps an experimental environment for evaluating alternative designs.

[3]Hoare, continued

Your document contains an example of a fragment of a model written in your favourite language (presumably). Different aspects of the model and design might be well expressed in different conceptual frameworks, for example, process calculus, the pi calculus, algebra, UML, biographical systems, etc. Even pictures can be quite helpful. The experienced mathematician is quite skilled in living simultaneously in a variety of modelling presentations. Such diversity should be welcomed, and theorists should be invited to develop techniques and tools that ensure consistency and coherence of the various aspects of the model. Thus domain modelling will provide motivation and source material for another Grand Challenge – Unifying Theories.

In your next step, I would advise you to 'choose your team'. Who are the world-wide leaders in the field whom you would most like to recruit to work on the project? Who are those whose antagonism to the project would be most damaging? The biblical number of apostles is twelve. When you have the leaders helping you, then you can cast your net to the whole community.

Comments, II of III

The current paper has already presupposed this advice.

4.5.3 Robin Milner's July 31, 2006 Observations

The E-mail Header

Thank you for putting so much effort into joining two GCs – GC6 and GC2/4 – in the context of automated traffic. I largely agree with what you are saying, and in the note at the end I try to map your way of thinking onto how we have already been thinking in GC2/4. I also comment inconclusively on how we would coordinate this junction; and finally I mention a few things that occurred to me while reading your piece.

I hope we can push this further. I'm puzzled at present as to who on the GC2/4 side will be proactive in the coordination, as opposed to proactive in their own research strand. This whole thing depends on the chance of finding the right individuals!

Here's one thought (but read my remarks below): we could avoid setting up a coordination team at the start, and instead – by luck or judgement – find just ONE (sub)-project that can arise between us because the right individuals need each other. What do you think?

Automated Traffic

You propose a project of 7-10 years on automated traffic. This is a valuable specialisation of the Ubicomp Grand Challenge; the GC must surely involve such special goals. The GC also aims for generic engineering principles and theories that cover a range of specialisations, and this is evident from the *general form in which its goals are stated. But I can't see any way to reach these goals other than specialisations like the traffic one.*

*The Ubicomp GC proposes that we identify foothill 'topics', or better 'activities', and for each activity there would be many (coordinated we hope, but loosely so) foothill 'projects', each perhaps lasting 3-4 years and tackling a specific aspect of the topic, taking care to involve at least some mix of the three GC themes: Experience, Design and Science. So I see your 'foothill project' more as a 'foothill activity'. This isn't just a matter of what words we choose; it emphasizes that to advance understanding of the 'topic' (by coordinating the *activity*) we need to identify many 'projects' with aims achievable in 3-4 years.*

With this in mind, we can think of your proposal in Section 3 as needing to be specialised to specific projects. (See Sect. 4.5.3.) Formulating achievable goals for these (sub)-projects is surely a creative task in itself. But, for example, Jean Bacon has one such project already running, involving traffic in Cambridge. I don't know the project as well as I should, but I believe it involves knitting together at least two of the GC themes: Experience (what do citizens and the City Council actually want?) and Design (what event-based model can express alternative designs to achieve this experience?)

One can think of this kind of project as helping to formulate an appropriate dynamic model, rather than as presupposing an already perfect such model.

A different project (prompted by Jon Crowcroft) could involve the comparison of two different design choices for a specific application: 'distributed control' (each car has its say) with 'centralised' control (the highway makes all the decisions). To do this comparison one still has to postulate a model in which both designs can be expressed; so the project can be regarded proposing and testing the Model as well as identifying the differences in Experience provided by the two designs.

Other projects can involve theoretical aspects of such models: how to express designs (in some mix of logics, process calculi, differential equations, and languages based on these); how to analyse the state-space of a design (by pencil and paper, or by model-checking) to ensure that some requirements are met; how all of this can be done stochastically; etc. These theoretical aims pertain to ALL ubicomp systems of course; but theories, like engineering principles. designs, get established through specific application.

Coordinating a Foothill Activity

In GC2/4 there is debate about how a foothill activity can be coordinated, and woven into the coordination of the whole GC activity. This is hard; and presumably it's even harder to do between two GCs! You already address this in 6.4.2 where you expect '20 to 30 groups' to be enlisted. The big question is: how do they interact? I'm sure it depends, as you are hinting, on gathering a steering group who really believe in the coordination as much as in their own specific research. This group ought to be more or less equally represent the *two GCs. What will not work is to try to do this top-down! It follows that, like democracy, it sometimes cannot be done at all. But we can continue to try.*

Specific Comments on DB's Early Draft of This Paper

- *At the start of 4.1.5 you introduce domain models as abstracting reality. I think one of our main problems is that they have to model 'future' reality, or future possible realities. That's where we have not the luxury of a natural science – where the reality is not (or is only very slowly) modified by the model!*
- *This raises a question-mark about the Dogma (Sect. 4.2.1), first bullet, where you say "First one must establish a ... domain theory". In GC2/4 we imagine that the model (or theory) will only be arrived at by the research; so it's one of the ultimate aims of the GC, not a first step.*

Comments, III of III

You will find that Milner's comments are addressed extensively in [33].

4.6 Project Management

4.6.1 Self-funding

This project frame proposal does not imply central funding. That is, each partner must find own funding. Each partner is expected to contribute at least 6 person months of research per year — and typically 6-18 such months per year.

4.6.2 A Patchwork of Overlapping, Individualistic Contributions

Challenges # 1–# 10 indicate rather clearly that there is room for individual groups to contribute to just one of them. It is then hoped, through "gentle coercion and persuasion" to get these or other individual groups interested in "filling the gaps".

Challenge # 11. Management: It is indeed a management challenge to get as many groups to work in order to "fill" the programme, so that all challenges (# 1–# 10) are taken care of, and it is indeed a management challenge to make sure that the patchwork hangs together, that is, to fulfill challenge # 0

4.6.3 Project Plan

The project plan can only be very loose: This project is not centrally funded, it is a research project with very many open issues, requires individuals of widely different skills that must be "fused", etcetera.

The best we can suggest at this time is for the reader to take a look at Fig. 4.1 on page 92 and at our sketch "milestone" plan, Sect. 4.6.5.

4.6.4 Resource Requirements

The project is characterised by a diversity of issues, from domains via requirements to software; from integration and composition of medium to large scale specifications expressed in many "styles", transitions from domain via requirements to software design specfications, the establishment of formal relations (for these transitions), and the verification of annotated code. This calls for many groups to contribute to the project, each with their specialty, some working already on other foothill projects of either of the two underlying grand challenges.

We therefore expect a number from 20 to 30 groups to be enlisted in this project for it to succeed.

Each such group may be from one person working at least half time to several (3–4) such persons. And for typically 3–4 years.

It is expected to hold an annual "Verified Software for Ubiquitous Computing" workshop — probably as part of some event of the two underlying Grand Challenges. It is deemed mandatory that each group participate three or four consecutive times in these workshops.

4.6.5 "Milestones"[4]

These are the short, medium and long term goals to be reached:

- Short Term Goals: Summer 2007
  The gathering of at least some 20 or more reasonably committed and reasonably strong research individuals (groups), the holding of a kick-off workshop, and hence the start of the project.
- Medium Term Goals:
  In chronological order we list one set of "deliverables", one could call them the "on the path from domains to verified software" deliverables:
  - ⋆ Domain Specifications 2008–...
  - ⋆ First Set of 1–2 Requirements 2009–...
  - ⋆ Next Set of 4–5 Requirements 2010–...
  - ⋆ Requirements to Domain Relations 2011–...

[4]Please recall that this document was written early 2006 for presentation in October 2006.

  - ⋆ First Set of 1–2 Software Designs 2012–...
  - ⋆ Next Set of 4–5 Software Designs 2013–...
  - ⋆ Software Design to Requirements Relations 2013–...
  - ⋆ Program Annotations to Requirements and Domain Relations 2015–...
  - ⋆ First Set of 1–2 Verified Program Code 2016–...
  - ⋆ Next Set of 4–5 Verified Program Code 2018–...

  We then list supporting theory "milestones". These are more thought of as individual contributions (i.e., papers) that are produced in a hopefully steady flow:

  - ⋆ Domain Theories 2009–...
  - ⋆ Requirements Theories 2011–...
  - ⋆ Integration Theories 2009–...
  - ⋆ Compositionality Theories 2009–...
  - ⋆ Theories of Ubiquity 2009–...
  - ⋆ Theories of Proof Score/Scripts &c. 2009–...

- Long Term Goals:
  - ⋆ Verified Software for Automated Highways 2020–...

4.6.6 Project Board and Management

Obvious one must have "such a thing": board, project leader, etc. Tony Hoare's and Robin Milner's comments in Sects. 4.5.2 and 4.5.3 are all relevant in this context.

4.7 Conclusion

We have sketched a foothill project within the two grand challenges of 'VSTTE: Verified Software, Theories, Tools and Experiments', and 'Ubiquitous Computing', the latter specifically in the context of 'Automated Highways'. Both of the two grand challenges had set somewhat different scopes and spans for their expected activities. The current proposal attempts to bring the two together on "one plate".

4.8 Acknowledgements

Discussions over the last six months (fall 2005 – spring 2006) with (in chronological order) Tony Hoare, Cliff Jones, Michael Jackson, Kokichi Futatsugi and Robin Milner are much appreciated.

5

The Triptych Process Model[1]

Process Assessment and Improvement

Abstract

The triptych[2] approach to software engineering proceeds on the basis of carefully monitored and controlled possibly iterated progression through domain engineering and requirements engineering to software design.

In this paper we will outline these three phases, show the many stages of development within each and also indicate the many steps within each stage. We will ever so briefly touch upon informal narration and formal description (prescription and specification) of domains (requirements and software designs), and the verification (theorem proving, model checking and testing) and validation of domain descriptions (requirements prescriptions and their relations to domain descriptions, as well as the software design specifications and their relations to requirements prescriptions). The importance of process management and its relations to software process assessment (SPA) and software process improvement (SPI) will then be underscored. Our measuring "stick" is that set up by Watts Humphrey's capability maturity model (CMM). We will suggest and discuss seven assessment and eight improvement categories. In closing we will have some few words to say about software procurement.

5.1 The Triptych Dogma

5.1.1 Background

In the past, as exemplified in major software engineering textbooks [109, 139, 200, 205, 228, 240], software engineering focused on requirements engineering and software design. The triptych dogma extends the two (requirements engineering and software design) into three (domain engineering plus the two phases already mentioned).

[1]This is an edited version of [38]. Invited talk at JASPIC 2006 (Japan Association for Software Process Improvement) meeting October 12-13, 2006 at Tsukuba.

[2]The term 'triptych' covers the three phases of software development: domain description, requirements prescription and software design.

5.1.2 The Dogma

- Justifying requirements prescriptions:
  - ⋆ Before software can be designed
  - ⋆ we must understand the requirements.
- Justifying domain descriptions.
  - ⋆ Before requirements can be prescribed
  - ⋆ we must understand the domain.
- Justifying the triptych:
  - ⋆ First analysing and describing the (application) domain,
  - ⋆ then analysing and prescribing the requirements, and
  - ⋆ finally analysing and specifying the software design and code.

5.1.3 New Aspects

The relatively new aspect of software development is here 'domain engineering'. This new aspect "translates" into a number of new methodological aspects of domain and requirements engineering. The next, the major section will survey these aspects. All of this is covered extensively in volume 3 of the three volume book [31–33]. All figures and tables in this chapter are re-used from [33](by permission from Springer).

5.2 The Triptych Process Models and Documents

5.2.1 Common Aspects

Process Models

The triptych process model is the composition of three process models: one each for domain engineering, requirements engineering and software design. We hint at this composition in Fig. 5.1 on the next page.

The internals of the three boxes (i.e., phases of development) of Fig. 5.1 on the facing page are outlined in Figs. 5.4 on page 112, 5.8 on page 116 and 5.9 on page 117, respectively Fig. 5.11 on page 119.

The DO edges of Fig. 5.1 on the next page enter topmost boxes of respective Figs. 5.4 on page 112, 5.8 on page 116 and 5.11 on page 119.

The REDO edges of Fig. 5.1 on the next page enter whichever boxes of Figs. 5.4 on page 112, 5.8 on page 116 and 5.9 on page 117, respectively Fig. 5.11 on page 119 that are found to be most appropriate. (More on this later.)

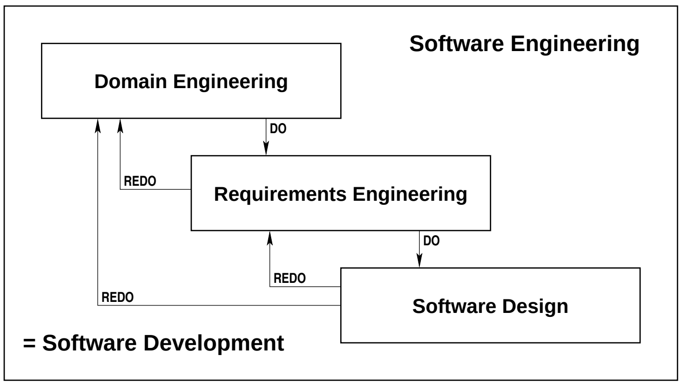


Fig. 5.1. A simplified view of the triptych process model

Documents

Common to all three phases of software development are that they primarily manifest themselves in documents. Figure 5.3 on page 111, Fig. 5.5 on page 114, Fig. 5.6 on page 115, Fig. 5.7 on page 116, and Fig. 5.10 on page 118, to be commented later, illustrate the breadth, depth and quite substantial number of such resulting documents. And common to each set of such documents is the more-or-less administrative "working out" of *information document*, cf. items 1 of Fig. 5.3 on page 111, Fig. 5.5 on page 114, Fig. 5.6 on page 115, Fig. 5.7 on page 116, and Fig. 5.10 on page 118. Figure 5.2 extracts item 1. from Fig. 5.3 on page 111, Fig. 5.5 on page 114, 5.6 on page 115, Fig. 5.7 on page 116, and Fig. 5.10 on page 118.

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
   (m) Plans
      i. Project Graph
      ii. Budget
      iii. Funding
      iv. Accounts
   (n) Management
      i. Assessement
      ii. Improvement
         A. Plans
         B. Actions

Fig. 5.2. Informative documents

Let us briefly review the pragmatics of Fig. 5.2. In any of the three phases of development, domain engineering, requirements engineering and software design, the information implied by the table-of-contents of Fig. 5.2 on the previous page must be carefully worked out. Take items ‘Assumptions and Dependencies’, and ‘Implicit/Derivative Goals’. The description, prescription or design work to be done in the phase to which the information documents apply rely on assumptions and dependencies. These must be fully understood, hence documented before any proper development takes place. Consider items ‘Current Situation’, ‘Needs and Ideas’, and ‘Concepts and Facilities’. The current situation which apparently warrants the proper development must be recorded. It might change thus necessitating change of development. Development — of whichever of the three phases — would not be undertaken unless someone, the customer and/or the developer, has some needs for the (approximately) expected results of the development, and, as well, has some ideas as how (methodologically) to basically develop whatever is to be developed (a domain description, a requirements description, a software design). The customer and/or developer also, initially have made some thoughts of the core concepts and facilities around which the development is expected to take place. All of this need be properly recorded as any review of project status occurs in the pragmatic context of ‘Assumptions and Dependencies’, ‘Implicit/Derivative Goals’, ‘Current Situation’, ‘Needs and Ideas’, and ‘Concepts and Facilities’.

5.2.2 The Domain Engineering Process Model

We first rough-sketch narrate the stages and steps of the domain engineering development of a domain model, then review the documents that should emanate from such development. Finally we diagram an essence of the narration and the document table-of-contents.

But first some words on domain models.

Domain Models

A main result of domain engineering development, as applied to some specific application domain[3], is a domain model. Domain models are in the form of descriptions. Domain descriptions describe what there is, and as it is. There is no presumption of requirements implied by these descriptions. They are not requirements prescriptions. By analogy, physicists [domain engineers] are describing mother nature [application domains] and engineers [requirements engineers and software designers] are prescribing and implementing requirements.

[3]Examples of domains are: (1) the financial service industry as a whole, (1.1) a bank, (1.1.1) a bank’s mortgage lending business; (2) the transportation industry as a whole, (2.1) a railway system, (2.1.1) an interlocking system; etcetera.

Domain Engineering, A Narrative

The domain engineering triptych dogma, and as argued in Chaps. 8–17 of [33], advocates (item 2.) the following stages of description development (after work on information documents [items 1.a–l] have been duly completed): (2.a) identification of as wide a spectrum of domain stakeholders, (2.b) acquisition of domain understanding, (2.c) establishment (and subsequent, throughout all stages, use and maintenance) of a domain terminology (ontological terms), (2.d) understanding and rough-sketching all relevant business processes, (2.e) domain modelling (all domain facets), and (2.f) the domain model completion (including consolidation). Intertwined with the domain description parts (item 2., subitems (a–f)) are the analysis parts with (3.a) domain analysis aiming at identifying inconsistencies, conflicts and incompletenesses, (3.b) domain validation, (3.c) domain verification, and (3.d) possible work on establishing a domain theory.

The new thing here is all of items 1.–2.–3.

Domain Engineering Documents

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
    - (m) Plans
        - i. Project Graph
        - ii. Budget
        - iii. Funding
        - iv. Accounts
    - (n) Management
        - i. Assessement
        - ii. Improvement
            - A. Plans
            - B. Actions
2. Descriptions
    - (a) Stakeholders
    - (b) The Acquisition Process
        - i. Studies
        - ii. Interviews
        - iii. Questionnaires
        - iv. Indexed Description Units
    - (c) Terminology
    - (d) Business Processes
    - (e) Facets:
        - i. Intrinsics
        - ii. Support Technologies
        - iii. Management and Organisation
        - iv. Rules and Regulations
        - v. Scripts
        - vi. Human Behaviour
    - (f) Consolidated Description
3. Analyses
    - (a) Domain Analysis and Concept Formation
        - i. Inconsistencies
        - ii. Conflicts
        - iii. Incompletenesses
        - iv. Resolutions
    - (b) Domain Validation
        - i. Stakeholder Walkthroughs
        - ii. Resolutions
    - (c) Domain Verification
        - i. Model Checkings
        - ii. Theorems and Proofs
        - iii. Test Cases and Tests
    - (d) (Towards a) Domain Theory

Fig. 5.3. Domain engineering document table-of-contents

Figure 5.3 on the preceding page summarises the plenitude of highly interrelated sets of documents that must all be carefully worked out and carefully correlated.

Domain Engineering Stages and Steps

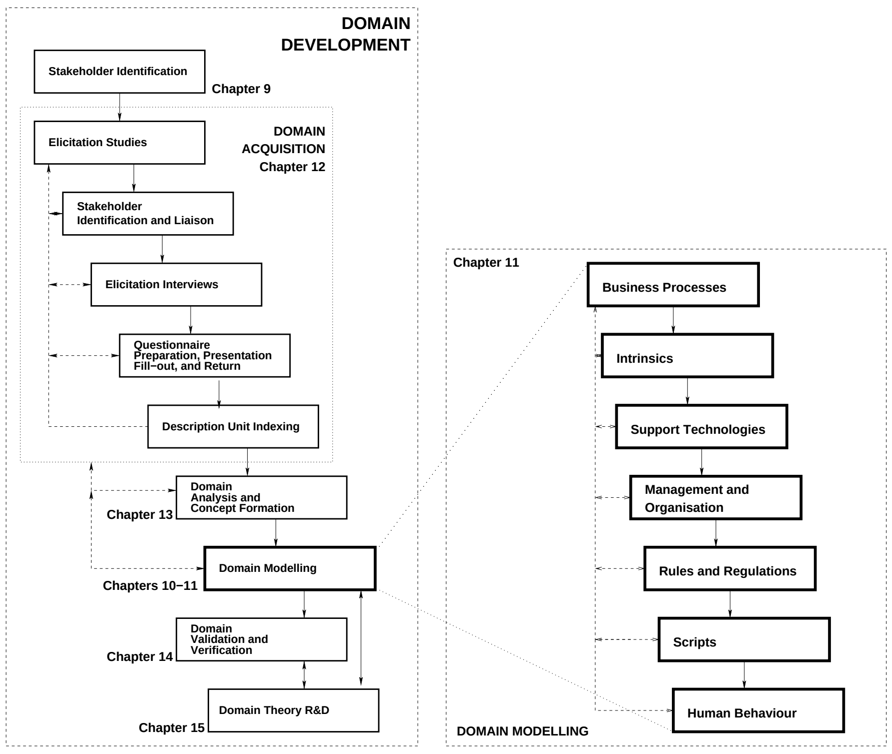

DOMAIN DEVELOPMENT
Stakeholder Identification
Chapter 9
Elicitation Studies
DOMAIN ACQUISITION Chapter 12
Stakeholder Identification and Liaison
Elicitation Interviews
Questionnaire Preparation, Presentation Fill−out, and Return
Description Unit Indexing
Domain Analysis and Concept Formation
Chapter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

Fig. 5.4. The domain engineering process model diagram

Figure 5.4 diagrams, in box-and-edge form, the stages and steps of domain engineering development and their interrelations. The diagram does not give a correct "picture" of the necessity for iteration: going "backwards and forwards" across the development, i.e., across the diagram. Obviously, having a precise understanding of the syntax, semantics and pragmatics of boxes and edges, helps developers and their managers monitor and control (including "contain") iterations.

5.2.3 The Requirements Engineering Process Model

We first rough-sketch narrate the stages and steps of the requirements engineering development of a requirements model, then review the documents that should emanate from such development. Finally we diagram an essence of the narration and the document table-of-contents.

But first some words on "the machine" and on requirements models.

The Machine

Requirements is about prescribing the machine: the hardware and the software which shall implement the requirements. The machine resides in the domain. Once developed we shall sometimes refer to that domain as the environment of the machine — with the machine + that environment becoming a new domain.

Requirements Models

A main result of requirements engineering development, as applied to some specific application domain[4], is a requirements model. Domain models are in the form of descriptions. Requirements prescriptions prescribe what there should be.

Requirements Engineering, A Narrative

The requirements engineering triptych dogma, and as argued in Chaps. 18–26 of [33], advocates (item 2.) the following stages of prescription development (after work on information documents [items 1.a–l] have been duly completed): (2.a) identification of as wide a spectrum of requirements stakeholders, (2.b) acquisition of requirements statements, (2.c) rough-sketching first ideas of a requirements model in order to ("eureka") discover un-formulated requirements, (2.d) establishment (and subsequent, throughout all stages, use and maintenance) of a requirements terminology (ontological terms), and (2.e) requirements modelling of all requirements facets: (2.e.i) business process reengineering (BPR),

(2.e.ii) domain requirements, (2.e.iii) interface requirements, (2.e.iv) machine requirements, and (2.e.v) completion of a full requirements prescription. Intertwined with the requirements prescription parts (item 2., subitems (a–e)) are the analysis parts with (3.a) requirements analysis aiming at identifying inconsistencies, conflicts and incompletenesses, (3.b) requirements validation,

[4]Examples of domains are: (1) the financial service industry as a whole, (1.1) a bank, (1.1.1) a bank's mortgage lending business; (2) the transportation industry as a whole, (2.1) a railway system, (2.1.1) an interlocking system; etcetera.

(3.c) requirements verification, and (3.d) possible work on establishing a requirements theory.

The new things here are the way in which (2.b) ‘acquisition of requirements statements’ is pursued, and items (2.c) and (2.c subitems i., ii., and iii.). Essentially (2.b) questionnaires are formulated on the basis of assumed existing domain specifications.

Essentially the questionnaires and the rough sketching of a domain and interface requirements model, after analysis of the requirements statements (3.a), is pursued basically as follows (2.e.ii): which of the entities, functions, events and behaviours described in the domain model must be partially or fully supported by the machine being requirements prescribed? Must those (entities, functions, events and behaviours) being so selected (i.e., projected) be made more determinate, and/or more concretely instantiated, and/or extended, and/or fitted with, or to other, elsewhere developed requirements? Similar for business processes of the “original” domain. Usually they need be reconsidered (2.e.i). Etcetera.

Requirements Engineering Documents

1. Information
   - (a) Name, Place and Date
   - (b) Partners
   - (c) Current Situation
   - (d) Needs and Ideas (Eurekas, I)
   - (e) Concepts & Facilities (Eurekas, II)
   - (f) Scope & Span
   - (g) Assumptions & 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
   - (m) Plans
     - i. Project Graph
     - ii. Budget
     - iii. Funding
     - iv. Accounts
   - (n) Management
     - i. Assessement
     - ii. Improvement
       - A. Plans
       - B. Actions

Fig. 5.5. Requirements engineering informative document table-of-contents

Figures 5.5, 5.6 on the next page and 5.7 on page 116 summarise the plenitude of highly interrelated sets of documents that must all be carefully worked out and carefully correlated.

Requirements Engineering Stages and Steps

Figure 5.8 on page 116 and 5.9 on page 117 diagram, in box-and-edge form, the stages and steps of requirements engineering development and their inter-

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. Business Process Re-engineering
         - Sanctity of the Intrinsics
         - Support Technology
         - Management and Organisation
         - Rules and Regulation
         - 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
            ⋆ Accessability
            ⋆ Availability
            ⋆ Reliability
            ⋆ Robustness
            ⋆ Safety
            ⋆ Security
         - Maintenance
            ⋆ Adaptive
            ⋆ Corrective
            ⋆ Perfective
            ⋆ Preventive
         - Platform
            ⋆ Development Platform
            ⋆ Demonstration Platform
            ⋆ Execution Platform
            ⋆ Maintenance Platform
         - Documentation Requirements
         - Other Requirements
      v. Full Reqs. Facets Doc.

Fig. 5.6. Requirements engineering prescription document table-of-contents

relations. The diagram does not give a correct "picture" of the necessity for iteration: going "backwards and forwards" across the development, i.e., across the diagram. Obviously, having a precise understanding of the syntax, semantics and pragmatics of boxes and edges, helps developers and their managers monitor and control (including "contain") iterations.

5.2.4 The Software Design Process Model

We first rough-sketch narrate the stages and steps of software design development of a software architecture (etc.), then review the documents that should

3. Analyses
   (a) Requirements Analysis and Concept Formation
      i. Inconsistencies
      ii. Conflicts
      iii. Incompletenesses
      iv. Resolutions
   (b) Requirements Validation
      i. Stakeholder Walk-through and Reports
      ii. Resolutions
   (c) Requirements Verification
      i. Model Checkings
      ii. Theorem Proofs
      iii. Test Cases and Tests
   (d) Requirements Theory
   (e) Satisfaction and Feasibility Studies
      i. Satisfaction: Correctness, unambiguity, completeness, consistency, stability, verifiability, modifiability, traceability
      ii. Feasibility: Technical, economic, BPR

Fig. 5.7. Requirements engineering analytic document table-of-contents

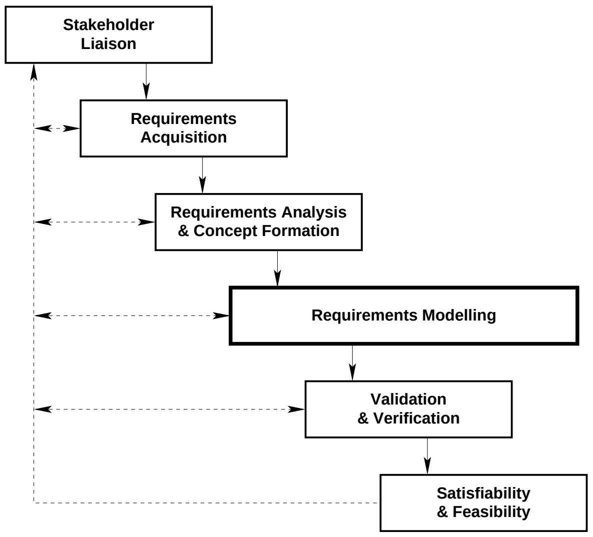

Stakeholder Liaison
Requirements Acquisition
Requirements Analysis & Concept Formation
Requirements Modelling
Validation & Verification
Satisfiability & Feasibility

Fig. 5.8. Diagramming a requirements process model

emanate from such development. Finally we diagram an essence of the narration and the document table-of-contents.

Software Design, A Narrative

The software design process is here simplified into four stages (Fig. 5.10 on page 118 items 2.a–d): software architecture design, component design, module design, and (module) program coding. Each of these may consist of two or more steps of development (cf. Fig. 5.11 on page 119). Between adjacent

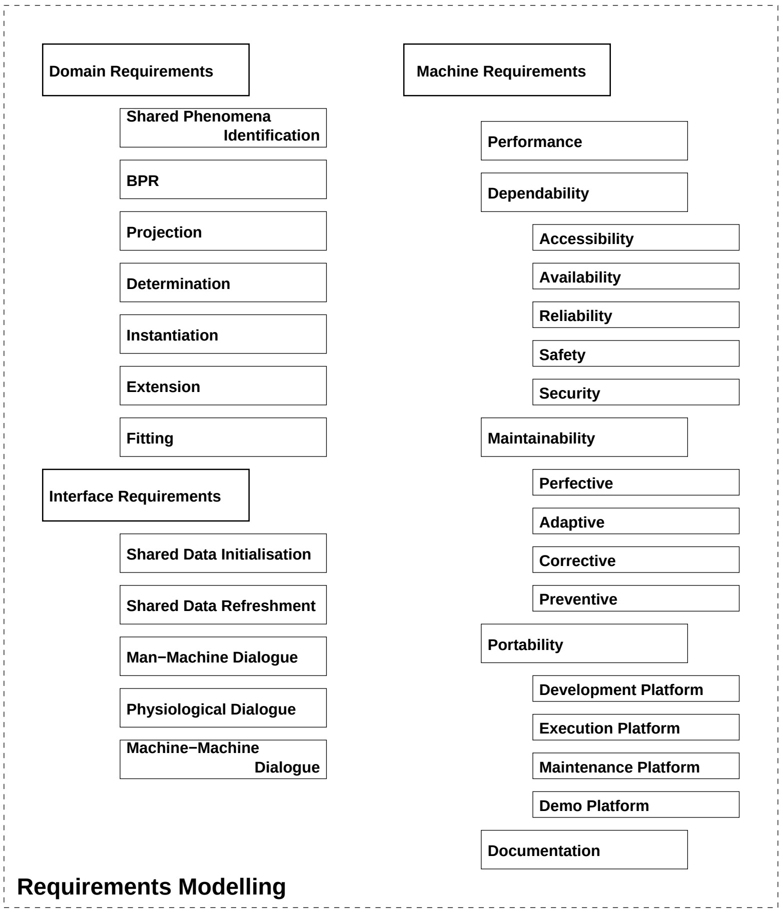


Fig. 5.9. The requirements modelling stage

steps there is a correctness obligation (V:MC:T, verification, model checking and testing). Verification proofs usually are of the kind: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ which means that the proof that the $\mathcal{S}$oftware implements the $\mathcal{R}$equirements entails reference to the $\mathcal{D}$.

Software Design Documents

Figure 5.10 on the next page summarises the plenitude of highly interrelated sets of documents that must all be carefully worked out and carefully correlated.

1. Information
   (a) Name, Place and Date
   (b) Partners
   (c) Current Situation
   (d) Needs and Ideas
   (e) Concepts and Facilities and Facilities
   (f) Scope and Span
   (g) Assumptions and Dependencies
   (h) Implicit/Derivative Goals
   (i) Synopsis
   (j) Standards Compliance
   (k) Contracts
   (l) The Teams
      i. Management,
      ii. Developers,
      iii. Consultants
   (m) Plans
      i. Project Graph
      ii. Budget, Funding, Accounts
   (n) Management
      i. Assessement Plans & Actions
      ii. Improvement Plans & Actions
2. Software Specifications
   (a) Architecture Design ($S_{a_1} \ldots S_{a_n}$)
   (b) Component Design ($S_{c_{1_i}} \ldots S_{c_{n_j}}$)
   (c) Module Design ($S_{m_1} \ldots S_{m_m}$)
   (d) Program Coding ($S_{k_1}, \ldots, S_{k_n}$)
3. Analyses
   (a) Analysis Objectives and Strategies
   (b) Verification (S_{i_p}, $S_i \sqsupseteq_{L_i} S_{i+1}$)
      i. Theorems and Lemmas L_i
      ii. Proof Scripts $\wp_i$
      iii. Proofs Π_i
   (c) Model Checking ($S_i \sqsupseteq P_{i-1}$)
      i. Model Checkers
      ii. Propositions P_i
      iii. Model Checks $\mathcal{M}_i$
   (d) Testing ($S_i \sqsupseteq T_i$)
      i. Manual Testing
         • Manual Tests $M_{S_1} \ldots M_{S_\mu}$
      ii. Computerised Testing
         A. Unit (or Module) Tests C_u
         B. Component Tests C_c
         C. Integration Tests C_i
         D. System Tests $C_s \ldots C_{s_{i_{t_s}}}$
   (e) Evaluation of Adequacy of Analysis

Legend:

S	Specification
L	Theorem or Lemma
$\wp_i$	Proof Scripts
Π_i	Proof Listings
P	Proposition
$\mathcal{M}$	Model Check (run, report)
T	Test Formulation
M	Manual Check Report
C	Computerised Check (run, report)
$\sqsupseteq$	"is correct with respect to (wrt.)"
$\sqsupseteq_\ell$	"is correct, modulo ℓ, wrt."

Fig. 5.10. Software design document table-of-contents

Software Design Stages and Steps

Figure 5.11 on the facing page diagram, in box-and-edge form, the stages and steps of software design development and their interrelations. The diagram does not give a correct "picture" of the necessity for iteration: going "backwards and forwards" across the development, i.e., across the diagram. Obviously, having a precise understanding of the syntax, semantics and pragmatics of boxes and edges, helps developers and their managers monitor and control (including "contain") iterations.

5.3 Review of the Triptych Process

5.3.1 The Process Model: Diagrams and Tables-of-content

We have surveyed the (mainly) software development processes according to the triptych dogma. We have seen that these processes can be diagrammed and also be "mapped" onto tables-of-content of the documents resulting from respective phases. Of course there is much more to these three phases, their very many stages (within phases), and their potentially very many more steps (within stages) than can be covered in chapter form.

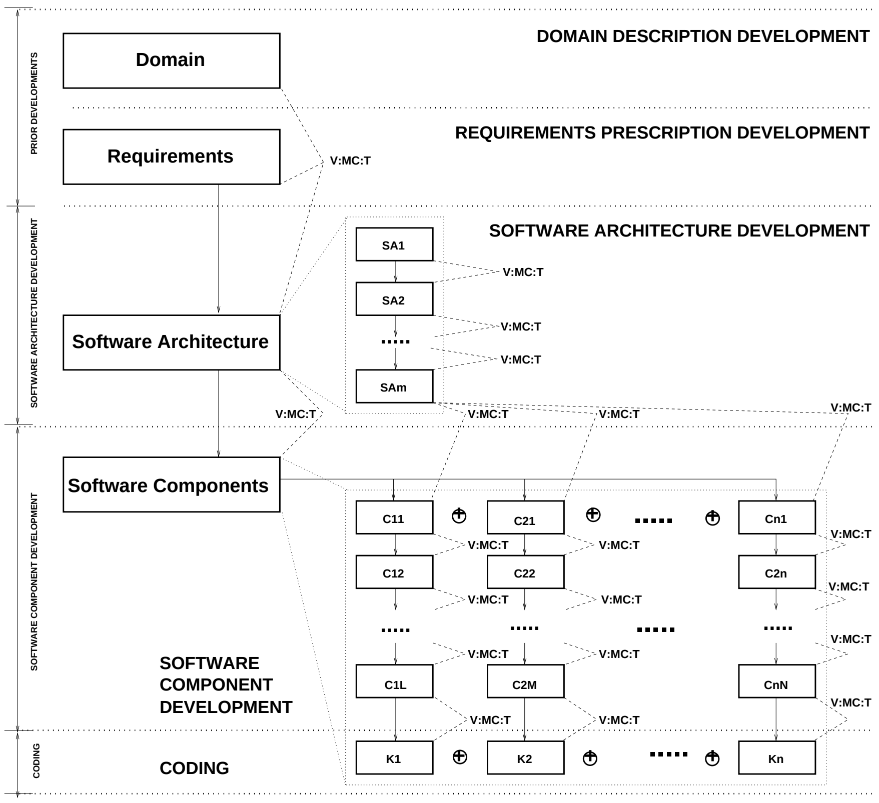

PRIOR DEVELOPMENTS
SOFTWARE ARCHITECTURE DEVELOPMENT
SOFTWARE COMPONENT DEVELOPMENT
CODING
Domain
DOMAIN DESCRIPTION DEVELOPMENT
Requirements
REQUIREMENTS PRESCRIPTION DEVELOPMENT
V:MC:T
SOFTWARE ARCHITECTURE DEVELOPMENT
Software Architecture
SA1
SA2
SAm
Software Components
C11
C12
C1L
C21
C22
C2M
Cn1
C2n
CnN
SOFTWARE COMPONENT DEVELOPMENT
CODING
K1
K2
Kn

Fig. 5.11. The software design development processes

5.3.2 Process Model Semantics

Diagrams, such as those of Figs. 5.1, 5.4, 5.8–5.9 and 5.11, reflect some pragmatics, has some syntax and embodies, hopefully some semantics. We wish, here, to emphasise the semantics:

> *What is important to mention here, justifying this separate section, is that each of the boxes of the description, prescription and software design parts of Figs. 5.4, 5.8, 5.9 and 5.11 and each of their interconnecting edges embody a clear set of method principles, techniques and tools with many of these techniques also being pursuable formally and supported, or supportable, by theory-based tools.*

In the following we shall assume that the above *paragraph* on the semantics of the process model diagrams is taken for granted.

5.3.3 Informal versus Formal Development

The term ‘development’ covers any combination of the three phases: domain, requirements or software design only; domain+requirements or requirements+software design, or all three phases “more-or-less” consecutively.

Development can, as shown in [33] be pursued **informally** or **formally**, and therefore in any “graded scale” combination of these.

0. Informal development means: no formalisation of domain descriptions, requirements prescriptions or software design specifications are attempted. Thus verification cannot be done using formal proofs or model checking. Only code testing.

There are, roughly speaking three “points” on the semi-formal to formal scale of development.

1. Systematic development formalises domain descriptions, requirements prescriptions and software design specifications. But that is just about as much formalisation that is attempted.

2. Rigorous development extends systematic development by stating all “crucial”[5] properties and maybe even sketch or carry through the proof or model checking of properties of some of these.

3. Formal development requires that all necessary (including correctness) properties are formally expressed and theorem proved or model checked.

The triptych paradigm allows for any of these latter three (1.–2.–3.) forms of development.

5.3.4 Adherence to Phases, Stages and Steps

It is important to stress the following assumption:

> *Adhering to the triptych paradigm, to us, means that all phases, stages and steps as outlined above are followed. This means that documents are produced as per the tables-of-contents shown in Fig. 5.3, Figs. 5.5–5.7 and Fig. 5.10.*

Our treatment, next, of process assessment and improvement, is based on, i.e., starts with the above assumption.

5.4 Process Assessment and Improvement Management

5.4.1 Notions of ‘Process Assessment’ and ‘Improvement’

In order to speak of ‘assessment’ and ‘improvement’ we must identify that which is being assessed and improved: the results of following one set of

[5]We do not here further characterise what we mean by ‘crucial’.

method principles, techniques, tools and their management, over following another such set. Process assessment is now about judging adherence of a given process to its process model, $\mathcal{P}$ragmatically, $\mathcal{S}$emantically and $\mathcal{S}$yntactically ($\mathcal{PSS}$, usually in reverse order): to which ($\mathcal{PSS}$) degrees does the process fulfill what is "laid down" in the process model. Process improvement is then about changing the assessed development processes such that the results of using the changed processes are assessed to have been improved.

By "assessment" and "improvement" we first of all mean "assessing and improving documents". The documents are those emanating from activities denoted by nodes and edges of the process model.

Each such box and each such edge may have many documents "attached" to it, and each such document has its syntax, semantics and pragmatics. The syntax and semantics can usually be given very precise definitions. Hence we can, in a sense, objectively "measure" (assess) whether a document "lives up" to that syntax and that semantics! For pragmatics the "measure" is more subjective. To be able to "measure" process improvement one must therefore attach to each planned document for each box and each edge a "measure" of compliance. Is a document in 100% compliance with those syntactic, semantics and pragmatic measures or is it not? Or more precisely: where on a scale from 0 to 1 lies the quality of a document wrt. an "ideal".

> SOFTWARE PROCESS ASSESSMENT 1 **Process Model Syntax and Semantics:** *In order to handle process improvement (à la CMM, from a lower to a higher level) — using the triptych approach — managers (as well as, of course, developers), must be intimately familiar with the syntax and semantics of the documents produced and expected to be produced by process model node and edge activities. This is a strong requirement and can not be expected by just any software development organisation. And there are really no shortcuts.*[6] *Process improvement — wrt. the precision of monitoring resource usage — is predicated on this assumption: that management is strongly based on professional awareness of triptych principles, techniques and tools. The "degree"*[7] *to which a development document adheres to the syntax and semantics of the relevant box or edge thus provides an assessment.*

Several groups, worldwide, the most well known is perhaps Praxis High Integrity Systems, http://www.praxis-his.com, practices this on a daily basis. So do many members of ForTIA: The Formal Techniques Industrial Association, www.fortia.org.

[6]In other branches of engineering project managers (i.e., project leaders) and developers, the "engineers at floor level" basically all have the same, normalising education. Hence they are intimately familiar with the syntax and semantics of their tasks. The problem is in software engineering.

[7]This "degree" notion is not defined here

Software Process Improvement 1 **Process Model Syntax and Semantics:** *To improve this general aspect of the possible processes that developers and managers might be able to pursue under the banner of the Triptych Process Model one simply has to resort to education and training. There is no substitute.*

We choose here to **also** "anchor" our discourse of 'process improvement' by referring to the *Capability Maturity Model* (CMM) of Watts S. Humphrey (WSH) [139]. CMM postulates five levels of maturity of development groups. Level 1 being a lowest, in a sense "least desirable", and level 5 being the highest, "most desirable" level of professionalism that WSH finds useful to define. Process improvement, by a development group, is now the improvement of the development processes such that the group (i.e., the software house) advances from level i to level $i + j$ where i, j are positive numbers and $i + j$ is less than 6. So let us first review WSH's notion of CMM.

5.4.2 The CMM: Capability Maturity Model

The following subsection are "lifted" from http://en.wikipedia.org/wiki/Capability_Maturity_Model:

1. **Level 1, Initial**: At maturity level 1, processes are usually ad hoc and the organization usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes. In spite of this ad hoc, chaotic environment, maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.

   Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes again.
2. **Level 2, Repeatable**: At maturity level 2, software development successes are repeatable. The organization may use some basic project management to track cost and schedule.

   Process discipline helps ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

   Project status and the delivery of services are visible to management at defined points (for example, at major milestones and at the completion of major tasks).

   Basic project management processes are established to track cost, schedule, and functionality. The minimum process discipline is in place to repeat earlier successes on projects with similar applications and scope. There is still a significant risk of exceeding cost and time estimate.

3. **Level 3, Defined**: The organization's set of standard processes, which is the basis for level 3, is established and improved over time. These standard processes are used to establish consistency across the organization. Projects establish their defined processes by the organization's set of standard processes according to tailoring guidelines.

   The organization's management establishes process objectives based on the organization's set of standard processes and ensures that these objectives are appropriately addressed.

   A critical distinction between level 2 and level 3 is the scope of standards, process descriptions, and procedures. At level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit.
4. **Level 4, Managed**: Using precise measurements, management can effectively control the software development effort. In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications.

   Subprocesses are selected that significantly contribute to overall process performance. These selected subprocesses are controlled using statistical and other quantitative techniques.

   A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.
5. **Level 5, Optimizing**: Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements. Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.

   Process improvements to address common causes of process variation and measurably improve the organization's processes are identified, evaluated, and deployed.

   Optimizing processes that are nimble, adaptable and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization's ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning.

A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process (that is, shifting the mean of the process performance) to improve process performance (while maintaining statistical probability) to achieve the established quantitative process-improvement objectives.

5.4.3 Process Models and Processes

One thing is the process model, viz., the graph-like structures shown in, for example, Figs. 5.4, 5.8, 5.9, and 5.11. (These are syntactic structures, but have semantic meanings.) Another thing is the actual usage of such models, that is, the actual processes that the software developers (domain, requirements and software design engineers) "steer through" when developing domain models, requirements models and software designs.

Graphs and Graph Traversal Traces

Assume some graph-like, let us call it, process model, see Fig. 5.12.

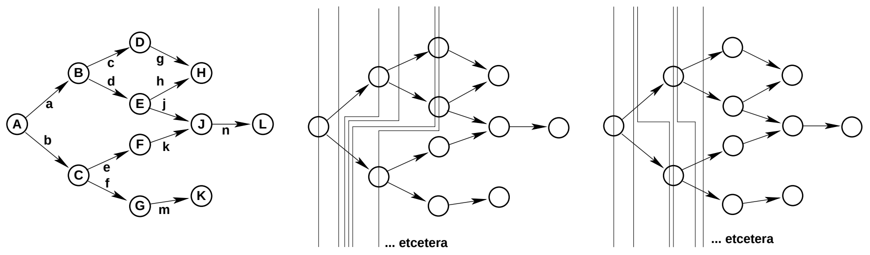


Fig. 5.12. A graph (left) and two (incomplete) traversal traces (center and right)

- So Fig. 5.12 shows a process model and two traces.
  - ⋆ REDOs, that is, iterations of phases, stages and steps lead to additional traces.
    - ◇ Let us call the totality (set) of these traces for OK traces.
  - ⋆ And "jumping" or just "skipping" phases, stages and steps lead to further additional traces.
    - ◇ Let us call these "jumped" or "skipped" traces for NOK traces.
- A process model thus denotes a possibly infinite set of such traces.

The leftmost part of Fig. 5.12 on the preceding page shows an acyclic graph. The graph consists of distinctly labeled nodes and (therefrom distinctly labeled) edges. The center and right side of the figure shows some possible traversal traces. By a traversal trace we understand a sequence of wavefronts.

By a wavefront we understand a set of node and edge labels such that no two of these are on the same path from an input (i.e., in-degree 0) to an output (i.e., out-degree 0) node, and such that there is a contribution to the set from any path from an input to an output node.

The third wave of the two traces shown in the two rightmost figures can thus be represented by $\{B, b\}$ and $\{a, C\}$.

A process model is here taken to be a graph: boxes denote activities that result in information and description, prescription or specification documents and edges denote analytic activities that result in documents that record results of (concept formation, consistency, conflict and completeness) analysis, verification, model checking, testing and possibly theory formation.

A development process is any trace over sets of these activities.

Figure 5.12 on the facing page's center figure thus portrays the following initial trace:

$$\langle\{\mathrm{A}\},\{\mathrm{a,b}\},\{\mathrm{B,b}\},\{\mathrm{c,d,b}\},\{\mathrm{D,E,b}\},\{\mathrm{D,E,C}\},...,\mathrm{etcetera}\rangle$$

Thus a process model denotes a set of such traces.

Incomplete and Extraneous Processes

The trace:

$$\langle\{\mathrm{A}\},\{\mathrm{a,b}\},\{\mathrm{c,d,b}\},\{\mathrm{D,E,b}\},\{\mathrm{D,E,C}\},...,\mathrm{etcetera}\rangle$$

appears to have skipped the activity (phase, stage or step) designated by B. Loosely speaking we call such processes incomplete with respect to their underlying (i.e., assumed) process model (Fig. 5.12 on the preceding page, the leftmost graph).

The trace:

$$\langle\{\mathrm{A}\},\{\mathrm{a,z}\},\{\mathrm{X}\},\{\mathrm{D,Y,b}\},\{\mathrm{D,E,C}\},...,\mathrm{etcetera}\rangle$$

appears to have performed some activities (z, X, Y) not designated by the process model of Fig. 5.12 on the facing page (the leftmost graph). Loosely speaking we call such processes extraneous (or ad hoc) with respect to their underlying process model.

Process Iterations

The trace

$$\langle\{\mathrm{A}\},\{\mathrm{a,b}\},\{\mathrm{B,b}\},\{\mathrm{a,b}\},\{\mathrm{B,b}\},\{\mathrm{c,d,b}\},\{\mathrm{B,b}\},\{\mathrm{c,d,b}\},\{\mathrm{D,E,b}\},\{\mathrm{D,E,C}\},...\rangle$$

designates an iterated process. After action B in {B,b} the process "goes-back" to perform action b (in {a,b}); and after (either of) actions c or d in {c,d,b} the process "goes-back" to perform action B in {B,b}. Loosely speaking we call such processes iterated with respect to their underlying process model.

The above trace only shows simple "one-step" (or stage or phase) "backward-and-then-onward" iterations. But the REDO idea, also indicated in Fig. 5.1 on page 109, can be extended to any number of steps (etc.).

Degrees of Process Model Compliance

We can now define two notions of process model compliance, a syntactic and a semantic. The syntactic notion of process model compliance has to do with "the degree" to which an actual process matches a possibly iterated, i.e., an OK trace of a process model. The semantic notion of process model compliance is concerned with adherence to the semantics of boxes and edges.

We shall not, in this paper define these notions precisely — that should be done in a future paper.

Suffice it to summarise that an ongoing process, i.e., an ongoing software development project can be assessed wrt. its syntactic and its semantics compliance wrt. its process model. One can precisely state which activities have been omitted (incompleteness), and which activities were extraneous (or ad hoc).

We first deal with syntactic compliance, then, in the next section, with semantics compliance.

Software Process Assessment 2 **Syntactic Process Compliance:** *Given the generic process models diagrammed in Figs. 5.4, 5.8, 5.9 and 5.11, and given the project-specific software development graph as exemplified by Fig. 5.13, one can now, in a process claimed to adhere to these models and graphs quite simply assess whether that actual process follows those diagrams.*

We assume that assessment takes place "regularly", that is, with a frequency higher than process wave transitions, that is, more often than the process evolves through steps and stages. Otherwise it may be too late (or too cumbersome) to "catch-and-do" an omitted step.

Software Process Improvement 2 **Syntactic Process Compliance:** *Adherence to the process model can, at least "formally", be improved by actually ensuring that the process steps and stages (or even phases) that were assessed to not having been performed, that these be performed.*

A "Base 0" for Triptych Developments

By a triptych development we mean a development which applies the principles, techniques and tools as prescribed by the triptych dogma. Either in a systematic, or in a rigorous, or in a formal way. A triptych development process therefore, "by definition" has its base point at level 4 in the CMM scale. This does not mean that a software development process which claims to follow the triptych dogma (or the software house within which that process occurs) at least measures at level 4. The dogma sets standards. The process may follow, or may not follow such standards. Whether they are followed or not is now an "easy" matter to resolve. The degree to which the dogma, in all its very many instantiations, is followed is now "fairly easy" to resolve. The "ease" (or "easiness") depends on how well developers and management understands the many triptych principles, techniques and tools, how well they understand the prescribed syntax and semantics of required documents, and on how well they understand their pragmatics, that is, the reason for these principles, techniques and tools.

The pragmatics is what makes management interesting. Well mastered pragmatics allows the managers leeway (i.e., discretion) in the dispatch of their duties, that is, allow them to skip (or "go light" on) certain activities, including choosing whether a step or even a stage should be performed "lightly" or more-or-less "severely", that is, be informal, or formal (and then in a scale from systematic via rigorous to formal).

SOFTWARE PROCESS ASSESSMENT 3 **Planned Syntactic and Semantics Compliance:** *If a process is assessed (SPA) to be in full compliance, syntactically and semantically with the process model then we claim that the software development in this case is at CMM level 4 (or higher).*

SOFTWARE PROCESS IMPROVEMENT 3 **Planned Syntactic and Semantics Compliance:** *If it is assessed that a process has not reached CMM level 4, and that at least CMM level 4 is desired, then one must first secure syntactic compliance, see process improvement #2 (Page 126), thereafter ensure that each of the steps (or stages, or phases) whose semantic compliance was assessed too low be redone and according to their semantic intents.*

5.4.4 Proactive Measures

The above spoke in general about assessment and improvement.

We are now ready to deal with more specific issues of process assessment and improvement. But first we need to refine our notion of process model.

Project Development Graphs

The process models (i.e., the graphs) are generic. They apply to any development — whatever the software. They must be instantiated to fit the particular problem frame (see [147] as well as [33, Chap. 28]).

Figure 5.13 shows the project development graph that was used in the development of the Danish Ada compiler [58, 77] (1981–1984).

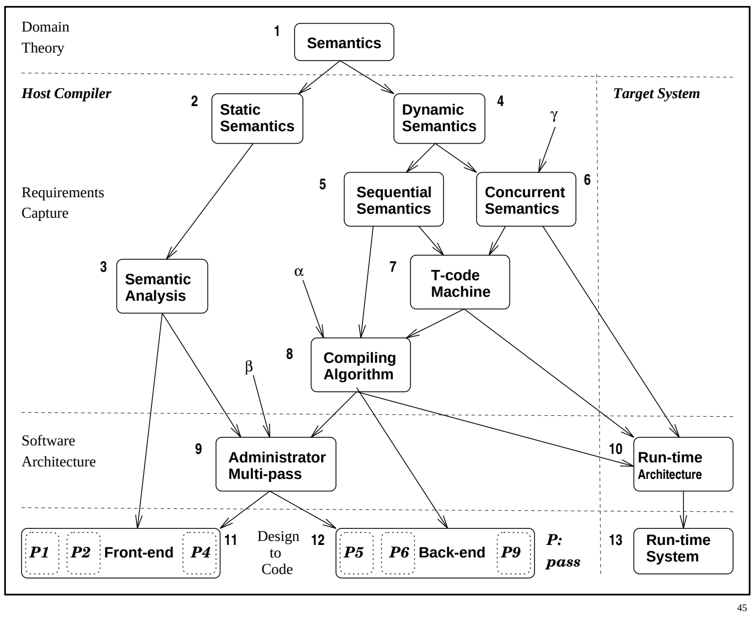


Fig. 5.13. Project development graph: Compiler development

The top horizontal and dashed line of Fig. 5.13 separates domain engineering from requirements engineering. The domain engineering box ("Semantics") represents a simplification of the usual domain engineering process diagram. (You are to put that usual diagram into the "Semantics" box (a form of supplementation)!) The second horizontal and dashed line of Fig. 5.13 separates requirements engineering from software design. (Again you are to supplement the requirements engineering and software design boxes etc. of Fig. 5.13 with the generic process models for requirements engineering and software design.)

The software (domain, requirements, software design) development graphs in the sense of supplementation are orthogonal to process models. They allow more meaningful assignment of semantics to boxes and edges and they allow more specific management (planning, monitoring and control).

In this paper we do not show how to construct a resulting pull graph from the combination of the earlier process models with the later, domain specific graph.

Management

So far, in this paper, we have not dealt with management. Management[8] is about planning, and monitoring and controlling process resource usage — including the quality of the documents emanating from the use of resources. Planning is about scheduling and rescheduling processes and allocating and re- and deallocating resources to (from) processes.

A primary resource in software development is the set of domain and requirements engineers and the set of software designers. Other primary resources are the time, space and tools used by these developers.

Planning — Scheduling and Allocation:

Planning starts with instantiating, selecting, or developing a new, tentative, software development graph and detailing (i.e., annotating) it wrt. process model concepts: phases (domain, requirements, software design), stages (stakeholder identification, acquisition, analysis, description (prescription, specification), verification, model checking, testing, validation, etc.), and make allowances for more crucial, detailed steps.

Based on the resulting software development graph management can, in a far more detailed (i.e., granular) way, ascribe resource usage (people, time, offices, equipment, software development tools) to each box and edge, and can schedule these in time and allocate them "in space".

> SOFTWARE PROCESS ASSESSMENT 4 **Resource Planning:** *How can one assess a software development project plan (i.e., graph), that is, something which designates something yet to happen? Well, one can compare to previous software development graphs purporting to cover "similar" (if not identical) development problems and their eventual outcome, that is, the process that resulted from following those graphs. Based on actual resource usage accounts one can now — "to the best of anyone's ability" — draw a software development graph and ascribe resource consumption estimates (time, people, equipment) to each and every node and edge. Thus 'assessment' here was "speculated assessment" of an upcoming project.*

Thus, if that 'speculated assessment' of an upcoming project is felt, by the assessors, i.e., the management, to be flawed, to be questionable, then one has to proceed to improvement:

> SOFTWARE PROCESS IMPROVEMENT 4 **Resource Planning:** *One must first improve the precision with which one designs the domain specific project development graphs. Then the precision with which we associate resource usage with each box and edge of such a graph. Etcetera.*

[8]We restrict management to the below items. That is: we do not consider product management (which products to develop and in which sequence of deliverables) nor project funding.

> *Some development projects are very much "repeats" of earlier such projects and one can expect improvement in project development graphs for each "repeat". Other projects are very much tentative, explorative, that is, are actually applied research projects — for which one only knows of a project development graph at the end of the project, and then that graph is not necessarily a "best such"!*

Monitoring & Controlling Resource Usage:

As the project (i.e., the process) evolves management can now check a number of things: adherence to schedule and allocation, and adherence to the syntactic and the semantic notions of process model compliance.

Most process models do not possess other than rather superficial and then mostly syntactic notions of compliance. In the triptych process model semantic compliance is at the very core: Every box and every edge of the process models have precise syntax and semantics of the documents that are the expected results of these (box and edge) activities.

> Software Process Assessment 5 **Resource Usage:** *No problems here. As each step (of the development process) unfolds one can assess its compliance to estimated plan.*

Should a resource usage assessment reveal that there are problems (for example: all resources used well before completion of step) then something must be done:

> Software Process Improvement 5 **Resource Usage:** *Well, perhaps not this time around, when all planned resources have already been consumed — no improvement can undo that — but perhaps "next" time around. An audit may reveal what the cause of the over-consumption was. Either a naïve, too low resource estimate, or unqualified staff, or some simple or not so simple mistakes? Improvement now means: make precautions to avoid a repetition.*

Resource usage is at a very detailed and accountable level and can thus be better assessed. Slips (usually excess usage) can be better foreseen and discovered and more clearly defined remedies, should milestones be missed or usage exceeded, can then be prescribed — including skipping stages and steps whose omission are deemed acceptable.

Skipping stages and steps result in complete, perhaps extraneous (ad hoc) processes. Given that management has an "ideal" process model and hence an understanding of desirable, possibly iterated processes, management can now better assess which are acceptable slips.

From Informal to Formal Development

By process improvement, to repeat and to enlarge on our previous characterisation of what is meant by process improvement, we understand something which improves the quality of resulting software. We "translate" the term 'resulting software' into the term 'resulting documents'. These documents can — as defined on in Sect. 5.3.3 — be developed either informally (without any use of any formalism other than the final programming language[9]), or systematically formal, or rigorously formal or formally formal!

Informal Development:

It is an indispensable property of the triptych approach to software development that the formalisable steps domain engineering, requirements engineering and software design be pursued in some systematic via rigorous to formal manner. Hence the informal aspects of development is restricted to the development of only the informative documents. Informative documents are usually "developed" by project leaders and managers. Hence an "upper" level of management is process assessing and possibly prescribing process improvements to a "lower" level of management!

> SOFTWARE PROCESS ASSESSMENT 6 **Informal Development of Informative Documents:** *We refer to Fig. 5.2 on page 109. That figure lists the kind of documents to be carefully developed — and hence assessed. Since no prescribed syntax, let alone formal semantics can be given for these documents — whose purpose is mainly pragmatic — assessment is a matter of style. It is easy to write non-sensical, "pat" informative documents which do not convey any essence, any insight. Assessment hence has to evaluate: dose a particular, of the many informative documents listed in Fig. 5.2 on page 109, really convey, in succinct form, an essence of the project being initiated?*

> SOFTWARE PROCESS IMPROVEMENT 6 **Informal Development of Informative Documents:** *If an informative document is assessed to not convey its intended message succinctly, with necessary pedagogical and didactical "bravour", then it must be improved. Only "seasoned", i.e., experienced managers can do this.*

Systematic, Rigorous and Formal Development:

The development of domain description, requirements prescription and software design documents as well as the development of analytic documents (tests, verification, model checking and validation) can be done in a spectrum from systematically via rigorously to formally.

[9]Thus we do not consider UML to be a formalism. For a "formalism" to qualify as being properly formal it must have a precise syntax, the syntax must have a precise semantics, and there must be a congruent proof system, that is, a set of proof rules such that the semantics satisfy the proof rules.

Software Process Assessment 7 **Staff and Tool Qualification:** *Given the syntax and semantics of the specific step — in the process model — of the tasks to be assessed a (syntax and semantics) a knowledgeable person, a project (task) leader or a manager, can assess compliance. That assessment is greatly assisted by the software tools*[10] *that support activities of those tasks: If they can process the documents then something seems OK. If not, assessment will have to be negative.*

There are now two distinct, "extreme" reasons for a failure to meet assessment criteria — with any actual reason possibly being a combination of these two "extremes". One is that the quality of the staff performing the affected tasks is not up to expectations. The other is that the tools being deployed are not capable of supporting the problem solution task.

Staff Qualification:

If the assessment of 'Systematic, Rigorous and Formal Development of Specifications and Their Analysis' is judged negative due to inadequate development decisions then we suggest the following kind of improvement.

Software Process Improvement 7 **Staff Qualification:** *It is suggested that improvement, when deemed necessary, takes either of three forms: Either "move" from a systematic to a rigorous level of development, or from a rigorous to a formal level of development when that is possible and redo the task(s) affected. Or educate and train staff to reperform the affected task(s) more accurately (while remaining systematic, rigorous, or formal as the case may be. Or replace affected staff with better educated and trained staff and redo the task(s) affected. These kinds of improvement decisions are serious ones.*

Tools

There are different categories of tools. Tools can serve management: for the design of software development graphs (a la Fig. 5.13 on page 128) and their "fusion" into the appropriate process model diagrams (a la Fig. 5.4, Fig. 5.8 and 5.9, and Fig. 5.11) and for the monitoring and control (i.e., assessment and improvement) of the process with respect to these diagrams. And tools can serve developers: syntactic and semantic description, prescription and software design tools as well as analytic tools: for testing, model checking and verification (proof assistance or theorem provers). These tools embody, that is, represent the formalisms of the textual or diagrammatic notations used — whether `Alloy` [146], `B` [1,71], `CafeOBJ` [89,90,99,100], `Casl` [11, 78,184,185], `Duration Calculus` [247,248], `LSCs` [80,128,153], `MSCs` [142–

[10]These software tools mainly support the use of the main tools, namely the specification languages, their transformation (or refinement) and their proof systems.

144], `Petri Nets` [148, 199, 210–212], `RAISE RSL` [31–33, 44, 101, 104, 106], `Statecharts` [123, 124, 126, 127, 129], `TLA+` [155, 156, 175, 176], `VDM-SL` [55, 56,95,96], `or Z` [132,133,229,230,242]. Thus the formal notations of the above listed thirteen languages, whether textual or diagrammatic, or combinations thereof, are tools, as are the software packages that support uses of these linguistic and analytic means.

Tool Qualification:

If assessment of 'Systematic, Rigorous and Formal Development of Specifications and Their Analysis' is judged negative due to inadequate tools then we suggest the following kind of improvement:

> Software Process Improvement 8 **Tool Qualification:** *Better tools must be selected and applied to the task(s) affected (i.e., judged negatively assessed). These tools are either intellectual, that is, the specification languages, whether textual or diagrammatic, and their refinement and proof systems, or they are the manifest software tools that support the intellectual tools. These are likewise a serious improvement decisions.*

5.4.5 Review of Process Assessment and Improvement Issues

We have surveyed, somewhat cursorily, a number of software process assessment and software process improvement issues. We characterise these from a another viewpoint below.

1. **Process Model Syntax and Semantics Assessment and Improvement:**
   We refer to Page 121.
   The issue here is whether the management and development staff really understands and, to a satisfactory degree, can handle the triptych process model in all its myriad of phases, stages and steps, specificationally and analytically, and with all its myriad of documentation demands. If not, then they cannot be effectively assessed and subjected to "standard" improvement measures.
   This is an assessment (and improvement) issue which precedes proper project start.
2. **Syntactic Process Compliance Assessment and Improvement:**
   We refer to Page 126.
   This issue is a "going concern", that is, an ongoing, effort of regular assessment and possibly an occasional improvement. It merely concerns whether a mandated step (or stage or even phase) of development and its expected production of related documents has taken or is taking place.
3. **Planned Syntactic and Semantics Compliance Assessment and Improvement:**

This is an assessment (and improvement) issue which, in a sense, sets a proper framework for the project: Does management wish to attain at least CMM level 4, or higher or lower? In that sense it precedes project start while determining the rigour with which the next assessments and improvements are to be pursued.

4. **Resource Planning Assessment and Improvement:**
   We refer to Page 129.
   This item of assessment and improvement takes place at project start and may have to be repeated when resource consumption exceeds plans. Assessment and improvement may involve "layers" of project leaders and management.
5. **Resource Usage Assessment and Improvement:**
   We refer to Page 130.
   This item of assessment and improvement takes place at regular intervals during an entire project and involves "layers" of project leaders and management. It may lead to replanning, see Item 4.
6. **Informative Document Assessment and Improvement:**
   We refer to Page 131.
   Informative documents are usually directed at client and software house management and not at software house software engineers. As such they are often the result of the combined labour of client and software house management. Assessments take place while the planned project is being discussed between these partners. Improvements may then be suggested at such mutual project planning meetings.
7. **Staff and Tool Qualification Assessment**
   We refer to Page 132.
   This form of assessment is probably the most crucial aspect of SPA (and hence of SPI). It strikes at the core of software development. The resources spent in what is being assessed conventionally represents a very large, a dominating percentage of resource expenditures.
   Thus this complex of "myriads" of process step, stage and phase (document) assessment must be subject to utmost care.
7. **Staff Qualification Improvement:**
   We refer to Page 132.
   The implications of even minor staff improvement actions may be serious: staff well-being, inavailability of staff, serious delays are just a few. Thus improvement planning must be subject to utmost care, both technically and socio-economically, but also as concerns human relations.
8. **Tool Qualification Improvement:**
   We refer to Page 133.
   The implications of even minor tool improvement actions may be serious: serious retraining or restaffing, serious time delays, and serious hence cost overruns.

5.4.6 Hindrances to Process Assessment and Improvement

What could be “standard” hindrances to assessment and improvement? And what could be similar hindrances to actually carrying out projects according to the triptych process model?

Lack of Knowledge of Methodology

Both management and development staff must be intimately familiar with the triptych process model and its syntactic, semantic and pragmatic implications, its need for from systematic via rigorous to formal development, its need for the creation, use, maintenance and correlation of myriads of documents, and its need for assessment and possible improvement. Lack of knowledge of the methodology, ever so sporadically, is a hindrance to proper software development processes.

Generation Gaps

Classically we see that young candidates join software houses as software engineers, fluent in the kind of methods: principles, techniques and tools inherent in the triptych approach. They are eager to use these. But they are usually stifled: their slightly older colleagues as well as their project leaders and managers do not possess the same skills, or are outright illiterate wrt. the triptych methods: principles, techniques and tools. Lack of knowledge of the methodology, across generations of staff, is a hindrance to proper software development processes — and even a few years (say ten) count as a generation today.

Lack of Tools

Above we pointed out that there we intellectual tools and there were software tools that support the use of the intellectual tools. Here we mean both.

On one hand, the problem being tackled in a particular software development project may be such that there are, as of today, year 2006, no obvious or no good intellectual tools (and a methodological approach, i.e., a process model) for the properly assessable and improvable pursuit of such a project. On the other hand, even when appropriate intellectual tools are (and a process model is) available there may not be good manifest, that is, software support tools available.

Lack of tools is a serious hindrance to proper software development processes.

Lack of Acceptance

By far the most common hindrance to proper software development processes — such as suggested by the triptych process model — processes that can be properly assessed and for which a continuum of improvement possibilities exists — is (1) the lack of acceptance of what is referred to as “formal methods”, and (2) the lack of acceptance of the necessity to do proper domain modelling before tackling requirements.

This is not the time and place to lament on those “facts”.

5.5 Conclusion

It is time to conclude.

5.5.1 Summary

We have overviewed a rather comprehensive process model, the triptych model which prescribes three development phases: domain engineering, requirements engineering and software design, and which, within these prescribes a number of stages and within these again a number of steps. Phases, stages and steps may be iterated, and phases, stages and steps, as well as the transition between them results in documents. We have modelled process models as acyclic graphs which denote possibly infinite sets of indefinite length traces of waves, where a wave is a set of nodes and edges of the graph not on the same path from an input node (of in-degree 0) to an output node (of out-degree 0), but where subsequences of traces may be repeated (due to process iterations: redoing “previous” tasks).

We have then identified a class of seven software process assessment categories and eight software process improvement categories, all in relation to the syntax and semantics of the triptych process model. Finally we briefly touched upon hindrances to process assessment and improvement.

5.5.2 Future

This is the first time the author has related the triptych model of [31–33] to SPA and SPI: software process assessment and software process improvement, and hence to CMM, Watts Humphrey’s Capability Maturity Model. It has been instructive to do so. Clearly, for actual projects to apply the triptych approach and to carry out the assessments and improvements suggested in this paper, more clarifying directions must be given. And support tools developed.

5.5.3 Software Procurement

Software

By software we shall here mean not just the executable code and some manuals on how to install, use and possibly repair this code, but also all the documents that emanates from a full project developing this code. That is, all the documents listed in Fig. 5.3, Figs. 5.5, 5.6 and 5.7, and in Fig. 5.10.

Procurement

In software procurement it is therefore natural that the procurement includes as large a set of the documents mentioned in those figures, and that all these documents have passed an assessment with some positive, CMM level-relatable degree of acceptance.

Domains and Problem Frames[1]

The Triptych Dogma and M.A.Jackson's PF Paradigm

Abstract

In this report we interpret Michael Jackson's Problem Frame concept [147]. We do so in the context of the transition from a domain model of some broad application domain to a set of requirements models — one for each of a sufficiently distinct set of domain requirements — but for what is claimed to be "the same" broad application domain.

We shall thus follow the triptych dogma of [33] — and this Monograph![2]

First we develop a domain model (for the application area of transportation nets). Then we sketch the development of a number of diverse domain requirements for the computerisation of transportation network management, monitoring and control. Finally we relate the diverse domain requirements to similarly different Problem Frames.

The claim of this report is that to better understand the underlying issues of Michael Jackson's Problem Frame one must see the concept of Problem Frames as a function of the relation between a domain model and a (domain) requirements model.

6.1 Domains and Problem Frames

Before software can be designed we must understand its requirements. Before requirements can be prescribed we must understand the domain[3]. In this paper we exemplify **one** domain description and **four** related requirements prescriptions. The latter are intended to illustrate distinct frames.

[1]This is an edited version of [27]. Invited talk at IWAAPF (International Workshop on Advances and Applications of Problem Frames), a satellite event of ICSE 2006 (International Conference on Software Engineering) Shanghai, May 2006.

[2] [33, 147] (together with references to the companion volumes [31, 32] of [33]) will be the only citations of this report.

[3]The term domain is here used instead of the — in problem frame contexts — perhaps more common term environment.

6.1.1 Aims & Objectives

Aims

We aim to illustrate aspects of problem frame independent domain engineering, problem frame dependent requirements engineering, and the interplay between various requirements prescriptions.

Objectives

Our objective is to plead for more systematic software engineering work around domain engineering, before requirements engineering sets in.

6.1.2 Structure of Paper

We first bring a long and undoubtedly boring domain description, then four requirements prescriptions. In the conclusion we relate this quadruple development to the problem frame approach, and briefly discuss a rôle for the triptych cum problem frame approach in the Verified Software: Theories, Techniques and Experiments (VSTTE) and the Ubiquitous Computing grand challenges!

We need the "multiple masses of details" in order to properly substantiate our aims and objectives.

6.2 The Domain

Our domain is that of transportation nets. We abstract in such a way as to capture both road, rail, air and shipping transport nets. The basic concepts of street segments between street intersections, rail lines between train stations, air lanes between airports and shipping lanes between harbours are abstracted into segments and the street intersections, train stations, airports and harbours are abstracted into junctions.

6.2.1 Net Topology

We "slowly" (read: carefully) narrate and formalise a number of concepts related to segments and junctions.

6.2.2 Nets, Segments and Junctions

Nets consists of one or more segments and two or more junctions.

```
type
   N, S, J
value
   obs_Ss: N → S-set
   obs_Js: N → J-set
axiom
   ∀ n:N • card obs_Ss(n) ≥ 1 ∧ card obs_Js(n) ≥ 2
```

Annotations:

- N, S, J are considered abstract types, i.e., sorts. N, S and J are type names, i.e., names of types of values. Values of type N are nets, values of type S are segments and values of type J are junctions.
- One can observe from nets, n, their (one or more) segments (obs_Ss(n)) and their (two or more) junctions (obs_Js(n)); n is a value of type N.
- Functions have names, obs_Ss, and obs_Js, and functions, f, have signatures, f: A → B (not illustrated), where A and B are type names. A designates the definition set of f and B the range set.
- A-**set** is a type expression. It denotes the type whose values are finite, possibly empty set of A values.
- These observer functions are postulated.
- They cannot be formally defined.
- They are "defined" once a net has been pointed out[4]
- The axiom expresses that in any net there is at lest one segment and at least two junctions. ■

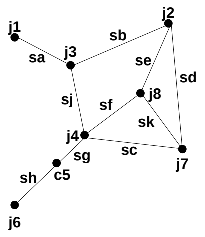


Fig. 6.1. A simple net of segments and junctions

Applying the observer functions to the net of Fig. 6.1 yields:

[4]Take the transportation net Europe. By inspecting it, and by deciding which segments and which associated junctions to focus on (i.e., "the interesting ones") we know which are all the interesting roads, rail tracks, air lanes and shipping lanes, respectively the interesting (associated) street intersections, trains stations, airports and harbours.

```
    obs_Ss(n) = {sa,sb,sc,sd,se,sf,sg,sh,sj,sk}
    obs_Js(n) = {j1,j2,j3,j4,j5,j6,j7,j8}
```

Nets, segments and junctions are physically manifest, i.e., are phenomena.

6.2.3 Segment and Junction Identifications

Segments and junctions have unique identifications.

```
type
    Si, Ji
value
    obs_Si: S → Si
    obs_Ji: J → Ji
```

Segment and junction identifications are abstract concepts. No two segments have the same segment identifier. And no two junctions have the same junction identifier.

```
axiom
    ∀ n:N • card obs_Ss(n) ≡ card {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
    ∀ n:N • card obs_Js(n) ≡ card {obs_Ji(c)|j:J • j ∈ obs_Js(n)}
```

Annotations:

- **card** set expresses the cardinality of the set set, i.e., its number of distinct elements.
- {f(a)|a:A • p(a)} expresses the set of all those B elements f(a) where a is of type A and has property p(a) [where we do not further state f, A and B. p is a predicate, i.e., a function, here from A into truth values of type **Bool**, for Boolean].
- The axioms now express that the number of segments in n is the same as the number of segment identifiers of n — which is a circumscription for: No two segments have the same segment identifier.
- Similar for junctions. ∎

The constraints that limit identification of segments and junctions can be physically motivated: Think of the geographic (x, y, z co-ordinate) point spaces "occupied" by a segment or by a junction. They must necessarily be distinct for otherwise physically distinct segments and junctions. Segments may thus cross each other without the crossing point (in x, y space) being a junction, but, for example, one segment may, at the crossing point be physically above the other segment (tunnels, bridges, etc.). Segments are delimited by two distinct junctions. From a segment one can also observe, obs_Jis, the identifications of the delimiting junctions.

```
type
    Jip = {|{ji,ji′}:Ji-set • ji≠ji′|}
```

```
value
    obs_Jis: S → Jip
```

Annotations:

- {|a:A • p(a)|} is a subtype expression. It expresses a subset of type A, namely those A values which enjoys property p(a) [p is a predicate, i.e., a function, here from A into truth values in the type **Bool**]. In the above p(a) is ji≠ji′.
- In this case Jip is the subtype of Ji-**set** whose values are exactly 2 element sets of Ji elements. ∎

Any junction has a finite, but non-zero number of segments connected to it. From a junction one can also observe, obs_Sis, the identifications of the connected segments.

```
type
    Si1 = {|sis:Si-set•card sis ≥1|}
value
    obs_Sis: J → Si1
```

Annotations:

- Si1 is the type whose values are non-empty, but finite sets of Si values. ∎

One cannot from a segment alone observe the connected junctions. One can only refer to them. Similarly: one cannot from a junction alone observe the connected segments. One can only refer to them. The identifications serve the role of being referents. In any net, if s is a segment connected to connectors identified by ji and ji′, respectively, then there must exist connectors j and j′ which have these identifications and such that the identification si of s is observable from both j and j′.

```
axiom
    ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒
        let {ji,ji′} = obs_Jis(s) in
        ∃! j,j′:J • {j,j′}⊆obs_Js(n) ∧ j≠j′ ∧
            obs_Si(s) ∈ obs_Sis(c) ∩ obs_Sis(c′) end
```

Annotations:

- We read the above axiom:
  - ⋆ for all nets n and for all segments s in n
  - ⋆ let ji and ji′be the two distinct junction identifications observable from s, then
  - ⋆ exists exactly two distinct junctions, j and j′ of the net, such that
  - ⋆ the segment identification of s is in both the sets of segment identifications observable from j and j′.

∎

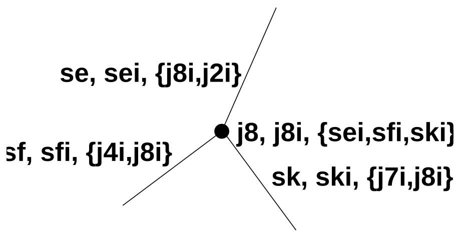


Fig. 6.2. One junction and its connected segments

Figure 6.2 illustrates the relation between observed identifications of segments and junctions.

The above constraints take on the mantle of being laws of nets: If segments and junctions otherwise have distinct identifications, then the above must follow as a law of man-made artifacts. Vice-versa: In any net, if j is a junction connecting segments identified by si, si′, . . . , si″ then there must exist segments s, s′, . . . , s″ which have these identifications and such that the identification ji of j is observable from all s, s′, . . . , s″.

```
axiom
    ∀ n:N, j:J • j ∈ obs_Js(n) ⇒
        let sis = obs_Sis(c), ji = obs_Ji(j) in
        ∃! ss:S-set • ss⊆obs_Ss(n) ∧ card ss=card sis ∧
        sis = {|obs_Si(s)|s:S•s ∈ ss|} end
```

Annotations:

- Let us read the above axiom:
  - ⋆ for all nets, n, and all junctions, j, of that net
  - ⋆ let sis be the set of segment identifications observed from j, and let ji be the junction identifier of j, then
  - ⋆ there exists a unique set, ss, of segments of n with as many segments as there are segment identifications in sis, and such that
  - ⋆ sis is exactly the set of segment identifications of segments in ss.

∎

6.2.4 Paths and Routes

By a path we shall understand a triplet of a junction identification, a segment identification and a junction identification.

```
type
    P = Ji × Si × Ji
value
    paths: N → P-set
    paths(n) ≡
    {(ji,si,ji′)|s:S,ji,ji′:Ji,si:Si•
                    s ∈ obs_Ss(n)∧{ji,ji′} ∈ obs_Jis(s)∧si=obs_Si(s)}
```

Annotations:

- Paths are modelled as Cartesians.
- One can generate all the paths of a net.
- It is the set of path triplets, two for each segment of the net and such that the pair of junction identifications, ji and ji′, observable from a segment is at either "end" of the triplet, and such that the segment identification is common to the two triplets (and in the "middle"). ∎

Paths, and as we shall see next, routes are mental concepts. By a route of a net we shall understand a list, i.e., a sequence of paths as follows:

- A sequence of just one path of the net is a route.
- If r and r′ are routes of the net such that the last junction identification, ji, of the last path, (_,_,ji) of r and the first junction identification, ji′, of the first path (ji′,_,_) of r′ are the same, i.e., ji=ji′, then $r\frown r'$ is a route.
- Only routes that can be generated by uses of the first (the basis) and the second (the induction) clause above qualify as proper routes of a net.

```
type
  R = {|r:P*•wf_R(r)|}
value
  wf_R: P* → Bool
  wf_R(r) ≡
    ∀ i:Nat • {i,i+1}⊆inds(r) ⇒
      let (_,_,ji)=r(i), (ji′,_,_)=r(i+1) in ji = ji′ end

  routes: N → R-infset
  routes(n) ≡
    let rs = {⟨p⟩|p:P•p ∈ paths(n)}
          ∪ {r⁀r′|r,r′:R•{r,r′}⊆rs∧wf_R(r⁀r′)} in
    rs end
```

Annotations:

- Routes are well-formed sequences of paths.
- A sequence of paths is a well-formed route if adjacent path elements of the route share junction identification.
- Give a net we can compute all its routes as follows:
  - ⋆ let rs be the set of routes to be computed. It consists first of all the single path routes of the net.
  - ⋆ Then rs also contains the concatenation of all pairs of routes, r and r′, such that these are members of rs and such that their concatenation is a well-formed route.

⋆ If the net is circular then the set rs is an infinite set of routes. The least fix point of the recursive equation in rs is the solution to the "routes" computation.

∎

6.2.5 Segment and Junction Identifications of Routes

For future purposes we need be able to identify various segment and junction identifications as well as various segments and junctions of a route.

```
value
   xtr_Jis: R → Ci-set, xtr_Sis: R → Si-set
   xtr_Jis(r) ≡ case r of ⟨⟩ → {}, ⟨(ji,_,ji′)⟩⌢r′ → {ji,ji′}∪ xtr_Jis(r′) end
   xtr_Sis(r) ≡ case r of ⟨⟩ → {}, ⟨(_,si,_)⟩⌢r′ → {si}∪ xtr_Sis(r′) end

   xtr_Ss: N × Ji → S-set
   xtr_Ss(n,ji) ≡ {s|s:S•s ∈ obs_Ss(n) ∧ ji ∈ obs_Jis(s)}

   xtr_C: N × Ji → C, xtr_S: N × Si → S
   xtr_C(n,ji) ≡ let j:J • j ∈ obs_Js(n) ∧ ji=obs_Ji(j) in j end
   xtr_S(n,si) ≡ let s:S • s ∈ obs_Ss(n) ∧ si=obs_Si(s) in s end

   first_Ji: R ~→ Ji, last_Ji: R ~→ Ji
   first_Ji(r) ≡ case r of ⟨⟩ → chaos, ⟨(ji,_,_)⟩⌢r′ → ji end
   last_Ji(r) ≡ case r of ⟨⟩ → chaos, r′⌢⟨(_,_,ji)⟩ → ji end

   first_Si: R ~→ Si, last_Si: R ~→ Si
   first_Si(r) ≡ case r of ⟨⟩ → chaos, ⟨(_,si,_)⟩⌢r′ → si end
   last_Si(r) ≡ case r of ⟨⟩ → chaos, r′⌢⟨(_,si,_)⟩ → si end

   first_J: R × N ~→ J, last_J: R × N ~→ J
   first_J(r,n) ≡ xtr_J(first_Ji(r),n)
   last_J(r,n) ≡ xtr_J(last_Ji(r),n)

   first_S: R × N ~→ S, last_S: R × N ~→ S
   first_S(r,n) ≡ xtr_S(first_Si(r),n)
   last_S(r,n) ≡ xtr_S(last_Si(r),n)
```

Annotations:

- Given a route one can extract the set of all its junction identifications.
  - ⋆ If the route is empty, then the set is empty.
  - ⋆ If the route is not empty than it consists of at least one path and the set of junction identifications is the pair of junction identifications of the path together with set of junction identifications of the remaining route.

    - ⋆ Possible double "counting up" of route adjacent junction identifications "collapse", in the resulting set into one junction identification. (Similarly for cyclic routes.)
- Given a route one can similarly extract the set of all its segment identifications.
- Given a net and a junction identification one can extract all the segments connected to the identified junction.
- Given a net and a junction identification one can extract the identified junction.
- Given a net and a segment identification one can extract the identified segment.
- Given a route one can extract the first junction identification of the route.
    - ⋆ This extraction should not be applied to empty routes.
    - ⋆ A non-empty route can always be thought of as its first path and the remaining route. The first junction identification of the route is the first junction identification of that (first) path.
- Given a route one can similarly extract the last junction identification of the route.
- Given a route one can similarly extract the first segment identification of the route.
- Given a route one can similarly extract the last segment identification of the route.
- And similarly for extracting the first and last junctions, respectively first and last segments of a route. ■

6.2.6 Circular and Pendular Routes

A route is circular if the same junction identification either occurs more than twice in the route, including if it occurs as both the first and the last junction identification of the route. Given a net we can compute the set of all non-circular routes by omitting from the above pairs of routes, r and r′, where the two paths share more than one junction identification.

```
non_circular_routes: N → R-set
non_circular_routes(n) ≡
   let rs = {⟨p⟩|p:P•p ∈ paths(n)}
            ∪ {r⌢r′|r,r′:R•{r,r′}⊆rs∧wf_R(r⌢r′)∧non_circular(r,r′)} in
   rs end
non_circular: R×R → Bool
non_circular(r,r′) ≡ card xtr_Jis(r) ∩ xtr_Jis(r′) =1
```

Annotations:

- To express the finite set of all non-circular routes
    - ⋆ is to re-express the set of all routes
    - ⋆ except constrained by the further predicate: non_circular.

- An otherwise well-formed route consisting of a first part r and a remaining part r′
    - ⋆ is non-circular if the two parts share at most one junction identification.

■

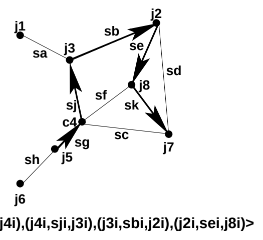


Fig. 6.3. A route, graphically and as an expression

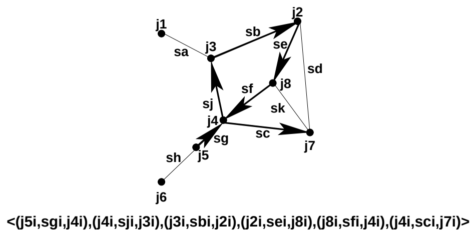

j1
sa
j3
sb
j2
se
sd
j8
sf
sk
sj
j4
sc
sg
j7
sh
j5
j6
<(j5i,sgi,j4i),(j4i,sji,j3i),(j3i,sbi,j2i),(j2i,sei,j8i),(j8i,sfi,j4i),(j4i,sci,j7i)>

Fig. 6.4. A circular route, graphically and as an expression

Let a path be (ji_f, si, ji_t), then (ji_t, si, ji_f) is a *reverse path*. That is: the two junction identifications of a path are reversed in the reverse path. A route, rr, is the reverse route of a route r if the ith path of rr is the reverse path of the $n - i + 1$'st path of r where n is the length of the route r, i.e., its number of paths. A route is a *pendular* route if it is of an even length and the second half (which is a route) is the reverse of the first half route.

```
value
   reverse: P → P
   reverse(jif,si,jit) ≡ (jit,si,jif)

   reverse: R → R
   reverse(r) ≡
      case r of
         ⟨⟩ → ⟨⟩,
         ⟨(jif,si,jit)⟩⌢r′ → reverse(r′)⌢⟨(jit,si,jif)⟩
```

```
    **end**

  reverse(r) ≡ ⟨reverse(r(i))|i **in** [n..1]⟩

  pendular: R → R
  pendular(r) ≡ r⌢reverse(r)

  is_pendular(r) ≡ ∃ r′,r″:R • r′⌢r″ = r ∧ r″=reverse(r′)
```

Annotations:

- The reverse of a path is a path with the same segment identification, but with reverse junction identifications.
- The reverse of a route, r, is
  - ⋆ the empty route if r is empty, and otherwise
  - ⋆ it is the reverse route of all of r except the first path of r concatenate (juxtaposed) with the singleton route of the reverse path of the first path of r.
- Given a route, r, we can construct a pendular route whose first half is the route r and whose last half is the reverse route of r.
- A (an even length) route is a pendular route if it can be expressed as the concatenation of two (equal length) routes, r′ and r″ such that r″ is the reverse of r′, that is, if its second half is the reverse of its first half. ∎

6.2.7 Connected Nets

A net is connected if for any two junctions of the net there is a route between them.

```
value
  is_connected: N → Bool
  is_connected(n) ≡
    ∀ j,j′:J • {j,j′}⊆obs_Js(n) ∧ j≠j′ ⇒
      let (ji,ji′) = (obs_Ji(j),obs_Ji(j′)) in
      ∃ r:R • r ∈ routes(n) ∧
        first_Ji(r) = ji ∧ last_Ji(r) = ji′ end
```

Annotations:

- A net n is connected if
  - ⋆ for all two distinct connectors of the net
  - ⋆ where ji and ji′ are their junction identifications,
  - ⋆ there exists a route, r, of the net,
  - ⋆ whose first junction identification is ji and whose last junction identification is ji′.

∎

6.2.8 Net Decomposition

One can decompose a net into all its connected subnets. If a net exhaustively consists of m disconnected nets, then for any pair of nets in different disconnected nets it is the case that they share no junctions and no segments. The set of disconnected nets is the smallest such set that together makes up all the segments and all the junctions of the ("original") net.

```
value
  decompose: N → N-set
  decompose(n) as ns
    obs_Ss(n) = ∪{obs_Ss(n′)|n′:N•n′ ∈ ns} ∧
    obs_Js(n) = ∪{obs_Js(n′)|n′:N•n′ ∈ ns} ∧
    {} = ∩{obs_Ss(n′)|n′:N•n′ ∈ ns} ∧
    {} = ∩{obs_Js(n′)|n′:N•n′ ∈ ns} ∧
    ∀ n′:N•n′ ∈ ns ⇒ connected(n′) ∧ ...
```

Annotations:

- A set ns of nets constitutes a decomposition of a net, n,
  1. if all the segments of n appear in some net of ns,
  2. if all the junctions of n appear in some net of ns,
  3. if no two or more distinct nets of ns share segments,
  4. if no two or more distinct nets of ns share junctions, and
  5. if all nets of ns are connected.
- **Comment:** It appears that items 3 and 4 are unnecessary, that is, are properties once items 1, 2 and 5 hold. ■

That is, we have the following:

```
Lemma:
  ∀ n:N •
     let ns = decompose (n) in
     ∀ n′,n″:N • {n′,n″}⊆ns ∧ n′≠n″ ⇒
        obs_Ss(n′) ∩ obs_Ss(n″) = {} ∧
        obs_Js(n′) ∩ obs_Js(n″) = {} end
```

The above items define a lot of what there is to know about transportation nets if we only operate with the sorts that have been introduced (N, S, Si, J, Ji) and the observer functions that have likewise been introduced (obs_Ss, obs_Js, obs_Si, obs_Ji, obs_Jis and obs_Sis). The relationships between sorts, i.e., net, segment, segment identification, junction and junction identification values are expressed by the axioms. The above is a so-called property-oriented model of the topology of transportation nets. That model is abstract in that it does not hint at a mathematical model or at a data structure representation of nets, segments and junctions, let alone their topology. By topology we shall here mean how segments and junctions are "wired up". The axioms above guarantee that no segment of a net is left "dangling": It is always connected to two distinct junctions; and no junctions of a net is left isolated: It is always connected to some segments of the net.

We have tacitly assumed that all segments are two way segments, that is, transport can take place i either direction. Hence a segment gives rise to two paths.

6.2.9 Multi-Modal Nets

Interesting transportation nets are multi-modal. That is, consists of segments of different transport modalities: roads, rails, air-lanes, shipping lanes, and, within these of different categories. Thus roads can be either freeways, motorways, ordinary highways, and so on.

General Issues

We introduce a concept, M, of transport modes. M is a small set of distinct, but otherwise further undefined tokens. An m in M designates a transport modality.

```
type
   M
```

Segment and Junction Modes

With each segment, s, we can associate a single mode, m, and with each junction we can associate the set of modes of its connected segments.

```
value
   obs_M: S → M
   obs_Ms: J → M-set
axiom
   ∀ n:N, j:J • j ∈ obs_Js(n) ⇒
      let ss = xtr_Ss(n,obs_Ji(j)) in
      obs_Ms(j) = {obs_M(s)|s:S • s ∈ ss} end
   ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒
      let {ji,ji′} = obs_Jis(s) in
      let {j,j′} = {xtr_J(n,ji),xtr_J(n,ji′)} in
      obs_M(s) ∈ obs_Ms(j) ∩ obs_Ms(j′) end end
```

Annotations:

- From a segment one can observe its mode.
- From a junction one can observe its set of modes.
- Let us read the first axiom:
  - ⋆ for all net, n, and all junctions, j, of that net

  - ⋆ let ss be the set of segments connected to j,
  - ⋆ now the set of modes of c is equal to the set of modes of the segments in ss.
- Let us read the second axiom:
  - ⋆ for all net, n, and all segments, s, of that net
  - ⋆ let ji and ji′ be the junction identifiers of the two junctions to which s is connected, and
  - ⋆ let j and j′ be the two corresponding junctions,
  - ⋆ then the segment mode is in both the set of modes of the two junctions.
- We can define a function, xtr_Ss, which from a net, n, and a junction identification, ji, extracts the set of segments, ss, connected to the junction identified by ji.
- xtr_Ss(n,ji) yields the set of segments, ss, in the net n for which ji is one of the observed junction identifications of s.
- And we can define a function, xtr_J, of signature N × Ji → J, which when applied to a net, n, and a junction identification, ji,
- extracts the junction in the net which has that junction identifier. ∎

Single-Modal Nets and Net Projection

Given a multi-modal net one can project it onto a set of single modality nets, namely one for each modality registered in the multi-modal net.

```
type
    mmN = {|n:N • card xtr_Ms(n) > 1|}
    smN = {|n:N • card xtr_Ms(n) = 1|}
value
    xtr_Ms: N → M-set
    xtr_Ms(n) ≡ {obs_M(s) | s:S • s ∈ obs_Ss(n)}

    projs: N → smN-set
    projs(n) ≡ {proj(n,m) | m:M • m ∈ xtr_Ms(n)}
    proj: N × M → smN
    proj(n,m) as n′
        post
            let ss = obs_Ss(n), ss′ = obs_Ss(n′),
                js = obs_Js(n), js′ = obs_Js(n′) in
            ss′ = {s | s:S • s ∈ ss ∧ m=obs_M(s)} ∧
            js′ = {j | j:J • j ∈ js ∧ m ∈ obs_Ms(j)}
            end
```

Annotations:

- A multi-modal net is a net with more than one mode. mmN is thus the subtype of nets, n:N, which are multi-modal.

- A single-modal net is a net with exactly one mode. smN is thus the subtype of nets, n:N, which are multi-modal.
- The xtr_Ms function extracts the mode of every segment of a net.
- The projs function applies to any net, n:N, and yields the set of single-modal subnets of n, one for each mode of n. The projs function makes use of the proj function.
- The proj function applies to any n, n:N, and any mode of that net, and yields the single-modal subnet on n whose mode is the given mode.
  - ⋆ The proj function is expressed by a post condition, i.e., a predicate that characterises the necessary and sufficient relation between the argument net, n, and the result net n′.
  - ⋆ In a single-modal net, n′, projected from a multi-modal net, n, and of mode m, we keep exactly those segments, ss′, of n whose mode is m,
  - ⋆ and we keep exactly those junctions, js′, of n whose mode contains m.
  - ⋆ No more is needed in order to express the necessary and sufficient condition for a single-modal net to be a subnet of a proper net.
  - ⋆ That is, some single-modal nets are not proper nets since in proper nets every junction have the set of modes of all the segments connected to the junction.

■

6.2.10 Sub-Junctions

Let ms:$\{m_1, m_2, ..., m_n\}$ be the set of modes of a junction j. To each such mode m_i we can associate a junction mj_{m_i}.[5] With any such junction mj_{m_i} we can associate a modal junction identification $mj_{m_{i_i}}$.

```
type
  MJ, MJi
value
  obs_MJs: J → MJ-set
  obs_MJ: J × M → MJ
  obs_MJi: MJ → MJi
  obs_M: MJ → M
axiom
  ∀ j:J,m:M•m ∈ obs_Ms(j) ⇒ let mj = obs_MJ(j,m) in obs_M(mj)=m end
```

6.2.11 Segment and Junction Attributes

Segment and Junction Attribute Observations

We now enrich our segments and junctions.

[5] mj_{m_i} is not to be confused with the junction identification of j_i of j.

Segments have lengths. Junctions have modality-determined lengths between pairs of (same such modality) segments connected to the junction. Segments have standard transportation times, i.e., time durations that it takes to transport any number of units of freight from one end of the segment to the other. Junctions have standard transfer time per modality of transport between pairs of segments connected to the junction. Junctions have standard arrival time per modality of transport. Junctions have standard departure times per modality of transport. Segments have standard costs of transporting a unit of freight from one end of the segment to the other end. Junctions have standard costs of transporting a unit of freight from the end of one connecting segment to the beginning of another connecting segment.
We can now assess (i) length of a route, (ii) shortest routes between two junctions, (iii) duration time of standard transport along a route, including transfer, stopover and possible reloading times at junctions, and iv) shortest duration time route of standard transport between two junctions.

```
type
   L, TI
value
   ms:M-set,                     axiom ms≠{}
   obs_L: S → L
   obs_L: Si × J × M × Si → L
   obs_TI: S → TI
   obs_TI: Si × J × Si → TI
   obs_TI: J × M $\xrightarrow{\sim}$ TI,          pre obs_TI(j,m): m ∈ obs_Ms(j)
   obs_TI: J × M × M $\xrightarrow{\sim}$ TI,  pre obs_TI(j,m,m′): {m,m′}⊆obs_Ms(j)
   obs_arr_TI: J × M $\xrightarrow{\sim}$ TI,     pre obs_arr_TI(j,m): m ∈ obs_Ms(j)
   obs_dep_TI: J × M $\xrightarrow{\sim}$ TI,   pre obs_dep_TI(j,m): m ∈ obs_Ms(j)
   +: L × L → L
   +: TI × TI → TI
```

Annotations:

- L and Ti are sorts designating length and time values.
- ms denotes a non-empty set of modes.
- From a segment one can observe, obs_L, its length.
- From a segment one can observe, obs_TI, a time duration for a normal conveyour of the mode of the segment to travel the length of the segment.
- From a junction and a mode (of that junction) one can observe, obs_TI, a time duration for a normal conveyour of the mode to cross, i.e., to travel through the junction.
- From a junction and a pair of modes (m and m′ of that junction) one can observe, obs_TI, a time duration which represents the normal time it takes to transfer freight from a conveyour of mode m to a conveyour of mode m′. (The two modes may be the same.)

- From a junction and a mode (of that junction) one can observe, obs_arr_TI, a time duration for an item of freight destined for a normal conveyour of the mode to arrive and be "entry" processed (including loaded) at that junction.
- From a junction and a mode (of that junction) one can observe, obs_dep_TI, a time duration for an item of freight destined for a normal conveyour of the mode to arrive and be "exit" processed (including unloaded) at that junction.
- One can add lengths.
- One can add time durations. ■

Route Lengths

One can compute the length of a route of a net and one can find the shortest such route between two identified junctions.

```
value
  length: R → N ~→ L
  length(r)(n) ≡
    case r of
      ⟨⟩ → 0,
      ⟨(jf,si,jt)⟩ → obs_L(xtr_S(si,n)),
      ⟨(ji1,sii,ji2),(jj1,sij,jj2)⟩^r′ →
        let si=xtr_S(sii,n),sj=xtr_S(sij,n) in
        obs_L(si) + obs_L(sii,xtr_J(ji2,n),sij) + length(⟨(jj1,sij,jj2)⟩^r′) end
    end
    pre: r ∈ routes(n) ∧ ji2=jj1
  shortest_route: Ji × Ji → N ~→ R
  shortest_route(jf,jt)(n) ≡
    let rs = routes(n) in
    let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=jf ∧ last_Ji(r)=jt} in
    let sr:R • sr ∈ crs ∧ ∼∃ r:R • r ∈ crs ∧ length(r)(n)<length(sr)(n) in
    sr end end end
    pre: {jf,jt}⊆obs_Jis(n) ∧ jf≠jt
```

Annotations:

- The length of a single modality route of a net
  - ⋆ is 0 if the route is empty,
  - ⋆ otherwise it is the length of the first segment of the route plus the length of the rest of the route computed as follows:
    - ◇ If the route consists of just one segment, then 0,
    - ◇ else, the length of the junction from incident segment to emanating segment plus
    - ◇ the length of the rest of the route computed as otherwise specified above.

- The shortest route of a net between two of its identified junctions (the precondition) can be abstractly determined as follows:
  - ⋆ First we find all the routes, rs, of the net.
  - ⋆ Then we find those routes, crs, whose first and last junction identifications are the given ones, jf and jt.
  - ⋆ Amongst those we find a shortest one, that is, one, in crs, for which there are no shorter routes, r, in crs.

■

Route Traversal Times

One can find the total time it takes to traverse a route, including the times it takes to pass through a junction, and one can find the quickest route between two identified junctions.

```
all_time: R → N → TI
all_time(r)(n) ≡
    obs_arr_TI(xtr_J(first_J(r),n),obs_M(first_S{r}))
    + time(r)(n)
    + obs_dep_TI(xtr_J(last_J{r},n),obs_M(last_S(r)))
time: R → N → TI
time(r)(n) ≡
    case r of
        ⟨⟩ → 0,
        ⟨(jf,si,jt)⟩ → obs_TI(xtr_S(si,n)),
        ⟨(ji1,sii,ji2),(jj1,sij,jj2)⟩^r′ →
            let si=xtr_S(sii,n),sj=xtr_S(sij,n) in
            obs_TI(si) + obs_TI(sii,xtr_J(ji2,n),sij) + time(⟨(jj1,sij,jj2)⟩^r′) end
    end
    pre: r ∈ routes(n) ∧ ji2=jj1
quickest_route: Ji × Ji → N → R
quickest_route(jf,jt)(n) ≡
    let rs = routes(n) in
    let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=jf ∧ last_Ji(r)=jt} in
    let qr:R • qr ∈ crs ∧ ∼∃ r:R • r ∈ crs ∧ all_time(r)(n)<all_time(qr)(n) in
    qr end end end
```

Function Lifting

Notice how the two functions shortest_route and quickest_route differ only by the length, respectively the time functions. Hence:

```
type
    Q
    FCT = R → N → Q
```

```
value
    less: Q × Q → Bool
    lowest: Ji × Ji → N → FCT → R
    lowest(jf,jt)(n)(fct) ≡
        let rs = routes(n) in
        let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=jf ∧ last_Ji(r)=jt} in
        let lr:R • lr ∈ crs ∧ ∼∃ r:R • r ∈ crs ∧ less(fct(r)(n),fct(qr)(n)) in
        lr end end end
```

Similarly one could also lift the 'less' predicate:

```
    Q
    PRE = Q × Q → Bool
    FCT = R → N → Q
value
    best: Ji × Ji → N → FCT → PRE → R
    best(cf,ct)(n)(fct)(pre) ≡
        let rs = routes(n) in
        let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=cf ∧ last_Ji(r)=ct} in
        let br:R • lr ∈ crs ∧ ∼∃ r:R • r ∈ crs ∧ pre(fct(r)(n),fct(qr)(n)) in
        br end end end
```

Transportation Costs

We can further assess (i) transport cost (tk:TK), (ii) lowest (per unit) freight cost (fk:FK) between two junctions, etc. We assume that if a freight item is transported into a junction and out of that junction by the same modality conveyour, then it is not reloaded, i.e., along segments of the same modality.[6]

```
type
    TK, FK, K = TK|FK
value
    obs_TK: (S|J) → TK
    obs_FK: (S|J) → FK

    +: K × K → K
    cost: R → N → K
    cost(r)(n) ≡
        case r of
            ⟨⟩ → 0,
            ⟨(jf,si,jt)⟩ →
```

[6]This grossly simplifying assumption will be removed later. For the time being it allows us to operate with the simple notion of routes that was introduced above. For the reloading case we need to decorate the route notion, effectively making it into a bill of ladings notion: one that prescribes possible reloading at junctions.

```
        obs_K(xtr_J(jf,n))+obs_K(xtr_S(si,n))+obs_K(xtr_J(jt,n))
     ⟨(jf,si,jt),(jf′,si′,jt′)⟩⌢r′ → assert: jt=jf′
        obs_K(xtr_J(jf,n))+obs_K(xtr_S(si,n))+...+cost(r′)
   end

cheapest: Ji×Ji → N → ((K×K)→K) → ((K×K)→Bool) → R
cheapest(jf,jt)(n) ≡
   best(jf,jt)(n)(λ(k1,k2):(K×K)•k1+k2)(λ(k1,k2):(K×K)•k1<k2)
```

6.2.12 Road Nets

We wish to view road nets at different levels of abstraction. At a most detailed such level we make no distinction between the road kinds, whether community roads, provincial roads, motor roads or freeways. At another level of abstraction we wish to make exactly those distinctions. And at least detailed level of abstraction we consider certain road junctions to designate road nets of smaller or larger communities.

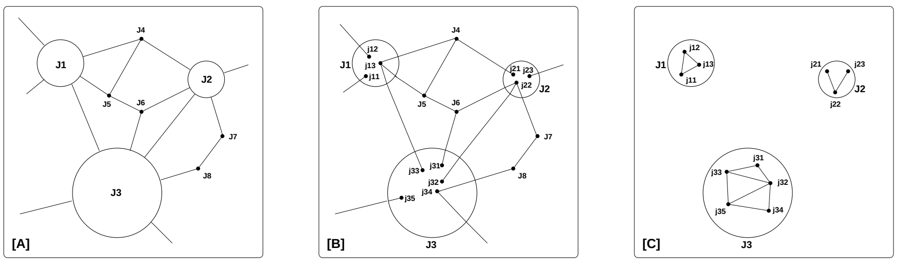


Fig. 6.5. Gross [A] versus semi-detailed [B] road net — and [C]ommunity road nets

Figure [A] 6.5 shows a road net. Instead of showing junctions J1, J2 and J3 as small black disks we show them as larger circles — for reasons that transpires from Fig. [B] 6.5.

Junctions J1, J2 and J3 are considered composite, that is, to represent communities.

We may consider the road net of Fig.[A] 6.5 to be an abstraction of the road net hinted at in Fig.[B] 6.5.

Junctions j11, j12, . . . , j35 are considered simple embedded junctions.

We decide to allow three kinds of junctions:

composite, simple embedded and simple.

They are as follows:

Composite junctions stand for road nets themselves. The junctions of those road nets are all simple embedded junctions. Simple embedded junctions are the junctions, hence, of composite junction road nets. Simple junctions are those junctions which are not composite (that is: are not standing for road nets) and are not simple embedded junctions (that is: simple, hence un-embedded junctions are those remaining junctions of a net which include modality road).

In Fig. [B] 6.5 on the preceding page we have left out the internal roads, that is, segments of junctions J1, J2 and J3, that is between the simple embedded junctions j11, j12 and j13, between j21, j22 and j23, and between j31, j32, j33, j34 ans j35.

The internal segments of junctions J1, J2 and J3 are shown in Fig. [C] 6.5 on the facing page. They are to be considered complete nets “in and by” themselves.

We may consider the implied junction identifications Ji1, Ji2 and Ji3 to be names of communities.

We may consider the implied junction identifications ji11, ji12 and ji13 to abstract to J1, ji21, ji22 and ji23 to abstract to J2, and ji31, ji32, ji33, ji34 and ji35 to abstract to J3.

We shall assume that from these junction identifications, say jikℓ, one can observe the more abstract junction identifications, i.e., Jik.

We shall, conversely, assume that from segment junction identifications one can observe whether they are identifications of composite, of simple embedded or of simple junctions, and, if of composite junctions, that one can further observe which simple embedded junction of the composite junction the segment is connected to.

In summary: When consider any multi-modality net and from it project, that is, consider only the net, n_r, of modality road, then we may find that some junctions are composite while are are simple. When then examining the road nets, r_n, contained in composite junctions then we will find that their junctions are simple embedded. The embedded road nets, r_n, otherwise satisfy all the properties (i.e., axioms) of nets in general. To link up the segments of n_r incident upon, that is, connected to composite junctions (in n_r) we provide their junction identifications with two levels of observability: the abstract one that made us see that they were connected to composite junctions (cf. Fig. [A] 6.5 on the preceding page), and a concrete one that enables us to decide which ones of the simple embedded junctions they are “finally” linked to (cf. Fig. [B] 6.5 on the facing page).

```
type
    M == road | ...
    Jc, Js, Jse
    Jic, Jis, Jise
    J = Jc | Js | Jse
    Cn
value
    is_composite_J: J → Bool
    is_simple_J: J → **Bool**
    is_simple_embedded_J: J → **Bool**
    obs_N: Jc → N
    obs_Jic: Jc → Jic, obs_Jis: Js → Jis, obs_Jise: Jse → Jise
    obs_Cn: Jic → Cn, obs_Cn: Jise → Cn
    obs_Jise: Jic → Jise
axiom
    ∀ j:Jc • is_composite_J(j) ∧ xtr_Ms(obs_N(j,road))={road},
    ∀ j:Js • is_simple_J(j),
    ∀ j:Jse • is_simple_embedded_J(j)

    ∀ n:N,j:J • j ∈ obs_Js(n) ∧ is_composite_J(j) ⇒
        let rn = obs_N(j) in

        end
```

6.2.13 Railway Nets

General

A transportation net of modality railway has segments be lines between stations and have junctions be stations.

We concretise the concept of modes. Mode m=railway will now designate railway nets:

```
type
    M == road | railway | ...
```

From a multi-modal transportation net we can project the railway net, rn:RN:

```
value
    proj: N × {railway} → RN
```

Junctions of a transportation net of modality railway have sub-junctions which are stations:

```
value
    proj: J × {railway} → ST
```

Segments of a transportation net of modality railway become lines:

```
value
    proj: S × {railway} → LI
```

Lines, Stations, Units and Connectors

Railway segments are thus called lines, and railway sub-junctions are thus called stations. A notion of connectors is introduced. It is not to be confused with the previous notion of junctions.

21. A railway net is a net of mode railway.
22. Its segments are lines of mode railway.
23. Its junctions are stations of mode railway.
24. A railway net consists of one or more lines and two or more stations.
25. A railway net consists of rail units.
26. A line is a linear sequence of one or more linear rail units.
27. The rail units of a line must be rail units of the railway net of the line.
28. A station is a set of one or more rail units.
29. The rail units of a station must be rail units of the railway net of the station.
30. No two distinct lines and/or stations of a railway net share rail units.
31. A station consists of one or more tracks.
32. A track is a linear sequence of one or more linear rail units.
33. No two distinct tracks share rail units.
34. The rail units of a track must be rail units of the station (of that track).
35. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.
36. A rail unit has one or more connectors.
37. A linear rail unit has two distinct connectors. A switch (a point) rail unit has three distinct connectors. Crossover rail units have four distinct connectors (whether simple or switchable), etc.
38. For every connector there are at most two rail units which have that connector in common.
39. Every line of a railway net is connected to exactly two distinct stations of that railway net.
40. A linear sequence of (linear) rail units is an acyclic sequence of linear units such that neighbouring units share connectors.

```
type
21. RN  = {| n:smN • obs_M(n)=railway |}
22. LI  = {| s:S • obs_M(s)=railway |}
23. ST  = {| c:C • obs_M(c)=railway |}
   Tr, U, K

value
   24.  obs_LIs: RN → LI-set
   24.  obs_STs: RN → ST-set
   25.  obs_Us: RN → U-set
   26.  obs_Us: LI → U-set
   28.  obs_Us: ST → U-set
```

```
31.  obs_Trs: ST → Tr-set
35.  is_Linear: U → Bool
35.  is_Switch: U → Bool
35.  is_Simple_Crossover: U → Bool
35.  is_Switchable_Crossover: U → Bool
36.  obs_Ks: U → K-set

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

axiom
24. ∀ n:RN • card obs_LIs(n) ≥ 1 ∧ card obs_STs(n) ≥ 2

26. ∀ n:RN, l:LI • l ∈ obs_LIs(n) ⇒ lin_seq(l)

27. ∀ n:RN, l:LI • l ∈ obs_LIs(n) ⇒ obs_Us(l) ⊆ obs_Us(n)

28. ∀ n:RN, s:ST • s ∈ obs_STs(n) ⇒ card obs_Us(s) ≥ 1

29. ∀ n:RN, s:ST • s ∈ obs_LIs(n) ⇒ obs_Us(s) ⊆ obs_Us(n)

30.  ∀ n:RN,l,l′:LI•{l,l′}⊆obs_LIs(n)∧l≠l′⇒obs_Us(l)∩ obs_Us(l′)={}

30.  ∀ n:RN,l:LI,s:ST•l ∈ obs_LIs(n)∧s ∈ obs_STs(n)⇒obs_Us(l)∩ obs_Us(s)={}

30.  ∀ n:RN,s,s′:ST•{s,s′}⊆obs_STs(n)∧s≠s′⇒obs_Us(s)∩ obs_Us(s′)={}

31.  ∀ s:ST•card obs_Trs(s)≥1

32.  ∀ n:RN,s:ST,t:Tr•s ∈ obs_STs(n)∧t ∈ obs_Trs(s)⇒lin_seq(t)

33.   ∀ n:RN,s:ST,t,t′:Tr•s ∈ obs_STs(n)∧{t,t′}⊆obs_Trs(s)∧t≠t′
            ⇒ obs_Us(t) ∩ obs_Us(t′) = {}

38. ∀ n:RN • ∀ k:K •
          k ∈ ∪{obs_Ks(u)|u:U•u ∈ obs_Us(n)}
             ⇒card{u|u:U•u ∈ obs_Us(n)∧k ∈ obs_Ks(u)}≤2

39. ∀ n:RN,l:LI •  l ∈ obs_LIs(n) ⇒
       ∃ s,s′:ST • {s,s′} ⊆ obs_STs(n) ∧ s≠s′ ⇒
```

```
let sus=obs_Us(s),sus′=obs_Us(s′),lus=obs_Us(l) in
∃ u,u′,u″,u‴:U • u ∈ sus ∧
    u′ ∈ sus′ ∧ {u″,u‴} ⊆ lus ⇒
    let sks = obs_Ks(u), sks′ = obs_Ks(u′),
          lks = obs_Ks(u″), lks′ = obs_Ks(u‴) in
    ∃!k,k′:K•k≠k′∧sks ∩ lks={k}∧sks′ ∩ lks′={k′}
end end
```

6.2.14 Net Dynamics

By net dynamics we shall mean the changing possibilities of flow of conveyors (cars, trains, aircraft, ships, etc.) along segments and through junctions. We speak of direction of flow along segments in terms of *"from the junction at one end of the segment to the junction at the other end"*. And we speak of flow through a junction as *"proceeding from one segment incident upon the junction into a (usually different) segment emanating from that junction"*. Segments connected to a junction are both incident upon that junction and emanates from that junction.

Segment and Junction States

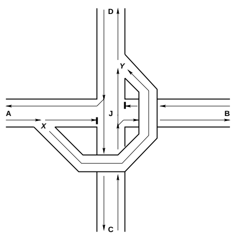


Fig. 6.6. A Special "Carrefour" Junction

Segments may be open for traffic in either or both directions (between the segments' two junctions [identified by ji_x and ji_y]) or may be closed. We model the state, $s\sigma : S\Sigma$, of a segment, $s : S$, as a set of pairs of junction identifications, namely of the two identifications of the junctions that the segment connects. This state, $s\sigma : S\Sigma$, is either empty, i.e., the segment is closed ($\{\}$), or has one pair, $\{(ji_x, ji_y)\}$, that is, the segment is open in direction from junction ji_x to junction ji_y, or another pair $\{(ji_y, ji_x)\}$, or both pairs $\{(ji_x, ji_y), (ji_y, ji_x)\}$, that is, is open in both directions. Junctions may direct traffic from any subset of incident segments to any subset of emanating segments. We model the state, $j\sigma : J\Sigma$, of a junction, $j : J$, as a set of pairs of segment identifications, namely of identifications of segments connected to the junction. Let the set of identifications of segments connected to junction j be $\{si_1, si_2, ..., si_m)\}$. If, in some state, $j\sigma$ of the junction, it is possible (allowed) to pass through the junction from the segment identified by si_j to the segment identified by si_k, then the pair (si_j, si_k) is in $j\sigma$. The junction state may be empty, i.e., closed: no traffic is allowed through the junction. Or the junction state may be "anarchic full", that is, it contains all combinations of the pairs of identifiers of segments incident upon the junction.

```
type
    SΣ = (Ji×Ji)-set
    JΣ = (Si×Si)-set
value
    obs_SΣ: S → SΣ
    obs_JΣ: J → JΣ

    xtr_Jis: SΣ → Ji-set
    xtr_Jis(sσ) ≡ {ji|ji:Ji • (ji,_) ∈ obs_sσ ∨ (_,ji) ∈ obs_sσ}
    xtr_Sis: JΣ → Si-set
    xtr_Sis(jσ) ≡ {si|si:Si • (si,_) ∈ obs_jσ ∨ (_,si) ∈ obs_jσ}
axiom
    ∀ s:S • xtr_Jis(obs_SΣ(s)) ⊆ xtr_Jip(s),
    ∀ j:J • xtr_Sis(obs_JΣ(j)) ⊆ xtr_Sis(j)
```

Observations:

- A junction, $j : J$, of just one segment, $s : S$, that is, s is a cul de sac, may either be closed, and vehicles trying to enter j will be queued up, or it is open, and vehicles entering j will be lead back to s.
- As a consequence segment s, in order for this latter routing to happen, must be open in both directions when j is "open".
- In general, if the state of a junction j (identified by ji) contains a pair (si_x, si_y) then the state of the designated segments, sx and sy, must respectively contain pairs (ji', ji), respectively (ji, ji''), where $\{ji, ji'\}$ and $(ji, ji''\}$ are the pairs of junction identifications associated with si_x and si_y respectively.
- And this must hold for all states of junctions and adjacent segments.
- This is captured in the axioms below.

```
axiom
    ...
```

The junction of Fig. 6.6 shows four segments, identified by A, B, C and D. The figure also suggests a state in which traffic lights prohibit movements from A into J, from B into J, from C via J into A, and from D via J into B. The "bypass" from A/X into Y/D appears to be such that traffic can always pass from A into D. The current state alluded to in Fig. 6.6 on page 163 appears to be:

$$j\sigma_J : \{(A,D),(C,B),(C,D),(D,A),(D,C)\}$$

(A,D) is potentially a member of every state that the junction can possibly be in — see next section.

Segment and Junction State Spaces

A state space is a set of states. A segment can be in one of several segments states. A junction can be in one of several junction states. Hence we introduce segment and junction state spaces.

```
type
    SΩ = SΣ-set
    JΩ = JΣ-set
value
    obs_SΩ: S → SΩ
    obs_JΩ: J → JΩ
axiom
    ∀ s:S • obs_SΣ(s) ⊆ obs_SΩ(s),
    ∀ j:J • obs_JΣ(j) ⊆ obs_JΩ(j)
```

6.2.15 More on Net Dynamics: Traffic

Vehicles and Positions

There is a further undefined notion of vehicles, V. And there is a notion of the position, P, of a vehicle. Either a vehicle is positioned in a junction, and then its position is designated by the junction identifier. Or a vehicle is positioned along a segment, and then its position is designated by a triplet: the identifier of the junction it is moving away from, the identifier of the junction it is moving towards, and the fraction of the distances from the position to the two junctions: If the fraction is 0, then the vehicle has just entered the segment, if the fraction is 1, then the vehicle is just about to leave the segment, and, hence, if the fraction is a proper real between o and 1, but neither 0 nor 1, then the vehicle is properly within the segment.

```
type
    F = {|f:Real•0≤f≤1|}
    P == mkP_at_J(ji:Ji) | mkP_along_S(fji:Ji,f:F,tji:Ji)
```

Traffic

Traffic is now a function from time to a pair of a net, and the positions of vehicles within the net.

```
type
    V
    T
    TF = T $\overrightarrow{m}$ (N $\times$ (V $\overrightarrow{m}$ P))
```

Proper Vehicle Positions

The positions of a traffic must designate proper junctions of the net.

```
axiom
    $\forall$ tf:TF •
        $\forall$ t $\in$ dom tf •
            let (n,vps) = tf(t) in
            $\forall$ p:P • p $\in$ rng vps $\Rightarrow$
            case p of
                mkP_at_J(ji) $\rightarrow$ ji $\in$ obs_Jis(n),
                mkP_along_S(jf,_,jt) $\rightarrow$ {jf,jt}$\subseteq$obs_Jis(n)
            end end
```

Other Traffic Constraints

Traffic must be smooth: Positions of vehicles do not “jump around”, i.e., movement are monotonic. No “ghost vehicles”: If at times t and t' considered close to one another a vehicle is in the traffic then it is also in the traffic at all times in between t and t'. We omit the formalisations of these constraints.

6.2.16 Time Tables and Traffic

By a time table we understand an entity which to named transport vehicles associate journey descriptions. By a journey description we understand a sequence of junction visits. By a junction visit we understand a triple: Arrival time, junction identifier and departure time.

```
type
    TT = Vn $\overrightarrow{m}$ Journey
    Journey = (at:T $\times$ ji:Ji $\times$ dt:T)$^*$
```

Scheduling

By scheduling we shall here, in a narrow sense, understand a function from nets and time tables to a possibly infinite set of traffics such that each traffic satisfies the time table.

```
value
    sched: TT → N → TF-infset
    sched(tt)(n) as tfs
        pre: wf_TT_and_N(tt,n)
        post: ∀ tf:TF • tf ∈ tfs ⇒ wf_TF(tf) ∧ sat(tf,tt)

    wf_TT_and_N: TT × N → Bool, ...
    sat: TF × TT → Bool, ...
```

6.3 And so on!

We have shown fragments of a description of a domain of transportation nets. There is, of course, much more. “Years of work still to be done!” But, for the time being we have enough to illustrate some reasonably interesting requirements.

6.4 A Set of Requirements

We shall consider the following four sets of requirements: (i) requirements for software to monitor net maintenance, (ii) requirements for software to monitor & control net traffic, (iii) requirements for software to simulate net traffic, and (iv) requirements for software to support transport logistics: optimal routes etc.

6.4.1 Plan of Development of Requirements

The plan is now to first give a brief, rough sketch narrative of the four sets of requirements. We do so, here, in this paper, in an unusual way. First we ‘extend’ the domain description given earlier. Then we ‘project’, ‘instantiate’, and make less non-deterministic (‘determination’) the extended domain description, that is: We transform the domain description into domain requirements prescriptions. But first we present the domain extensions. After that the plan is to analyse these four domain extension sketches wrt. such common “features” that may be shared by the four (or triples of three or pairs of two) software implementations; to present the requirements for each of the four specific software “packages”; and finally to present the requirements for such a shared “core” of software. That is, we are ‘fitting’. ‘Extension’, ‘projection’, ‘instantiation’, ‘determination’ and ‘fitting’ are three major domain description-to-requirements prescription operations. (See Chap. 19 of [33] for details.)

Brief Narratives of Four Domain Requirements

By domain requirements we understand requirements that can be expressed by sôlely using terms of the domain (and ordinary, non-technical language).[7] In this paper we shall only consider domain requirements. Of course, many, if not most of the interesting problems of software development in relation also to ‘problem frames’ may be those due to interface and machine requirements.

A Caveat on Formalisation

We shall not formalise our narratives. Instead we refer to [33,41,43,44]. To do justice to a proper, “non-brief” narrative and its proper formalisation means that we present systematically enumerated narratives and corresponding formalisations to an extent that would trip the size of this chapter. The aim of this chapter is not to show domain-to-requirements ‘transformations’, but to relate the triptych approach to Jackson’s Problem Frame approach.

6.4.2 ‘Net Maintenance’ Software

We propose a (parameterised) software package to be developed for monitoring and supporting the management of the maintenance of both road and rail nets. An instantiation parameter (road,rail) shall determine whether the package works for road or for rail nets.

Domain Description – An Extension

Segments and junctions need be maintained, that is, we may associate a set of quality attributes related to the upkeep of segments and junctions, as well as of any traffic signals associated with these, we may further associate actual and estimated date(s), cost(s), and duration(s) of previous and next maintenance services, etc., and we may keep “such” records of all segments, junctions and signals of the net. To monitor the net quality attributes, in the domain, some need perform work that help advise maintenance staff to evaluate and report quality attributes of segments, junctions and signals, follow-up on missing such

[7]By machine requirements we understand requirements that can be sôlely expressed using terms of the machine (and ordinary, non-technical language). By interface requirements we understand requirements that can be expressed only by using terms of both the domain and the machine (and ordinary, non-technical language).

reports, and help update the attributes of the records kept when reported. To support the management of net maintenance some need perform, in the domain, work that help management schedule and allocate resources for the monitoring of net quality and corresponding update of records, for the actual maintenance work, and for handling "unforeseen" reports on segment, junction and signal malfunctioning (i.e., in need of repair).

Domain Requirements

Entities

Segments and junctions need be maintained, that is, we must associate a set of quality attributes related to the upkeep of segments and junctions, as well as of any traffic signals associated with these, we must further associate actual and estimated date(s), cost(s), and duration(s) of previous and next maintenance services, etc., and we must keep "such" records of all segments, junctions and signals of the net.

Monitoring Functions

To monitor the net quality attributes, in the domain, the software must have functions that help advise maintenance staff to evaluate and report quality attributes of segments, junctions and signals, follow-up on missing such reports, and help update the attributes of the records kept when reported.

Management Functions

To support the management of net maintenance the software must have functions that help management schedule and allocate resources for the monitoring of net quality and corresponding update of records, for the actual maintenance work, and for handling "unforeseen" reports on segment, junction and signal malfunctioning (i.e., in need of repair). ... HERE FOLLOWS PRECISE REQUIREMENTS DETAILS (OMITTED) ...

Domain to Requirements Operations

We give a terse summary. Projection: Most of the net attributes have been kept. Many of the concepts (routes, ..) and evaluation functions (time, length, ...) have been "projected away". Instantiation: Usually the software, when delivered to a client, is instantiated to the specific net characteristics of the client. Determination: No example as looseness and non-determinism is basically absent from this part of the domain. $\mathcal{E}$TCETERA!

6.4.3 'Traffic Control' Software

We propose a software package to be developed for monitoring and controlling road net traffic not just at local junctions but along segments, and providing for "green" flow along certain route directions.

Domain Description – A Rough Sketch Extension

Traffic control in the conventional, non-technological net domain is done by traffic police controlling junction flows or by local sensors and actuators positioned near junctions; sensors monitor only local traffic and actuators control only local junction semaphores. An assessment is made (by police or sensors) of local traffic density only, and appropriate arm signals or semaphore signalling (red, yellow, green) acts as controls.

Domain Requirements

Net Representation "In the Machine"

The road net must be represented: segments, junctions and signals. Signals must be controlled. Segment, junction and signal states must be represented. Segment lengths and segment and junction (e.g., average) "traversal" times must be represented. Vehicle positions in segments and junctions must be represented. Vehicle positions must be monitored. We assume sensors to record and inform of "density" of vehicles at segment lanes in vicinity of junctions and leading into these. ... HERE FOLLOWS PRECISE REQUIREMENTS DETAILS ...

Traffic Monitoring Functions

Functions shall regularly sample traffic density. There must be functions for inquiring about and reporting on unusual traffic situations (accidents, fog, road conditions in general). It is assumed that there are functions which otherwise report on the statues of the road net. (That is, functions which relate to the net maintenance software.) ... HERE FOLLOWS PRECISE REQUIREMENTS DETAILS ...

Traffic Control Functions

The objective of the use of these functions is to ensure smooth traffic. Individual functions shall determine the setting of signals at junctions. Composite functions shall determine the setting of signals, say in "green waves" along routes — hence the road net representation must be augmented with information about major and minor routes, time of day preferred directions: am "into town", pm "out of town", and the like. ... HERE FOLLOWS PRECISE REQUIREMENTS DETAILS (OMITTED) ...

Domain to Requirements Operations

Projection: Only the junction and segment state attributes need be kept. Instantiation: The net is instantiated to a particular road net of a particular city, i.e., that of the client. Determination: Some segments are designated as priority segments, with determined directions being "favoured" for "green traffic flow" at determined time intervals of the day. Accordingly some junction state transitions are "favoured" over others. $\mathcal{E}$TCETERA!

6.4.4 'Traffic Simulation' Software

We propose a software package to be developed for simulating road net traffic. In the domain there is, we assume, as yet no such simulation software. So we cannot domain describe what we mean by simulation — or rather: any such domain description becomes the domain requirements.

Net Representation

Net representation " in the machine": The road net must be represented: segments, junctions and signals. Segment, junction and signal states must be represented. Segment lengths and segment and junction (e.g., average) "traversal" times must be represented. Vehicle positions in segments and junctions must be represented. Assumptions: Vehicles, when moving, move at statistically determined velocities, etc.

Simulation Concepts

We suggest, not as part of the requirements, but as a software implementation idea, the following two ideas:

Representation of segment geodetic profile: A segment is decomposed into geodetic blocks. The curvature of each block is represented by two 3D vectors, from which a Bezier curve for that block can be constructed.

Representation of segment velocity profile: A segment is decomposed into velocity blocks. The increase/decrease of speed for each block can be represented by two 2D vectors, from which a Bezier velocity curve for that block can be constructed: The computation of the curve will, depending on vector characteristics (long or short vectors), compute close, or less close, or "far away" points on the curve, and we shall take the varying density of these computed points to designate positions of a vehicle at any one time, one vehicle per computation of the velocity curve.

Traffic Simulation Functions

Here are some functions: (i) initialise states of segments and junctions wrt. signals; (ii) initialise states of segments and junctions wrt. vehicle positions. That is: (iii) allow vehicles to start their journey along segments and in junctions when the simulation begins, and/or at different times during the simulation (say according to some time table). (iv) Schedule simulation interval and resolution (granularity, i.e., one unit of simulation time equals r units or real time.[8]); (v) "play, stop, recommence" simulation; (vi) change granularity while "playing"; and (vii) insert vehicles during simulation.

[8] r can be any real above 0. If r is less than 1 simulation is microscopic; if it is 1 simulation is "real"; if it is larger than 1 simulation is macroscopic.

Domain to Requirements Operations

Projection: We project away almost all but the net and time tables. We adhere to definition of traffic (i.e., TF). Instantiation: We instantiate to a specific net. Determination: We may decide to constrain to segment-determined constant velocity traffic. $\mathcal{E}$TCETERA!

6.4.5 'Transport Logistics' Software

We propose a software package to be developed for supporting freight (incl. container) transport logistics.

Domain Description – An Extension:

In the domain planning a journey, for travelling (on a crucial trip) as a passenger on trains, by bus, airplane or by ship, usually requires the use of one or more time tables. Considerations of alternative routes, of multi modal travel, of cost: fast, perhaps expensive, hurried travel versus slower, perhaps less costly, and of overnight stays en route may be important. This applies to freight transport too: refrigeration of freight load, "first to market", etc.

Domain Requirements

Net Representation "In the Machine"

The multi modal net must be represented: segments and junctions Segment lengths and average traversal times and traversal costs of segments and junctions[9] must be represented — usually the latter (times and costs) are provided by transport vehicle (truck, train, boat and aircraft) time tables. We may thus discover that we need to extend our domain description: Junction hubs, where freight is transferred from one modality transport to another, may need be further detailed, e.g., as to warehouse facilities (godowns), etc.

Logistics Functions

$\mathcal{E}$tcetera !

Domain to Requirements Operations

Projection:

The net, its segments and junctions, their length, time, and cost attributes. Also time tables. Most functions related to these. Instantiation: Maybe we instantiate to only a shipping net, or only a rail net? Determination: As a representation of the segment and junction traversal times we may rely on the time tables. $\mathcal{E}$TCETERA!

[9]The traversal time and cost of junctions could be differentiated wrt. modalities: freight being unload/loaded when incoming and outgoing segment modalities are different, etc.

6.4.6 Requirements Prescription of Shared Software

All four rough sketch requirements prescriptions projected into their requirements a core of the net, its segments and junctions. We therefore conclude that a repository, i.e., a database, is needed, one in which representations of segments and junctions are stored. A repository (software system) which allows flexible representation of segment and junction attributes, their initialisation, retrieval and update. So we decide on using some relational database management system.

Net Repository

Informal Rough Sketch

Segment representations are in the form of relation tuples. Segment attributes are attributes of relations. Junction representations are in the form of relation tuples. Junction attributes are attributes of relations.

Formalisation – a Very Rough Sketch

```
type
   SR = ST-set
   JR = JT-set
   ST :: si:Si ftj:Jip m:M le:L ti:TI k:K f:F sσ:SΣ sω:SΩ
   JT :: ji:Ji si:Si-set m:M ti:TI k:K f:F jσ:JΣ jω:JΩ
```

This is not quite first normal form relational representation. A junction connected to n segments and with a state-space of m possible states — in (primitive) first normal form would require $m \times n$ tuples. Of course "smarter" ways of representing sets of segment identifiers and state space (ω) can be devised. That is not a requirements issue, but a software design issue.

Repository Functions

Rough Sketch Ideas

The observer functions of the domain description are now simple tuple projections. Query facilities offered by the relational DBMS[10] being deployed can be used in connection with many of the functions transformed from the domain description into the specific domain requirements prescriptions. They are the functions that make "heavy" use of observer functions. The various domain requirements prescriptions additionally prescribe repository initialisation and refreshment (i.e., update) functions — and again their design and implementation can be greatly facilitated by the update functions of the chosen relational DBMS. Of course, queries "against" an RDBMS really deposit results in a designated workspace and displays this on the GUI.

[10]DBMS: Database Management System, like FRONTBASE WWW.FRONTBASE.COM the best, or DB2 www.ibm.com/db2 or SQL www.oracle.com.

Specific Function Signatures

value
  obs_Jip: S → Jip
  sql_project: RelNm×{|″`si=seg_name`″|}×{|″`ftj`″|}×Wn → Jip×GUI

The former function is the "further undefined" domain specification observer function. The latter function "approximates" an SQL query — where we do not show the functional arguments for the RDBMS and the workspace.

"General" Function Signatures

We intimate database retrieve (query, observer), initialise, and refresh (update), function signatures:

value
  query: retrieve_function × RDBMS × Wn → GUI
  init: (S|J)-**set** × RDBMS → RDBMS × GUI
  refresh: (S|J)-**set** × RDBMS → RDBMS × GUI

6.5 And So On — What Have we Covered

We have given a rather large fragment of a *domain description*. We have postulated and given small fragments of four *domain requirement prescriptions*. We have indicated how these domain requirements were *"derived"* from the domain description. We have formalised the domain description. We hardly formalised the domain requirements. But could (easily) do that! The four domain requirements reflect *different problem frames*.

We claim to have intimated the following problem frames (PF):

- Common Software: II: *Information Intensive* PF.
- Maintenance: *Weak Reactive*[11] ⊕ II PF
- Traffic Control: *Strong Reactive*[12] ⊕ II PF.
- Simulation: *Computation* ⊕ *Virtual Real-time* ⊕ II PF.
- Logistics: *Computation* ⊕ II PF.

6.6 The Triptych and the Problem Frame Approaches

6.6.1 General Observations

The triptych approach advises that software development includes: domain engineering (DE), requirements engineering (RE), and software design (SD). The triptych approach does not replace the PF approach. To me the triptych approach augments, supplements the PF approach.

[11]Weak reactive: Non real-time
[12]Strong reactive: "Critical" (i.e., hard) real-time

6.6.2 Specific Observations

The triptych approach does not mandate strict linear adherence to DE $\rightarrow$ RE $\rightarrow$ SD but assumes DE $\leftrightarrow$ RE $\leftrightarrow$ SD $\leftrightarrow$ DE iteration. In fact: It is impossible to "discover" all that is relevant about the domain before proceeding to understand the requirements, and all that is relevant about the requirements before proceeding to design the software, $\mathcal{E}$tcetera!

6.7 Grand Challenges of Computing Science

6.7.1 The Grand Challenge of VSTTE

The GC of VSTTE[13] to me appears to focus on "a million lines" of program code that to me appears to be verified with respect to program code annotations where it is not clear to what extent those annotations relate to properties of the code, to requirements, and to domain assumptions.

6.7.2 The Grand Challenge of Ubiquitous Computing

The grand challenge of *ubiquitous computing* appears to offer a very nice opportunity for a "foothill"[14] experimental project. Take the proposed *Automated Highway* project. As it could be conceived one is thinking of deploying computers and communication wherever feasible (sometime in future) in the safe and efficient driving of cars, in sorting out cross traffic, etc. So here a far more detailed domain description of transportation nets than intimated here is needed. Etcetera!

6.8 Conclusion

So I immodestly propose that research into and use of the PF approach be augmented by research into and use of the triptych approach, and to adjoin the ("otherwise") highly laudable VSTTE effort with some serious, viz., triptych-oriented program code development. I hope to be able to contribute to the grand challenge of ubiquitous computing.

6.9 Bibliographical Notes

We refer to [31–33] for a complete coverage of informal as well as formal abstraction and modelling principles and techniques [31], principles and techniques specification of systems and languages [32], and principles and techniques of domain engineering, requirements engineering and related software

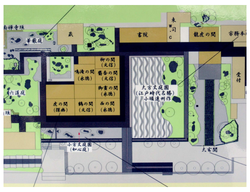
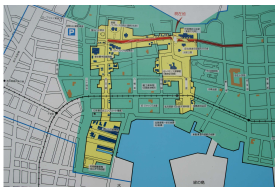
[13]VSTTE: Verified Software: Theories, Techniques and Experiments

[14]"Foothill" project: This is one of those terrible "americanisms": apparently used to characterise a pre-cursor like, or perhaps rather initial stage project.

design [33]. Chapter 28 ("Domain Specific Software Architectures) of [33] surveys a number of problem frames from the viewpoint put forward in this paper.

Part III

Experimental Evidence

7 Documents[1]

A Rough Sketch Domain Analysis

Caveat

This chapter is incomplete. Its basis, [25], is even more so. We could have wished to bring a more complete analysis of the Document domain.

Summary

The concept of document is all pervasive. It occurs in as widely different contexts as scientific, technical, business or tourism papers or reports, as sales offers, marketing brochures and personal letters, as inter- and intra-departmental memos in public administration, as legal entities such as law texts, jurisprudence and law court verdicts, as e-mails and web pages, as credit card payment slips, as music and movie recordings (subject, possibly, to DRM (digital rights management), etcetera. A patient medical report is a dossier of documents: annamnese, analysis reports, diagnostics and treatment plans. In the Danish Board of National Health report on patient medical reports (as a basis for electronic patient journals) the term document seems to take on a bewildering and not insignificantly large set of different meanings.

Here we shall look at some of the aspects of the semantics of documents: their creation, being edited (on the basis, i.e., in the context of other document), their being read, being copied, being distributed and being shredded. Thus we shall not look at the possible syntax(es) of documents nor of their pragmatics — just some of the semantics — that which is independent of the particular form (syntax) and contents of documents. That semantics, to us, includes the actors who create, edit, read, copy, distribute and shred documents, and the times and locations at which these operations take place. We shall defer to subsequent chapters (Chaps. 8–10) speculating on the issues of which kinds of actors, what kind of "need-to-operate", and, more generally, which authorisations and in which contexts actors may act.

[1]This is an edited version of [25].

7.1 Some Background Remarks

7.1.1 A Source of Software Failures

With computing and communication now supporting the creation, editing and sending/receiving (including copying) of documents there are “rich”, but dangerous grounds for misinterpretations as to what constitutes a document.

7.1.2 An Evolving Report

In this evolving[2] report we shall “ever so slowly” try establish a domain theory of documents. In doing so we may unveil insufficiencies in abstractly modeling a number of common domain phenomena and concepts in the usual property- or model-oriented specification languages such as B [1,71], CafeOBJ [89,90, 99,100], CASL [11,78,184,185], VDM-SL [55,56,95,96], RSL [31–33,44,101, 104,106], Z [132,133,229,230,242] or similar specification languages.

7.1.3 Structure of This Chapter

Section 7.3 brings a naïve model of documents. It is the basis for our dissatisfaction. We would like to express a number of domain properties more elegantly in current formal specification languages. But it seems we cannot.

7.2 What Are Documents?

7.2.1 Varieties of Documents

The above enumeration of examples of documents shall serve as an admittedly rather loose delineation of what we mean by a document: something visual or audial or audio-visual (intellectual, artistic, legal, technical, scientific) that can be created, can be read (i.e., rendered; listened to or viewed), can (in some cases) be edited, can be copied, can be moved from place to place, and can be destroyed (shredded, deleted).

One of the fundamental ideas of computer science is that “data”, like documents, can best be understood in terms of the operations on the data, i.e., the documents. So that is what we shall do. Not the operations on personal letters, or a poem, or a collection of poems; not the operations on a credit slip, or on a CD ROM of music, or a DVD recorded movie; but those operations that seem to be generic to all documents: create, render, edit, copy, move and destroy.

[2]By ‘evolving’ we mean that there should hopefully appear a sequence, a trace of edited versions of this Technical Note: from a base note (draft report). As of 12-Feb-2006 this is the base note. The present chapter is a mere, simple editing of the 12-Feb-2006 draft report.

7.2.2 On the Domain of Documents

When, as we shall now do in this chapter, and as from the next chapter, namely describe a domain of documents, then the reader will, at time, perhaps object, saying *"but that's not feasible in the domain!"*.

We shall, of course, at almost "every turn of the road", defend our position: namely that *if you can rationally think it in the domain, then it must be described.*

Thus a domain description describes **what there is.** Not how you would like it to be (that may imply either some business process re-engineering or requirements for software). In describing **what there is** we have to accept that we must describe aspects of the domain that we may not like, or that we may not think possible.

Let us taken some examples — some claims about what it is possible to describe — in our domain: We claim, as you shall soon see, that from every document, d, we can observe its entire past history: If it is an unedited copy of a prior document, d_p, then we can, from that copy, d, observe the document, d_p, from which it was copied. If d is an edited version of a document, d_p, then we claim that we can observe the un-edited version of d, i.e., d_p. We claim, as we shall later substantiate, that we can observe the time and location of when and where operations were performed on documents.

Now that's a "pretty tall story!" you should say. Can one really, from an edited document "see" its un-edited form. Yes we claim. Suppose it was hand-written. Then also editing was hand-written. And we claim that it is possible to talk about which of the hand-writing was there first, which was added as editing changes. Can one really from a copy "see" the document from which it was copied. Yes we claim. Certainly, if d_c is a copy of d, then whatever information d had d_c must have. In addition it must be possible to claim that d_c is not the master (i.e., the basis for the copy), but that d is. There are some subtle distinctions here. The main gist of these distinctions is that *it is in principle possible* to make these claims, i.e., to talk about these properties of documents, their copies, their edited versions, etc. Whether it is in practice always possible to identify these matters is another issue.

Now, whether what we claim can be observed are really the actual things, or surrogates thereof, is another matter which we shall, at the moment not touch upon. And: whether we get the correct time and location, when we, for example, say that we can observe time and location of when and where an operation was performed on a document, is another matter also: the domain is fallible. We may get a wrong answer, but an answer we get! Similarly, if we claim to be able to observe the predecessor of a copy, i.e., the master from from which the copy was made, then what we may see may not be that master, but a corrupted version of it. So be it. We see something claimed to be "such and such"!

Another, simpler way of approaching the issue of being able to observe what has happened to a document is to accept the fact that *things that have* *indeed happened has happened and therefore we can talk about them. We can also talk about which operations they were, by which actors, at which locations and at which times. We therefore think of a never failing oracle, one that always speak the truth. It is what this oracle can observe that we model.*

We shall elaborate further on the above when we, in Sect. 7.3, go through various aspects of documents.

7.2.3 Semantics of a Document Concept

Semantics is the study and knowledge, incl. specification of meaning in language. Semantics determine, it appears, how we edit documents. In this chapter we shall not be concerned about the contents of document, i.e., the information to be edited. But we shall be very much interested in the interrelated semantics of the sets of operations performed on documents. We claim that the semantics of these interrelated sets of operations is an integral part of the semantics of documents. With computing and communication now supporting the creation, editing, sending/receiving (including copying) and destruction of documents there are "rich", but dangerous grounds for misinterpretations as to what constitutes a document.

7.2.4 Syntax of Documents

By syntax we mean the ways in which words are arranged to show meaning (cf. semantics) within and between sentences, and the rules for forming syntactically correct sentences.

Since we are not, in this chapter, interested in the syntax of document information — that part of a document which is of interest to the casual user of documents — we shall not be dealing with syntaxes for document contents. But we shall, over the next many pages, be building up an implicitly expressed abstract syntax for a number of document attributes. These attributes have to do with such things as: when and where (time and location) was an operation performed on the document, was the document copied, the pre-history of the document — since creation, etc. Together this ensemble of attributes imply an abstract syntax of document, not of its contents, but of its management.

Thus our treatment shall in no way reflect the issues of either the *Open Document Architecture, ODA*[3] or the *OpenDocument*[4] or the *Office Open XML*[5].

7.2.5 Structure of This Report

Section 7.3 presents a conventional model of documents such as we have tried to encircle that phenomenon ("documents") above. That is, we shall model the create, copy, edit and move operations.

[3]http://en.wikipedia.org/wiki/Open_Document_Architecture
[4]http://en.wikipedia.org/wiki/OpenDocument
[5]http://en.wikipedia.org/wiki/Office_Open_XML

Chapter 9, along a related line of exploration, introduces the notion of document license. A license is also a rendering function: applied to a pair, a license and a document, it yields those parts of a document contents for which the agent (issuing the rendering function) has rendering rights — together, possibly, with an updated document (one that perhaps changes future license-based rendering rights on that document).

7.2.6 On Reading This Report

In the first releases of this document the formalisations are expressed in terms of the `RAISE` specification language `RSL`.

That language, besides [31–33, 44, 101, 104, 106], is also introduced in [31–33]. Those three volumes, besides, gives an extensive introduction to abstraction and modeling.

For readers not familiar with the tradition of formal specification languages whose origins goes back to John McCarthy [171–174] and Peter Landin [157–160], and which took its first form in the `VDM` meta (or specification) language [14, 55], now ISO standardised into `VDM-SL` [5, 6, 161, 201] [and of which `RSL` [104] is a derivate], we provide annotations that explain the notation. As the chapter "wears on" these annotations "peters off".

7.3 A Simple Model of Documents

7.3.1 Originals, Copies and Versions

There are documents. Documents are either *create*d, *edit*ed or copied, *copy*. One can claim that a document is either an original, oD, or an edited version, for short, a version, wD, of a document, or a copy, cD, of a document. When we edit a document there is some editing text, E. One can claim that a document can either (i) only (say: "most recently") be an original, oD, or (ii) only (say: "most recently") be an edited (a version of a), eD, document, or (iii) only (say: "most recently") be a copy, cD, of a document. The pragmatic intention of documents is to embody document content, Ct. We leave the notion of document content undefined. There is information, I. Information is either document content, C, or the absence of such, *void*. To create a document needs no document content. From a document one can observe its most recent information.

type
  D, oD, eD, cD, C, E
  I == void | C
value
  create: **Unit** → oD, edit: E × D → eD, copy: D → cD
  is_oD,is_eD, is_cD: D → **Bool**

axiom
  $\forall$ d:D •
    is_oD(d)$\vee$is_eD(d)$\vee$is_cD(d) $\wedge$
    is_oD(d)$\Rightarrow\sim$(is_eD(d)$\vee$is_cD(d)) $\wedge$
    is_eD(d)$\Rightarrow\sim$(is_oD(d)$\vee$is_cD(d)) $\wedge$
    is_cD(d)$\Rightarrow\sim$(is_oD(d)$\vee$is_eD(d))
value
  obs_I: D $\to$ I
axiom
  obs_I(create()) = void $\wedge$
  $\forall$ d:D • is_oD(d) $\Rightarrow$ obs_I(copy(d)) = void

Annotations:

- The sectioning literal **type** designates that the following text (up to a next sectioning literal) introduces abstract and concrete type definitions. An abstract type definition is like a sort.
  - ⋆ D, oD, eD, cD, C and E introduces the sorts of documents, original documents, edited documents, document copies, document contents and document editing. (We shall not elaborate further on E till Sect. 7.3.2 on the facing page.
  - ⋆ The equation I == void | C defines document information as either being void or C. (We are not here telling you what void means.) The alternatives of U == V | W | X | Y ... are, by the == constructor., being defined as disjoint types.
- The sectioning literal **value** designates that the following text (up to a next sectioning literal) introduces values of defined types. Six such values are introduced. We see from their types (... $\to$ ...) that they are all function values.
  - ⋆ create designates the create function. It is a type **Unit** $\to$ oD. Thus it takes no arguments (designated by the value literal **Unit**) and yields an original document.
  - ⋆ edit designates the editing function. It is a type E $\times$ D $\to$ eD. Thus it takes two arguments: some editing value and a document and yields an edited document.
  - ⋆ copy designates the copy function. It is a type D $\to$ cD. Thus it takes one argument, a document and yields a document: the copied document. The function signature says nothing about "what happened" to the input argument. As we shall see, it is still there, "somewhere".[6] To define *copy*, instead, with the signature copy: D $\to$ D$\times$D would give the erroneous impression that copy(d) **as** (d′,d″), where the first of the result arguments, d′, is to be the "original" and d″ the copy, might yield d$\neq$d′ whereas, what we mean is d=d′, thus we do not need the signature copy: D $\to$ D$\times$D but can do with copy: D $\to$ D.

[6]Adding 3 and 7, yielding 10, does not, in any way, destroy or influence 3 and 7.

  - ⋆ is_oD designates a predicate observer function. It is a type D → **Bool**. If the document is an original then truth is yielded, otherwise falsity.
  - ⋆ is_eD designates a predicate observer function. It is a type D → **Bool**. If the document is an edited version of a document then truth is yielded, otherwise falsity.
  - ⋆ is_cD designates a predicate observer function. It is a type D → **Bool**. If the document is a copy then truth is yielded, otherwise falsity.

  The functions are all postulated. They are claimed to exist. They are not defined. Instead their properties will be revealed through axioms.
- The sectioning literal **axiom** designates that the following text (up to a next sectioning literal) introduces a number of properties — typically over the types and values introduced before these axioms.
  - ⋆ The clause ∀ a:A • "reads": *for all values a of type A it is the case that.* In RSL all quantifications are typed.
  - ⋆ The proposition is_oD(d)∨is_eD(d)∨is_cD(d) ∧ "reads" *a document d is either an original or an edited version or a copy, and ... .*
  - ⋆ The proposition is_oD(d)⇒∼(is_eD(d)∨is_cD(d)) "reads" *if a document is an original then it is neither an edited version or a copy, and, etcetera.*
- The signature obs_I: D → I expresses a an observer function which from a document observes its information.
- The axioms obs_I(create()) = void and ∀ d:D • is_oD(d) ⇒ obs_I(copy(d)) = void expresses that copies of copies of ... of copies of originals still have no proper information content. ■

7.3.2 Editing and Versions

Editing a document modifies its information. An edited document is a version of the document from which it was edited. Editing a document does not amount to establishing a new document. From an edited document one can observe the information immediately before it was most recently edited, and how that information was edited, i.e., the resulting content. One way of modelling the edit function is by means of two functions: a forward editing function and a backward, "undo" editing function. The forward editing function takes an information argument and delivers an information result. The backward editing function takes an information argument and delivers an information results. The backward editing function is the inverse of the forward editing function.

type
  $E' = FE \times BE$
  $FE,BE = I \to I$
  $E = \{|(fe,be):E \bullet \forall\, i:I \bullet be(fe(i))=i\ |\}$
value
  $obs\_E: D \overset{\sim}{\to} E$

```
axiom
    obs_E(create()) = chaos ∧
    ∀ d:D,e,(fe,be):E
        obs_E(edit(e,d)) ≡ e ∧
        obs_I(edit((fe,be),d)) ≡ fe(obs_I(d)) ∧
        obs_I(d) ≡  be(obs_I(edit((fe,be),d))) ∧
        obs_E(copy(edit(e,d))) ≡ e ∧ ... /∗ induction ∗/
```

Annotations:

- E′ is a concrete type. It is defined as the Cartesian of two types: FE and BE.
- Both FE and BE are total functions from information to information.
- E is the subtype of E′ which constrains the backward editing function be:BE to be an inverse of the forward editing function fe:FE.
- obs_E is a partial observer function. It applies to documents.
- From an original document one cannot observe any editing functions: obs_E(create()) = **chaos**.
- From edited documents (whether since copied) one can (still) observe the editing functions.
- The parenthesised clauses: (whether since copied) and (still) are not expressed by obs_E(copy(edit(e,d))) = e, but intimated by the ellipses clause ... — to be formalised below. ■

7.3.3 Document Traces

From a document one can observe its immediate predecessor document. An original document has no predecessor. A copy, d_c of a document, d, had d as its immediate predecessor document. An edited document, also called a version,, d_e of a document, d, had d as its immediate predecessor document. And so on, "ad finitum", till the original document is encountered.

Let us call the document from which an edited version arises for the master document. And let us call the document from which a copy is made also for the master document. Thus the predecessor documents are masters wrt. the successors.

```
value
    obs_Pre: D ~→ D
axiom
    obs_Pre(create()) = chaos ∧
    ∀ d:D,e:E • obs_Pre(copy(d)) = d = obs_Pre(edit(e,d))
```

Annotations:

- From any document other than an original one can observe, obs_Pre, its predecessor.

- Thus obs_Pre(create()) is not defined, that is, is **chaos**.
- For all documents and editing functions the predecessor of a copy of d, i.e., copy(d), is d, and the predecessor of the e edited version, edit(e,d) of d is also d. ■

Observations:

- We could decide, instead of making obs_Pre a partial function, to let obs_Pre(create()) yield create().
- Then obs_Pre would be a total function.
- And then obs_Pre(copy(create())) would be "the same" as create().
- We shall review and modify our predecessor function, obs_Pre, later in this chapter.

A document trace is a history trail, i.e., a sequence of documents, from an original to the present document, whether a copy or a version such that the first document of the sequence is the document, the *i*th document in the sequence is the predecessor of the $i - 1$st document in the sequence, and hence such that the last document in the sequence is the original. Thus one can establish the full history that any document has undergone since the creation of its "ultimate predecessor".

```
value
  obs_doc_trace: D → D*
  obs_doc_trace(d) ≡
    if is_oD(d) then ⟨d⟩ else ⟨d⟩^obs_doc_trace(obs_Pre(d)) end
```

Annotations:

- We name the document trace function obs_doc_trace since is it really an observer function (it is being "defined" solely in terms of, in this case one observer function).
- The document trace of an original document is the singleton sequence of that document.
- The document trace of a copy or an edited version (d) is the prefix concatenation of the singleton sequence of that document (d) with the document trace of the predecessor document of d.
- Termination is guaranteed since only a finite number of copies and edits can have taken place on any document. ■

We can now complete the induction part of the axiom above obs_E(copy(edit(e,d))) = e ∧ ....

```
axiom
  ∀ d:D •
    ∀ i:Nat • i ∈ inds obs_doc_trace(d) ⇒
      is_eD(obs_doc_trace(d)(i)) ⇒
        ∀ j:Nat • j ∈ inds obs_doc_trace(d) ∧ j<i ∧
```

```
            ∀ k:Nat • k ∈ {j,i−1} ∧ is_cD(obs_doc_trace(d)(k)) ⇒
               obs_E(obs_doc_trace(d)(i)) = obs_E(obs_doc_trace(d)(k))
```

Annotations:

- For all documents
- and for all indices, i, into the trace of such documents
- if the i′th document of that trace is an edited version
- then for all lower indices j, before i,
- if all documents (obs_doc_trace(d)(k)) of the trace properly j and i−1 are copies,
- then we can observe in these copies the same editing value. ■

7.3.4 Annotated Original Documents

We modify the copy function and the notion of an original document, od:oD. We now annotate original documents by a trace of "has been copied" markers. The (first original) document resulting from create() has an empty such trace. The document resulting from copy(create()) has a singleton trace of one "has been copied" marker. Each additional copying of a marked original adds one "has been copied" marker to the trace.

Two original documents which differ only in number of "has been copied" markers are otherwise considered the same original.

```
type
   hbc_Mark == hbc
value
   obs_hbc_Marks: oD → hbc_Mark*
axiom
   obs_hbc_Marks(create()) ≡ ⟨⟩ ∧
   ∀ od:oD • obs_hbc_Marks(copy(od)) = ⟨hbc⟩^obs_hbc_Marks(od)
value
   disregard_Marks: D → D
   disregard_Marks(d) as d′
      obs_hbc_Marks(d′) = ⟨⟩ ∧ obs_Pre(d) = obs_Pre(d′)
   differ_by_1_Mark: D × D → Bool
   differ_by_1_Mark(d,d′) ≡
      obs_hbc_Marks(d) = tl obs_hbc_Marks(d′) ∧
      disregard_Marks(d) = disregard_Marks(d′)
```

Annotations:

- hbc_Mark names a concrete type. Its only value is hbc. hbc is not further defined.
- obs_hbc_Marks is an observer function. It applies to original documents and yields a possibly empty list of hbc:hbc_Marker* of hbc markers.

- The list of hbc markers of a fresh, “virgin” original is empty.
- The list of hbc markers of any original that has been copied (once or more) has one more hbc marker than the original from which it was copied.
- We can view a document without its “has been copied” marks. That is the function of the disregard_Marks function.
- Two documents are, in a sense, the same even if they differ by one or more marks. ∎

We now redefine the predecessor observer function.

```
value
    obs_Pre: D $\overset{\sim}{\rightarrow}$ D
axiom
    obs_Pre(copy(d)) = d $\wedge$
    $\forall$ d:D,e:E •
        obs_Pre(edit(e,d)) = d $\wedge$
    $\forall$ od:oD •
        obs_hbc_Marks(od) = $\langle\rangle$ $\Rightarrow$ obs_Pre(od) = chaos $\wedge$
        /* the above is the same as */ obs_Pre(create()) = chaos $\wedge$
        obs_Pre(copy(od)) = od
```

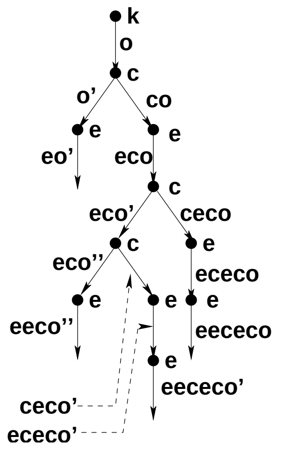
Later we shall augment the “has been copied” marker with location and time of copying.

7.3.5 Document Family Trees

Each document creation may give rise to a whole set of documents: copies of documents (for each copy a new document arises while the document from which it was copied basically remains), and edited versions of documents (for each version (i.e., editing) the number of documents remain the same). Given an original document one can establish what we shall call a family tree of documents descending from the originally created document — with each path from the tree root to a leaf of the family tree of documents denoting a document, that is, the number of leaves of a family tree of documents denotes the number of documents descending from the root original document.

By a tree we, in general, understand something which satisfies the following, conventional characterisation: (i) A tree has a root and zero, one or more sub-trees; (ii) a [sub-]tree with zero sub-trees consists of a node, the root, and a stem, the only branch of the [sub-]tree; (iii) a [sub-]tree with non-zero, i.e., $n \leq 1$ [sub[sub]-]trees consists of a node, the root, and n branches; the branches are either (iii′) stems denoting a latest instance of a document, or (iii″) whose non-root end denote proper sub-trees.

A document family tree arises as follows: (0) In the beginning there is nothing, i.e., no tree. A create() operation amounts to a tree being initialized. It has a (i) root (say labelled with the operation designation: create()); (ii) a leaf stem labelled with the original document, d_o, resulting from execution



o is original
o’ is o after copying o
co is copy of o
eo’ is edited version of o’
eco is edited version of co
eco’ is eco after copying eco
ceco is copy of eco
eco’ is eco after copying eco
ceco’ is copy of eco’
ececo is edited version of ceco
ececo’ is edited version of ceco’
eececo is edited version of ececo
eececo’ is edited version of ececo’

Fig. 7.1. A document family tree

of create()); and (iii) no sub-trees. Let us now take the general case. Let us call the leaf stem, i.e., document, to which an operation may be applied for d. Now there are either of three possibilities: (1) no operations are ever applied to d — and the tree does not grow from the stem labelled d; or (2) an edit operation edit(e,d) is applied to the leaf stem labelled d — where e is a suitable pair of editing functions: forward and backward, and the tree grows by the grafting of a “root” onto the “dangling” end of the input trees’ stem, labelled with edit(e,d), with a single leaf stem (emanating from that root) and labelled with the document, d_e resulting from the edit(e,d) operation; or (3) a copy operation copy(d) – and the tree grows by the grafting of a “root” onto the “dangling” end of the input trees’ stem labelled with edit(e,d) with two leaf stems (both emanating from that root), one labelled with the input document, d, the other labelled with a copy, d_c, of document d.

A document family tree thus consists of nodes and stems (i.e., branches). The root node designate the create() operation and its only branch denotes the original document, o, resulting from the create() operation. Nodes, other than the root node, designate editing or copying operations performed on documents. Stems other than the root stem designate documents d: originals, d_o, or copies, d_c, or edited versions, d_e, of either originals or versions.

A node, other than the root node, has one input stem and, for any node, one or two output stems. The input stem of a node is (also) said to be incident upon that node and to designate the predecessor document of the document resulting from the node operation. The output stem is — or the output stems are — said to emanate from the node.

If a node designates an edit operation then it has one output stem which designates the edited version of the document designated by the stem incident upon the edit node. If a node designates a copy operation then it has two output stems: one of these stems designate the (input, the master) document designated by the stem incident upon the copy node while the other stem designates the copy of that (input) document.

Finally a document family tree ends in leaves which are stems, i.e., documents. From any stem in a document tree one can establish the unique path of stems and nodes from that stem back to the original document designated by the stem emanating from the root node. Such a path is a document trace. As for the general, i.e., abstract concept of trees one can speak of subtrees. If a stem is incident upon a node, then that node is the root of a subtree which we shall here call a document tree (as distinguished from a document family tree). A (sub-)root of a document [family]tree[7] may have one or two subtrees, i.e., document trees: one of the (sub-)root designates the [create] (edit) [8] operation, two if it designates the copy operation.

```
type
  DFT′ = mkCreate() × oD × DT
  DFT = {|dft:DFT′ • wfDFT(dft)|}
  DT == nil | ET | CT
  ET = mkET(mkEdit(efns:(fe:FE,be:BE)),(ed:eD,dt:DT))
  CT = mkCT(mkCopy(),(d:D,dt:DT),(cd:cD,dt′:DT))
value
  wfDFT: DFT′ → Bool
  wfDFT(_,od,dt) ≡
    case dt of
      nil → true,
      _   → wfDT(dt)(od)
    end

  wfDT: DT → D → Bool
  wfDT(dt)(d) ≡
    case dt of
      nil → true,
      mkET((fe,be),(ed,dt′))
        → pre_post_Edit((fe,be),d,ed) ∧ wfDT(dt′)(ed),
      mkCT(mkCopy(),(d′,dt′),(cd,dt″))
        → pre_post_Copy(d,d′) ∧ wfDT(dt′)(d′) ∧ wfDT(dt″)(cd)
    end

  pre_post_Edit:  E × D × eD → Bool
  pre_post_Edit((fe,be),d,ed) ≡ ...
    /∗ see postcondition of the edit function on page 186 ∗/
```

[7]The phrase: *(sub-)root of a document [family]tree* reads as follows: *root of a document family tree or sub-root of a document tree.*

[8]The phrase: *(sub-)root designates the [create] (edit)* reads as follows: *root designates the create or sub-root designates the edit.*

pre_post_Copy: D × D → **Bool**
pre_post_Copy(d,d′) ≡ disregard_Marks(d′)=d

Annotations:

- DFT′ defines the Cartesian of not necessarily well-formed document tree.
- mkCreate(), oD and DT are the types of the components of the document tree.
- mkCreate() is strictly speaking not necessary, but is introduced so that all nodes possess an operation designator.
- oD designates the stem emanating from the mkCreate() node.
- DT designates the possibly empty sub-tree "attached" to the stem, i.e., upon which the stem may be incident.
- DT is thus either nil (i.e., the stem is a leaf) or is an edit tree et:ET or a copy tree ct:CT.
- An edit tree mkET(mkEdit(efns:(fe:FE,be:BE)),(ed:eD,dt:DT)) has sub-root node mkEdit(efns:(fe:FE,be:BE)) and one sub-tree (ed:eD,dt:DT).
  - ⋆ The sub-root node designates the editing functions mkEdit(efns:(fe:FE,be:BE)).
  - ⋆ The forward editing function fe "works" on the document of the stem incident upon this sub-root node.
  - ⋆ The backward editing function be "works" on the document of [the edited version stem ed:eD] emanating from this sub-root node.
  - ⋆ dt:DT designates a possible sub-tree of the stem emanating from this sub-root node.
- A copy tree mkCT(mkCopy(),(d:D,dt:DT),(cd:cD,dt′:DT)) has sub-root node mkCopy() and two sub-trees (d:D,dt:DT) and (cd:cD,dt′:DT).
  - ⋆ The sub-root node designates the copy function mkCopy().
  - ⋆ One (here shown as "the left") sub-tree (d:D,dt:DT) designates the document d:D being copied, hence "carried" forward, and its sub-tree dt:DT.
  - ⋆ One (here shown as "the right") sub-tree (cd:cD,dt′:DT) designates the document copy cd:cD, and its sub-tree dt′:DT.
- A number of constraints must be satisfied for a document history tree, dft, to be proper, i.e., to be well-formed wfDFT(dft).
  - ⋆ We can ignore the Cartesian mkCopy() component of dft.
  - ⋆ If the sub-tree component dt is nil then the whole document history tree is well-formed.
  - ⋆ Otherwise the well-formedness of dft is the well-formedness of dt in the context of the incident document od.
- The well-formedness wfDT(dt)(d) of a sub-tree dt in the context of an incident document d is likewise defined by cases:
  - ⋆ If dt is nil then well-formedness is guaranteed.
  - ⋆ If dt is an edit sub-tree mkET((fe,be),(ed,dt′)) then well-formedness is a conjunction of

  ◇ the edit pre/post condition pre_post_Edit((fe,be),d,ed) explained earlier, and
  ◇ the well-formedness of the version document sub-tree dt′.

⋆ If dt is a copy sub-tree mkCT(mkCopy(),(d′,dt′),(cd,dt″)) then well-formedness is a conjunction of
  ◇ the copy pre/post condition pre_post_Copy(d,d′) where d′ is the document being copied — and after copying,
  ◇ the well-formedness of the master[9] document sub-tree wfDT(dt′)(d′), and
  ◇ the well-formedness of the copied document sub-tree wfDT(dt″)(cd).

■

7.3.6 Document Family States

A state of a document family tree is a breadth-first set of stems of the tree. A breadth-first set of stems of a document family tree is one whose stems belong to distinct paths. Fig. 7.2 shows 11 states of a document family tree.

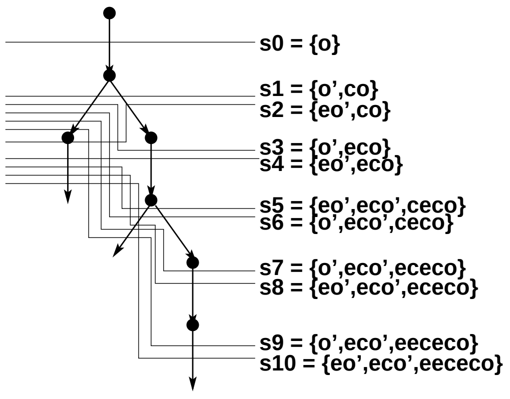


Fig. 7.2. Document family states

The idea is that there is an initial state, here s0, of the tree, and that there is a final state, here s10, of the tree. The initial state, here s0, designates the initial, i.e., the original document. The final or current state, here s10, designates a notion of final (or current) documents. A final state means that no further operations are to be performed on members of a set of documents. ("Case closed.") A current state means that we are observing the document family tree "right now". Please note that the initial, current and final states of any document family tree are unique. Please also note that a void document,

[9]We shall move this notion way back, towards the front of this chapter: the master document is the document being copied.

i.e., a copy of a copy of ... a copy of an original document may be a current or final document.[10] Intermediate states designate possible collections of non-final documents. Thus a non-final state has one or more successor states. Usually there may be several ways of making state transitions from the initial state to the final state. Possible sequences of states are indicated by:

s0 $\mapsto$ s1 $\mapsto$ s3 $\mapsto$ s6 $\mapsto$ s7 $\mapsto$ s9,
s0 $\mapsto$ s1 $\mapsto$ s2 $\mapsto$ s4 $\mapsto$ s8 $\mapsto$ s10.

From a document family tree we can compute all states and all possible initial to final state sequences.

```
type
    $\Sigma = \{|\delta\sigma$:D-**set**•$\delta\sigma \neq \{\}|\}$
value
    States: DFT $\rightarrow$ $\Sigma$-**set**
    Traversal: DFT $\rightarrow$ $\Sigma^*$
```

7.3.7 Document Community

By a document community we mean a set of uniquely identified document family trees.

```
type
    Did
    DoCo = Did $\overrightarrow{m}$ DFT
```

No two states of (two) distinctly named document family trees share states, i.e., have one or more documents in common.

```
value
    wfDoCo: DoCo $\rightarrow$ **Bool**
    wfDoCo(doco) $\equiv$
        $\forall$ did,did′:Did • {did,did′}$\subseteq$**dom** doco $\wedge$ did$\neq$did′ $\Rightarrow$
            disjoint_docs(doco(did),doco(did′))

    disjoint: DFT $\times$ DFT $\rightarrow$ **Bool**
    disjoint_docs((_,od,dt),(_,od′,dt′)) $\equiv$
        {od}$\cup$ extract_docs(dt) $\cap$ {od′}$\cup$ extract_docs(dt′) = {}

    extract_docs: DT $\rightarrow$ D-**set**
```

[10]The reader may feel uncomfortable having such void copies “floating” around, seemingly to no effect. But that is the cost of not imposing constraints that would otherwise impose what we consider unnatural limitations on what can be done to documents.

```
extract_docs(dt) ≡
    case dt of
        nil → {},
        mkET((fe,_),(d,dt′)) →
            {fe(d)}∪ extract_docs(dt′),
        mkCP(_,(md,dt′),(dc,dt″)) →
            {md,dc}∪ extract_docs(dt′)∪ extract_docs(dt″)
    end
```

7.3.8 Document Processing States

Instead of only include "stable" documents, that is, documents not, at the moment being subject to operations, that is, stems in a state concept, we could also include the operation "states", that is, nodes in a state concept. We call this state notion a document processing state. Figure 7.3 shows some "early" document processing states for the same document family tree as in Fig. 7.2 on page 193.

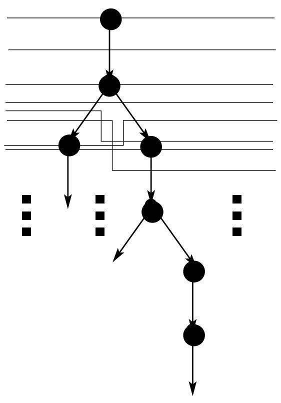


Fig. 7.3. Document processing states — hinted at by horisontal (traversal) lines

7.3.9 Shortcomings of Model So Far

There seems to be a number of problems with the model so far: Documents — whether manifest by humans senses (such as paper documents) or by technical/scientific apparata (such a MS Word, LaTeX (.tex) files, portable document format (.pdf) files or postcript (.ps)files) — always have a unique location in space. Some operations have not been mentioned or modelled: moving a document (fomr one location to another) and shredding. Operations on documents occur at certain times and these operations may, or may not "take time to perform". Finally we did not mention any notion of document identity: two documents which differ in some way (location, time of application of, say, most recent operation, content, etc.) can be claimed to have unique, i.e., distinct indentities. We will, accordingly, in the next two sections propose concrete models of locations and time of operation invocation.

7.3.10 A Basic Concrete Model of Time

An Axiom System for Time

We refer to Appendix Sect. F.1 on page 415 for an axiom syste of time.

We make a distiction between time and time intervals. Time is considered absolute time and is with respect to some initial, i.e., 0 time. Time intervals are the same as elapsed time, that is, the difference between two times. Both time and elapsed time are expressed in hours, minutes and seconds, or just in seconds.

And we make a syntactic distinct between date and time. Both are absolute. Dates change every 24 hours, so dates are less fine-grained than time. And dates (time) can be converted into time (dates). A dates to time conversion results in a measure which is modulo 24 hours. Vice versa: a time to date conversion results in a measure which is also modulo 24 hours and thus looses "prcision".

Mathematically we consider time to be a linear dense point ordering. Each document operation: create, copy, edit and move occurs at a specific time (and lasts no time). One can extend to model to include that some operations take time. Such a model would have some notion of a document (document) being inaccessible to other than the operator during the operation time interval.

From documents we can now observe the time of their last operation. When creating, copying, editing and moving a document a single time is provided (below referred to as 'given time'). The original document being created "receives" the given time. The document copy being established likewise "receives" the given time. The (master) document from which the copy was made retains its time.[11] The document resulting from an edit "receives" the given time. A document move shall result in the moved document being marked with the given time. We shall, when now considering the create, copy, edit and move operations not consider whether the implied times are coincident with times of other documents of other the same family or other families. Predecessor documents of any documents retain their times of operation applications.

[11] One could change the model to have the master document that is "passed on" also be ascribed the given time — the "has been copied" marker can also be used to embody the copying time.

```
type
    T
value
    obs_T: D → T
    create: T → oD
    copy: D × T → D × cD
    edit: E × D × T → eD
    move: D × T ∼→ D
axiom
    ∀ t:T,e:E •
        obs_T(create(t)) = t ∧
        let (d′,cd) = copy(d,t) in pre_post_Copy′(d,d′)∧obs_T(cd)=t end ∧
        obs_T(edit(e,d,t))=t ∧
        obs_T(move(d,t)) = t
```

Annotations:

- The pre_post_Copy′ is a version of pre_post_Copy which accepts documents with time attributes. ■

7.3.11 A Concrete Model of Locations

We introduce a spatial notion of location. Mathematically we consider a location to be a dense point set equipped with some "neighbourhood" (or "infinitisimally close" predicate). No two otherwise distinct documents can occupy overlapping locations. Thus all distinct documents of a document family state occupy distinct, non-overlapping locations. And similarly for document communities.

From documents we can now observe their location. When creating or copying a document a single location is provided. The original document being created "receives" the given location. The document copy being established likewise "receives" the given location. The document from which the copy was made retains its location. The document resulting from an edit retains the location of the document being edited. We finally add a new operation on documents: Moving a document from one location to another, therefrom distinct location. The move shall result in the location of the moved document changing from what it was before the move to the given location. We shall, when now considering the create, copy, edit and move operations not consider whether the implied locations may interfere with locations of other documents of a family or community.

```
type
    L
value
    =: L × L → Bool
    infinitesimally_close: L × L → Bool
axiom
    ∀ l,l′:L • infinitisimally_close(l,l′) ⇒ l≠l′
    ∀ l,l′,l″:L • l′≠l″ ∧
        infinitesimally_close(l,l′)∧infinitisimally_close(l,l″) ⇒
            infinitesimally_close(l′,l″) ...
value
    obs_L: D → L
    create: T × L → oD
    copy: D × T × L → D × cD
    edit: E × D × T × L → eD
    move: D × T × L ~→ D
axiom
    ∀ t:T,l:L,d:D,e:E •
        ∀ t:T,l:L • obs_L(create(t,l))=l ∧
        ∀ d:D,t:T,l:L •
            let dc = copy(d,t,l) in infinitesimally_close(obs_L(dc),l) ∧
            pre_post_Copy′(d,dc)∧obs_L(d)=l end ∧
        ∀ d:D,t:T,l:L • obs_L(edit(d,t,l))=l ∧
        ∀ d:D,t:T,l:L • obs_L(move(d,t,l))≠l
```

7.3.12 Located and Timed Documents

An Axiom System for Time/Space

We refer to Appendix Sect. F.2 on page 416 for an axiom system of time/space.

We wish to record that for every document that has been copied the fact that it has been copied: time and location.

```
value
    has_been_copied: D → Bool
    when_where_copied: D ~→ L×T
    otherwise_the_same: D×D → Bool
axiom
    ∀ d:D •
        ∼has_been_copied(d) ≡ when_where_copied(d)=chaos ∧
        ∀ l:L,t:T,e:E •
            let (d′,cd) = copy(d,l,t) in
            has_been_copied(d′) ∧ when_where_copied(d′)=(l,t)
            otherwise_the_same(d,d′) end ...
```

Annotations:

- Axioms for the has_been_copied predicate is only sketched. ∎

No document can at the same time be in two different locations:

```
axiom
  ∀ d,d′:D •
      let (l,t) = (obs_L(d),obs_T(d)),(l′,t′) = (obs_L(d′),obs_T(d′)) in
      (t=t′ ∧ l=l′ ≡ d=d′) ∧ (t=t′ ∧ l≠l′ ≡ d≠d′) end
```

7.4 Discussion

We have brought a rough sketch domain analysis of some aspects of the 'Document' concept. In the next chapter we bring a similar rough sketch domain analysis of some aspects of the 'Public Administration' concept. We emphasise that these, i.e., Chaps. 7 and 8 are just sketches. When we bring these two sketches together with the Chap. 10 "story" on 'Scripts, Licenses and Contracts' then we can see a larger perspective: one that combines the three sets of basically distinct issues of Chaps. 7–8 and Chap. 10 with further issues of document 'Authorisation and Security' — such as the latter is for example strongly hinted at in Chap. 9. To do this four way combination (Chaps. 7–10) amounts to a PhD thesis topic and work or a proper industry project.

8

Public Government[1]

A Rough Sketch Domain Analysis

Abstract

In this report we investigate a concept of 'public government' through the concept of 'document'. Public government is seen as consisting of the usual three branches: the law-making, the law-enforcing and the law-interpreting government (the latter also known as the 'judiciary'). Common to all these are the citizens (and private companies) with which the law, the daily administrative, and the law-decision documents deal. We focus on document handling: the creation, editing, reading, copying, distribution, shredding, and tracing of documents. We close by examining the issue of turning this abstract, the general domain view, into requirements for the electronic handling of documents — and hence we conclude by making some statements about essential E–government. This chapter is a torso. On one hand, it is intended as a basis for PhD student work, and, on the other hand it lacks a proper comparison to possibly existing literature.

8.1 An Informal View of Public Government

Public government, in this chapter, consists of the *lawmakers:* parliament; the *law enforcers:* central and local government; and the *law interpreters:* judiciary system. *Citizens* interact with all three branches of government.

The [A] part of Fig. 8.1 on the next page indicates the three branches of government.

Parenthesised numerals refer to the [B-C] parts of Fig. 8.1 on the following page.

Citizens (1), through the process of debate, provoke on their parliament to discuss societal problems. A parliament committee (2) discusses a specific societal problem. Their deliberations are "sent" as a law proposal to the parliament (3) which debates the issue and passes some law.

[1]This is an edited version of [30].

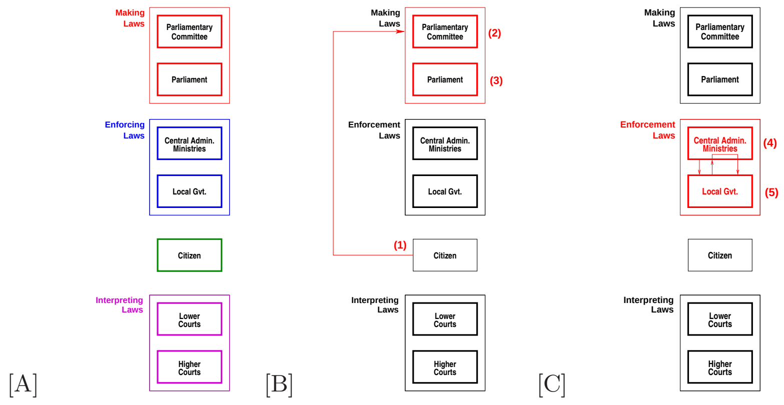


Fig. 8.1. [A] The three branches.
[B] From citizens to lawmakers.
[C] Lawmaking.

The law is passed on to an appropriate ministry (of the central government, (4) and that ministry formulates basic rules & regulations for how local governments shall administrate uses of the law. The local governments (5) makes provisions for handling the law locally.

Parenthesised numerals now refer to Fig. 8.2 on the next page.

The citizen is either contacted by the local government (6) and asked to report on some issue (tax, traffic violation, or other), and the citizen replies (6). or the citizen contacts (7) the local government in order to apply for something (passport, pension benefits, and other) And the local government replies (7).

The citizen is either accepts the decision of local government, or the citizen does not accept the decision, and complains to the courts (9). The "due process of law" takes place (11). Eventually the judiciary system hands down a decision either in favour of the citizen, or in favour of the government, or both (11) !

8.2 Flow of Documents in Public Administration

8.2.1 Between Citizens and Lawmakers

Citizens may direct (12) a problem petition to parliament — in the form of a *document* signed by many citizens. Parliament (13) decides to "do something" (or not to do anything) about the problem. A response *document* is produced. A designated parliament committee (14) requests an appropriate ministry to prepare some "background" *document*. The parliament committee (15) passes discussion and a law proposal *documents* to parliament.

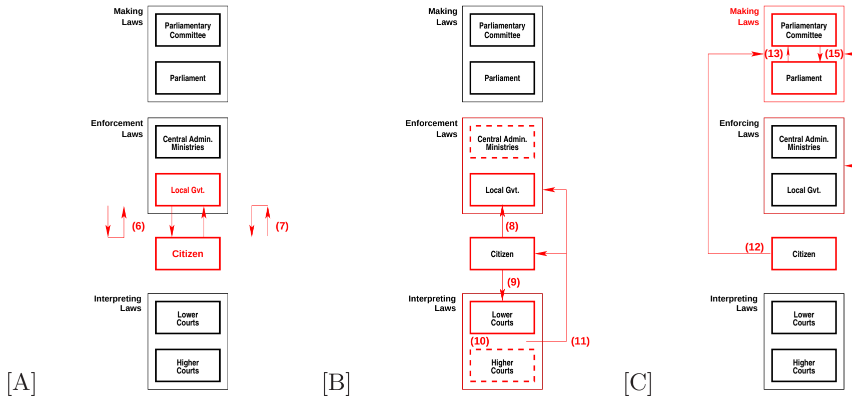


Fig. 8.2. Document Flows: [A] Citizens and local government.
[B] Citizens and the judiciary.
[C] Citizens and parliament.

8.2.2 Between Lawmakers and Ministries and Local Government

Parenthesised numerals now refer to Fig. 8.3 on the following page.

Parliament requests (16) further "background" *documents* from the central administration, and receives these. Parliament debates the law proposal and passes a law, which, as a *document* (17) is sent to the appropriate ministry for further handling — and otherwise published in the law gazette. The ministry and its departments, i.e., the central administration, formulates procedures for the enforcement of the law and sends (18) these, as *documents* to local administrations.

8.2.3 Between Citizens and Local Government

A citizen (19) applies for some permission, that is, an application *document* is sent to a local administration — or a citizen breaks the law (symbolised with the virtual arrow from citizen to a local authority). The local administration (20) sends a receipt (a citation) *document*, possibly forwards further *documents* to be filled in, and gives a conditional date by which a decision can be expected. The citizen (21) sends in the possibly further requested *documents*. The local administration (22) communicates various *documents* related to the case to/from other public government offices. Finally the citizen (23) receives a response *document*.

8.2.4 Between Citizens and The Judiciary

The citizen either (23a) sends an acceptance *document* to the local administration, or (23a) rejects it, informing the local administration of this, and directs

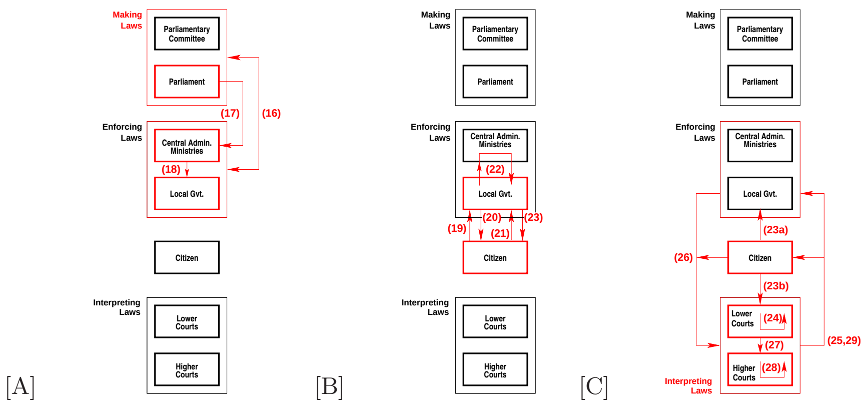
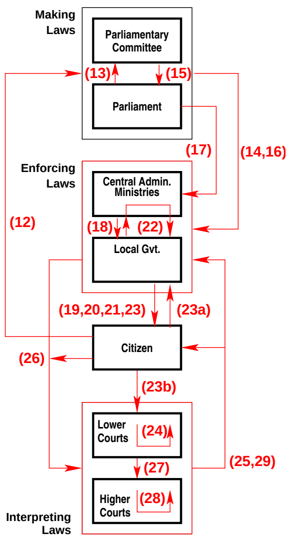


Fig. 8.3. Document Flows: [A] Parliament and admin.
[B] Citizen and local government.
[C] Citizen and judiciary

(23b) a *complaint* at the law courts. The first instance law court (24) deliberates (i.e., *documents* are produced), a *decision* is sent (25) to the citizen and the local administration, Either the local administration or the citizen both accepts the decision and further actions are curtailed, or at least one of them (26) appeals the decision. Lower court decision documents are passed (27) on to a higher court. Steps (24–25) are repeated (28–29) till a final decision is passed (29).

8.2.5 Summary of Documents

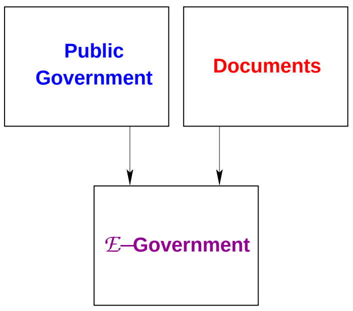
12. Citizen petition
13. Parliament response
14. Background briefing
15. Subcommittee discussion
    and law proposal
16. Further background briefing
17. Law and record of parliament debate
18. Handling procedures and forms
19. Citizen application
    (or “breaking law event”)
20. Local authority reply
    (or citation)
21. Citizen response to local authority
22. Public admin. docs. handling
    citizen request or infringement
23. “Final” local authority reply/decision
    (a) Citizen acceptance or rejection
    (b) Citizen lawsuit
24. Lower court handling
25. Lower court decision
26. Citizen reaction and/or
    Public administration reaction
27. Transfer of lower court documents
28. Higher court handling
29. Higher court decision (final)



8.3 Documents — A Closer Analysis

We present a “story” of documents that is a bit different from what you may be used to. The reason is that we are building up, towards the end of this chapter, to a “story” on computers, communication and documents. Our “story” on documents is also “completely” independent of our previous “narration” of public government. Again there is a reason: towards the end of this chapter we merge the “story” on documents with the “narration” of public government into the subject of $\mathcal{E}^2\mathcal{G}$: $\mathcal{E}$ssential $\mathcal{E}$-$\mathcal{G}$overnment

[A]



[B]

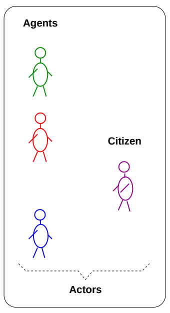


Fig. 8.4. [A] From public government to documents — and beyond.
[B] Actors: citizens and three kinds of agents.

8.3.1 Overview of Document Issues

We summarise a number of document issues in “SMS” form:

Document Operations: Documents are created, are edited, and can be read, copied, moved, can be the basis for searches and calculations, and can be shredded — by actors. *Document Authorisation:* Actors have varying degrees of creation, editing, reading, copying, distribution and shredding authority. *Document History:* Document may reveal their creation, editing, reading, copying distribution, search & calculation history — who did what, when, where! *Document Licensing:* Unauthorised actors must be prevented from performing designated operations on documents. We achieve this indirectly by introducing and enforcing a regime of licensing based on authorisation.

8.3.2 Actors: Citizens and Agents

Document operations are performed by *actors*. *Actors* are either *agents* or *citizens*, An *agent* is either a person working for a branch of public government, or is that branch of public government. Of course *citizens* may be agents in some other context, that is, a *citizen* is a person interacting, not as an agent, with agents of public government.

8.3.3 Document Operations

Document operations are performed by *actors*. Documents can be *created*. Documents can be *edited*. Documents can be *read!* Documents can be *copied*. Documents can be *moved*. Documents can be *shredded* (not shown). An *actor* can at most be performing one operation at a time. Documents are "marked" by the *time* and *location* of the operation and by the identity of the performing *actors*. Together the *time*, *location* and *actor identity* forms a *unique document identification*.

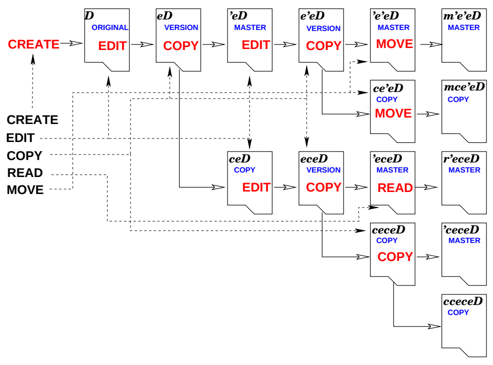


Fig. 8.5. Five document operations

We overview five of the operations.

Documents can be *created*. Created documents contain no substantial information other than administrative information about time and location of creation, identity of actor who created the document.

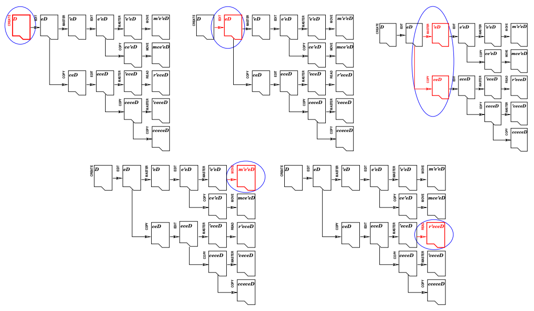


Fig. 8.6. Five document operations

Documents can be *edited.* Edited documents are *versions* of the document on the basis of which they were edited. Edited documents contain substantial information. One can read documents while editing them! One cannot copy, move or shred documents during editing. An edited document is different from the document input. The difference amounts to the changed text, time and location of edit, and identity of editing actor.

Documents can be *copied.* As a result the *master* "goes on" to exist and a copied, *"the copy"*, document is constructed. One cannot edit, move, read or shred the master or the copy documents while copying. The location and time of copying is the same for both master and copy. To *audially communicate,* i.e., to *tell* (speak about) a document (content) to other listeners is the same as *copy*ing it.

Documents can be *moved.* They are physically moved from one *location* to another distinct *location.* The document "is almost" the same before start and after end of move, only location has changed. Documents cannot be copied, edited, read or shredded while being moved. Instead of *moved* we shall sometimes use the term *distributed.*

Documents can be *read*! One cannot edit, copy, move or shred document while only reading them. Reading leaves the document unchanged — except that it has now been read, at some time and at some location, by some actor.

Figure 8.7 shows a sequence of document operations: 1: create, 2: edit, 3: copy, 4: edit, 5: copy, 6: edit, 7: copy, and 8: read.

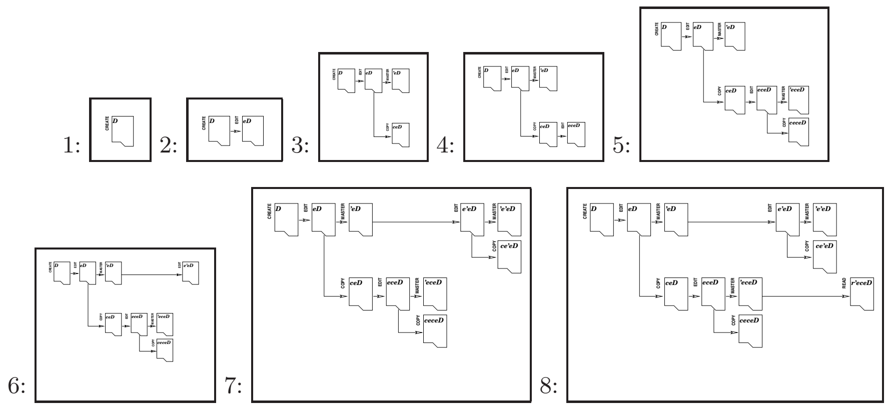


Fig. 8.7. A sequence of document operations

8.3.4 Document Family and Document Versions

Document Family: (Fig. 8.8, left) The structure to the left designates one document family. Every create gives rise to a document family. It may grow (copying) and shrink (shredding).

Document Version (Fig. 8.8, right) A "just" create document has version 0. Every edit of a document creates a new version of that "same" document. All other operations leave the version attribute unchanged. (Let us not be bothered by how version numbers are generated!)

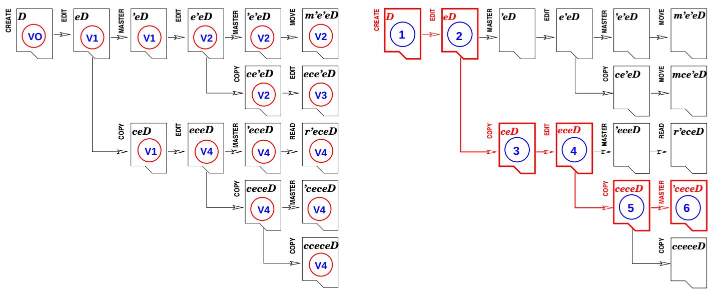


Fig. 8.8. Document family and document versions

8.3.5 Document History

From a document we can, "in theory", *trace its unique past.* In reverse order of operations: (6) Take the lower right MASTER. (5) It was most recently the BASIS for a *copy*ing. (4) Before that it was the BASIS for an earlier *copy*ing. (3) Before that it was the RESULT of a *copy*ing. (2) Before that it was an *edit*ed version (1) of a *create*d document.

See Fig. 8.9 on the facing page.

(1) *Create:* An actor named *Nm1* performs operation *create* at time *time1* on location *loc1*. A document is created with just about the only information you see in the lower left corner to the right.

(2) *Edit:* An actor named *Nm2* performs operation *edit* at time *time2* on location *loc1* Document *D* is extended with text *text* into document *eD*.

(3) *Copy:* An actor named *Nm3* performs operation *copy* (on *eD* and we focus on the 'copy') at time *time3* on location *loc1*. Document *eD* text *text* remains unchanged but is referred to as *ceD*.

(4) *Edit:* An actor named *Nm4* performs operation *edit* at time *time4* on location *loc1*. Document *ceD* text *text* is changed into document *eceD* *text*′.

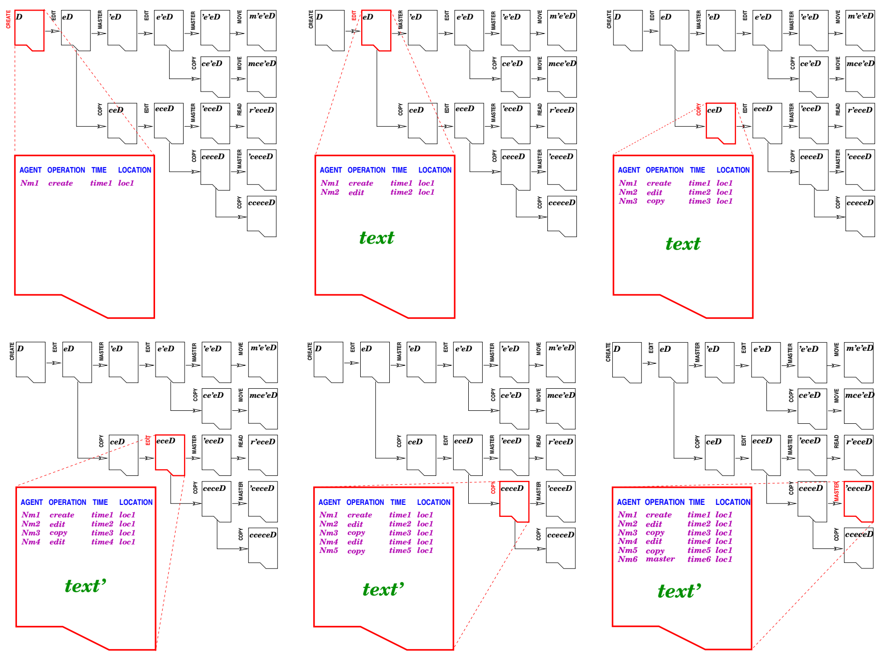


Fig. 8.9. Document annotation

(5) *Copy:* An actor named *Nm5* performs operation *copy* (on *eceD* and we focus on the 'copy' *ceceD*) at time *time5* on location *loc1*. Document *ceceD* text *text*′ remains unchanged.

(6) *Master:* An actor named *Nm6* performs operation *copy* (on *ceceD* and we focus on the 'master' ′*ceceD*) at time *time6* on location *loc1* Document ′*ceceD* text *text*′ remains unchanged.

8.3.6 Document Authorisation

Rationale for Authorisation

We explain the need for introducing a concept that we shall call 'authorisation'. Some documents may contain information that not all *actor*s should be aware off. For an *actor* to be allowed to perform an operation upon (create or edit or copy or read or move or shred) a document, that *actor* must be so authorised (by some *agent*). *Authorisation* can be in the form of a *license*, a *permit*, to perform an *operation* on some entity, as here, a *document*.

Authorisation of Actors

Actors can be authorised with respect to (wrt.) one specific document whose identity will then be given, or wrt. a finite, identified set of documents, or wrt. a potentially indefinite class of documents, so we have to introduce a notion of a document class. (An example class of documents could be the class of all social security application forms [based on same template]. More on this later.)

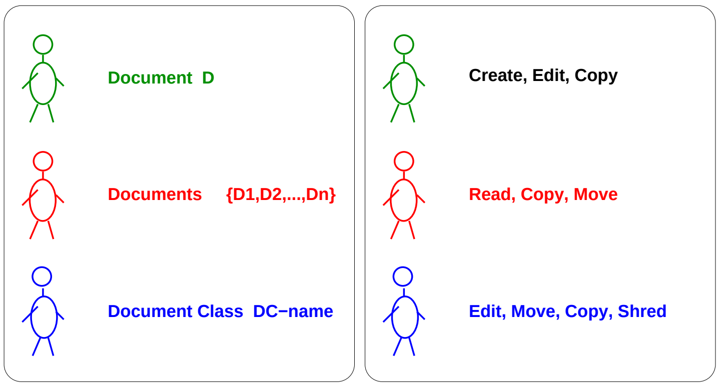


Fig. 8.10. Document and operation authorisations

Actors can, within the designated set of such documents, be authorised wrt. which operations can be performed on these documents: *create, edit, copy, search, move, read, shred and compute.*

Actors may be authorised to *grant* document handling authorisations to (other) actors, *extend* or *limit* previously (by others or by that same actor) granted such document handling authorisations, or outright *withdraw* such document handling authorisations altogether.

The problem of how to initialise the system of document handling authorisations is an interesting one — which we may have time to come back to later.

License Scripts

A license script is a named text which is issued by one actor to another. The license specifies a grant, an extension, a limitation or a withdrawal of specified document operations on specified (licensed) documents. A license can also grant permission to issue further licenses.

Example Licenses

ln1: **actor** a1 **grants operations** {op1,op2,...,opn}

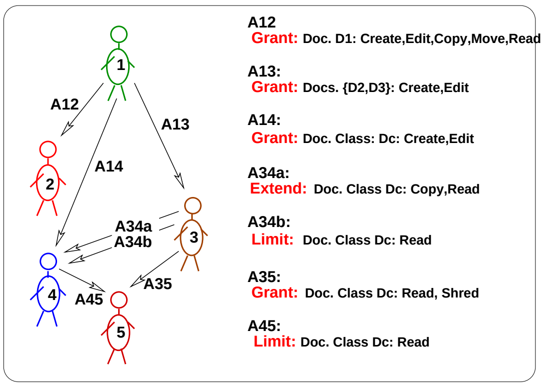

1
2
3
4
5
A12
A13
A14
A34a
A34b
A35
A45
A12
Grant: Doc. D1: Create,Edit,Copy,Move,Read
A13:
Grant: Docs. {D2,D3}: Create,Edit
A14:
Grant: Doc. Class: Dc: Create,Edit
A34a:
Extend: Doc. Class Dc: Copy,Read
A34b:
Limit: Doc. Class Dc: Read
A35:
Grant: Doc. Class Dc: Read, Shred
A45:
Limit: Doc. Class Dc: Read

Fig. 8.11. Granting, Extending and Limiting Licences

to actor a2 **on documents** {d1,d2,...,dm}

actor a3 **extend license** ln2 **of actor** a4 **with operations** {op′,op″,...,op‴}

actor a5 **limit license** ln3 **of actor** a6 **with operations** {op$a$,op$b$,...,op$w$}

actor a7 **withdraw license** ln4 **from actor** a8

actor a9 **grants licensing right**
ln5: **actor** a10 **grants operations** {opx,opy,...,opz}
to actor a11 **on documents** {da,db,...,dc}

actor a12 **withdraw license** ln6 **from actor** a13

8.3.7 Special Edit Operations

We have hinted at only a very rudimentary *edit* operation. One that takes only one document and basically adds, modifies or deletes *text*. More general, in fact a whole family of *edit* operations are present in everyday handling of documents: *merge* of two documents into a third, implying a *copy*ing and a *create* operation. into one of the two documents, implying a *copy*ing and a simple *edit* operation. *split* of one document into two documents implying a *copy*ing and two *create* operations. $\mathcal{E}$*tcetera*

8.3.8 A Document Class Concept

Examples

General Public Administration Document Classes (I)

Different branches of government work on and produce different classes of document: Parliamentary committees handle *societal background problem* documents and produce *committee discussion* and *law proposal* documents.

General Public Administration Document Classes (II)

Parliament handles *committee discussion* and law proposal documents and produces *parliament discussion* and law documents. ... And law courts receive *law suit* documents, deliberate over *law court proceeding* documents and issue *verdicts* (i.e., documents).

Specific Public Administration Documents

Taxation Document Classes: A tax office issues *tax declaration template* documents and handles *tax declaration form*[2] documents.

Traffic Police Documents: A traffic police officer handles *traffic violation citation template* documents and issues *traffic violation citation forms* (i.e., documents).

Budget and Account Documents: An accountant handles *budget* documents, fills in *account template* and *account form* documents, and aggregates (calculate) over *account form* and *budget* documents to produce budgets (forms).

Document Classes

Generic Document Classes

By the generic class of documents we understand a class of documents that is independent of the specific application domain. We make the following generic classification: General, unformatted, un-structured text documents. *Comment*: No computations can be done on such documents. *Example*: Any "spur-of-the-moment" note. Formatted, semi-structured text documents. *Comment*: Trivial searches can be done over such documents. *Example*: Most public administration documents. Specifically formatted, template-based document. *Comment*: Non-trivial computations can be done over such documents. *Example*: Most application and related documents.

[2]A form is a filled-in template document

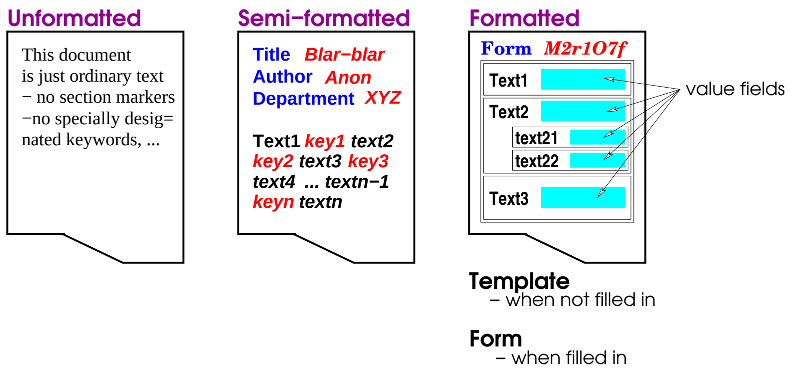


Fig. 8.12. Indication of Document Classes

Specific Government Document Classes

For the application-specific category of public administration we suggest to let the class of government documents be determined by their relevance to specific laws or law proposals:[3] Some specific examples: (i) law proposal background document; (ii) law proposal discussion document; (iii) law document; (iv) law admin./handling document; (v) law rules & regulation document; (vi) local admin. letter to citizen; (vii) citizen reply to local administration letter; (viii) citizen complaint to law court; (ix) law court inquiry document; and (x) law court decision.

8.3.9 Document [Cross–]References

Documents in one family of documents may cross-refer to documents in another family of documents as shown in Fig. 8.13 on the next page. Reasons seem obvious: chronologically "later" documents were derived in the context of knowledge of the chronologically "earlier" documents which can be made aware of the of the "later" uses.

Examples of Document [Cross–]References:

(a) Law proposal background doc. *may refer to other laws.* (b) Law proposal discussion doc. *may refer to background doc.* (c) Law *usually refers to many other laws.* (d) Law admin./handling doc. *usually refers to many other handling docs. and refers to the law.* (e) Law rules & reg. doc. *refers to the law.* (f) Local admin. letter to citizen *refers to the law.* (g) Citizen reply to local admin. letter *refers to that letter.* (h) Citizen complaint to law court *refers to*

[3]— all other documents are simple administration documents found also in most other forms of administration.

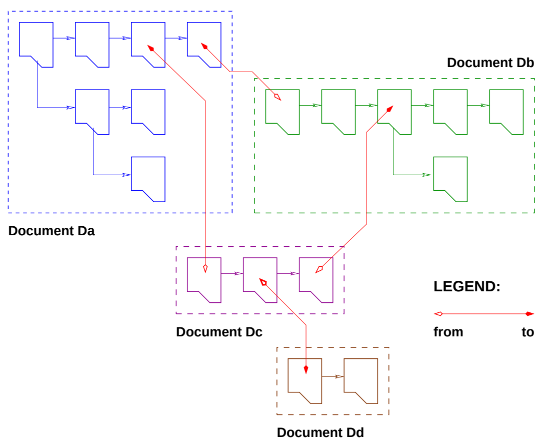


Fig. 8.13. Cross-references between documents of four document families

local admin. letter and law. (i) Law court inquiry doc. *refers to citizen and local admin. letters and the law.* (j) Law court decision *refers to the law and other law court decisions.*

8.3.10 Document Computations

We can distinguish amongst different kinds of computations over documents.

Document Identifier Searches: *Example*: Intra- and inter–document-family tracing.

Attribute Searches: *Example*: Searches based on document attributes — incl. form numbers.

Trivial Document "Syntactic Content" Searches: *Example*: Documents related to a specific agent or topic (a citizen, a law, or other). Searches look for text parts only.

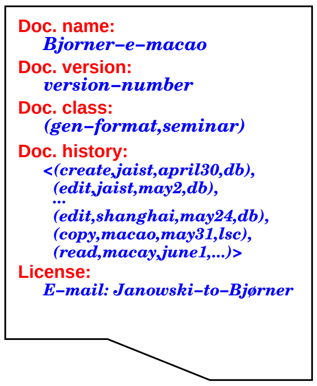
Non-trivial Document "Semantic Content" Searches: *Example*: Documents containing strongly formatted text parts, e.g., XML embedded and coded texts. Searches try to deduce meaning ("data mining").

Full-blown Computations: *Example*: Tax computations.

8.3.11 Summary of Document Attributes

These are the generic classes of document attributes: (1) *document name*, (2) *document version number*, (3) *document classes* (generic and special), (4) *history trace* of operations[4] (operation, location, time, actor) and (5) *reference to relevant license. A unique document identifier can be calculated from the above.*

[4]including computations



Doc. name:
Bjorner–e–macao
Doc. version:
version–number
Doc. class:
(gen–format,seminar)
Doc. history:
<(create,jaist,april30,db),
(edit,jaist,may2,db),
...
(edit,shanghai,may24,db),
(copy,macao,may31,lsc),
(read,macay,june1,...)>
License:
E–mail: Janowski–to–Bjørner

Fig. 8.14. Document attributes

8.3.12 Actor Attributes

(1) *Unique Actor Identifier*, (2) *Actor Location*, (3) *Licenses*: (3.1) Current "Own" Licenses (i.e., work being pursued) and (3.2) Current Granted Licenses (i.e., work being managed), (4) Which *Documents* (by reference): (4.1) have be *Operated* upon, (4.2) at *Locations* and (4.3) and at *Times*.

8.4 $\mathcal{E}^2\mathcal{G}$: $\mathcal{E}$ssential $\mathcal{E}$-$\mathcal{G}$overnment

Law-based public administration handles, like any other public or private administration, "thousands" of kinds of "zillions" of documents. Of those we single out now, we focus only on those public government documents which are *based in law*, that is, which ultimately *refer to laws*. That is we *do not* in the following consider such kinds of documents as procurement, (ordinary) budget, accounting, personnel, etc., documents whose handling is like in any other, not law-based administration, whether public or private.[5]

8.4.1 The Meaning of $\mathcal{E}^2\mathcal{G}$

So, by $\mathcal{E}$ssential $\mathcal{E}$-$\mathcal{G}$overnment we shall thus mean: Any electronic handling of documents, that is, *planning, discussion, decision, preparation, communication, etcetera* which are based on and results in (possibly new, possibly edited) documents *such that these documents directly or indirectly refer to laws.*

[5]Of course, financial reporting is subject to laws, so, perhaps we should not have been so definite in our *"do not"* statement.

From Domain to $\mathcal{E}^2\mathcal{G}$ Requirements

To illustrate what "such electronic handling" might mean, we *systematically go through* the domain(s) of public government such as outlined in Sect. 8.2 and of documents, such as outlined in Sect. 8.3, in order to *decide* whether "such–and–such" agent (institution, i.e., branch of government) and citizen interactions, documents and document attributes, functionalities, "etcetera" *must* be *"made more–or–less electronic"*.

So Here We Go: Towards $\mathcal{E}^2\mathcal{G}$

Which branches of government should be included in $\mathcal{E}^2\mathcal{G}$?

One answer could be: All branches interfacing with citizens. The double–arrowed line indicates this. Within these branches we must list the relevant departments. Another answer could be: Only such and such a subset of the indicated branches. (We then omit some of the $\leftrightarrow$s.) See Fig. 8.15.

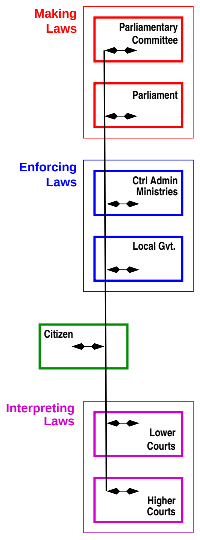


Fig. 8.15. Branches of government included in $\mathcal{E}^2\mathcal{G}$

Which document classes should be included in $\mathcal{E}^2\mathcal{G}$?

One answer could be: All the (in this example) *19* document classes implied by the *red* arrows in figure to the right. For each arrow-head we then list relevant document classes. Another answer could be: Only such and such a subset of the indicated document classes. (That is, we omit some of the $\rightarrow$ document classes.) See Fig. 8.16 on the next page.

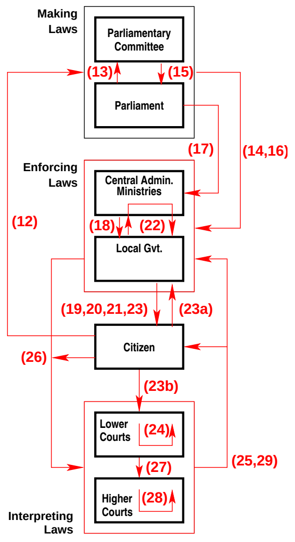


Fig. 8.16. Document classes included in $\mathcal{E}^2\mathcal{G}$

Which authorisation on document classes should be $\mathcal{E}^2\mathcal{G}$ scripted?

For each selected document class scripts of zero, one or more authorisations need be designed and their instantiation wrt. specific documents must be computerised, as must the monitoring and control of the implied document family. See Fig. 8.17.

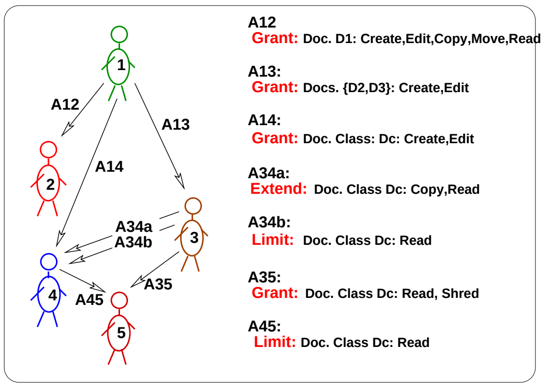

1
A12
A13
2
A14
A34a
A34b
3
4
A45
A35
5
A12
Grant: Doc. D1: Create,Edit,Copy,Move,Read
A13:
Grant: Docs. {D2,D3}: Create,Edit
A14:
Grant: Doc. Class: Dc: Create,Edit
A34a:
Extend: Doc. Class Dc: Copy,Read
A34b:
Limit: Doc. Class Dc: Read
A35:
Grant: Doc. Class Dc: Read, Shred
A45:
Limit: Doc. Class Dc: Read

Fig. 8.17. Document authorisations to be $\mathcal{E}^2\mathcal{G}$ scripted

Which new operations should be $\mathcal{E}^2\mathcal{G}$ supported?

For each selected document class as well as combinations of related document classes all conceivably $\mathcal{E}^2\mathcal{G}$ supported operations must be defined: All relevant *searches*. All relevant $n \geq 1$ *computations*. All *trace* functions. A new *release* operation. The *shred* operation. See Fig. 8.18.

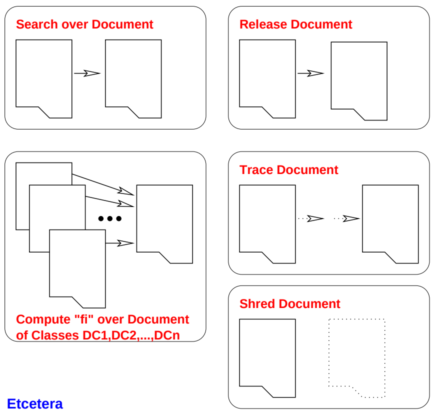


Fig. 8.18. New operations to be $\mathcal{E}^2\mathcal{G}$ supported

More on $\mathcal{E}^2\mathcal{G}$ Document Computations (II): One class of computations have the computation be based on *values* of a possibly only partially *filled-in template*, i.e., *form*. The computation would then yield, not necessarily a document, but new values that can serve as basis for further computations and possibly be fed to a repository of documents. Such computations presume design of templates, i.e., forms, and the implementation of the computations. *Yet More on $\mathcal{E}^2\mathcal{G}$ Document Computations (III):* Another class of computations collects data from several possibly distinct documents and produce an aggregated document. An example could be income and property tax returns of several tax payers aggregated into a community tax budget. The leftmost, i.e., the resulting document may be an edited (i.e., an update) version of one of the rightmost, i.e., input documents.

See right part of Fig. 8.19 on the facing page.

Transparency Implies $\mathcal{E}^2\mathcal{G}$ Traceability

Given a document, say the "bottom-right-most" $\surd$*-checked* document one may wish to trace all its "ancestor" documents. Such a trace may — as indicated — lead to several traces, that is, to sequences of document references. For an agent or a citizen such *traceability* implies *transparency* of public government. See Fig. 8.20 on the next page.

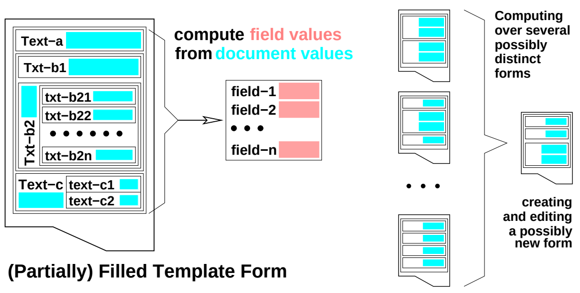

Text−a
Txt−b1
Txt−b2
txt−b21
txt−b22
txt−b2n
Text−c
text−c1
text−c2
compute field values from document values
field−1
field−2
field−n
(Partially) Filled Template Form
Computing over several possibly distinct forms
creating and editing a possibly new form

Fig. 8.19. $\mathcal{E}^2\mathcal{G}$ document computations

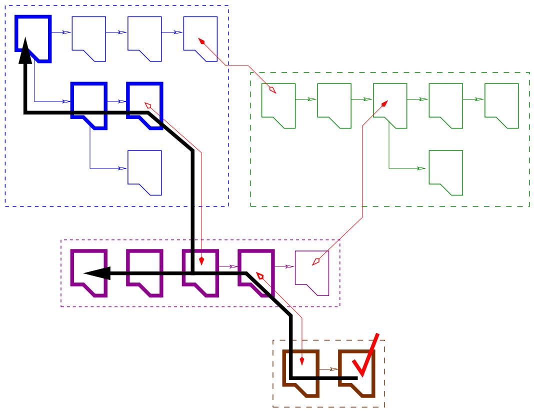


Fig. 8.20. Traces

8.4.2 Summary of $\mathcal{E}^2\mathcal{G}$

First Summary

So $\mathcal{E}$ssential $\mathcal{E}$-$\mathcal{G}$overnment, $\mathcal{E}^2\mathcal{G}$, assumes *paper-less* administration, i.e., all electronic documents and focuses on all/most such *document*-manifested actions which are *based in law*. Such documents entail: (1) Which *branches* of public government are *included/excluded*? (2) Which *document classes* are *included/excluded*? (3) Which *authorisations*, and which *scripts* are *mandated*? (4) Which *operations* on documents are *supported*? (5) Which form of *traceability* is *supported*?

Second Summary of $\mathcal{E}^2\mathcal{G}$

Figure 8.21 intimates some properties of an assumed modular, i.e., parameterised $\mathcal{E}^2\mathcal{G}$ server. The dotted rounded-edge left box stands for such a server. You may think of one server per actor. The figure shows an agent actor. For the citizen actor one omits the license and document interface. The syntax of the up and down pointing license and document interfaces (to central, respectively local agents) is a simplification of the possibility of n such sets of license and document interfaces, one pair to each "other" agent being interfaced. The syntax of the leftward license and document interface to citizens is a simplification of the possibility of m such sets of license and document interfaces, one pair to each citizen being interfaced.

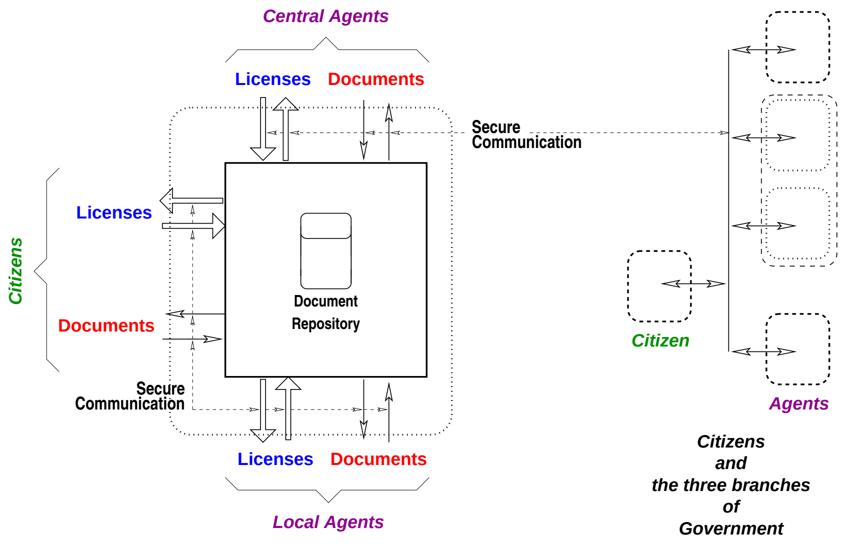


Fig. 8.21. A possible modular software server

Figure 8.21 intends to show that every actor receives licenses and documents, that is, with each document received there follows a license which details the authorisations, that is, the rights, that that actor is given wrt. that document.

Figure 8.21 further intends to show that any actor may send licenses and documents, that is, each document sent (moved, distributed) there is accompanied with license which details the authorisations, that is, the rights, that the receiving actor is given wrt. that document.

Finally Fig. 8.21 should be augmented to imply that the square full line box within the dotted rounded-edge left box checks that licenses (that are) sent harmonises with licenses (that are) received for "derived" documents.

8.5 Closing

8.5.1 What Have We Covered?

What Have We Covered? We have covered: (1) three branches of government $\leftrightarrow$ citizens; (2) flow of documents; (3) actors: citizens and agents; (4) documents: (4.1) create, edit, copy, move, read, shred, search, compute; (4.2) document version, history, reference, trace; and (4.3) authorisation, license; and (5) $\mathcal{E}^2\mathcal{G}$: $\mathcal{E}$ssential $\mathcal{E}$-$\mathcal{G}$overnment.

8.5.2 What Did We Try to Achieve?

We have attempted to illuminate: (a) a systematic approach to understanding the *domain* of public government; (b) another approach to asking for $\mathcal{E}^2$-$\mathcal{G}$overnment; (c) a focus of *law-based documents*; and (d) a view of *good governance*: (d.1) *Transparency of document handling* $\Rightarrow$ and (d.2) *transparency of the 'Rule of Law'.*

8.5.3 Relation to Chapters 7, 9 and 10

(We refer to the 'Discussion' section, Sect. 7.4 on page 199 of the previous chapter.)

In Chap. 7 we initially covered the notion of documents, both informally and formally.

In Chap. 9 we shall cover the notion of security, both informally and formally — but with only a superficial treatment of authorisation.

In Chap. 10 we shall cover the notion of authorising scripts, licenses and contracts, both informally and formally.

8.5.4 Towards a Theory of Documents

Together Chaps. 7–10 amount to an emerging Theory of Documents. Either such a theory is more fully researched and developed in a PhD study or is an industrial R&D project by a company, like `Xerox`, `Microsoft` or `IBM`, where that company claims to offer, for example, public administration, a comprehensive and consistent, expanding set of services, hardware and software for document handling.

9

Towards a Model of IT Security[1]

The ISO Information Security Code of Practice

An Incomplete Rough Sketch Analysis

Caveat

This chapter is incomplete. Its basis, [36], is even more so. We could have wished to bring a more complete analysis of the IT Security domain. At least 12 man-months are needed in order to bring a more complete and comprehensive treatment.

Summary

We analyse the domain of IT systems and 'add' to that domain the concept of IT Security Rules (and Regulations). The analysis is done, first informally, then formally. The informal analysis and its presentation follows the 'dogmas' set out in Vol.3 of Software Engineering [33]. The formal presentation follows the principles and techniques and uses the tools outlined in Vols.1-2 of the afore-mentioned book [31, 32].

9.1 Introduction

IT systems are becoming increasingly ubiquitous and vulnerable: they are everywhere, integrating 'seamlessly' into our everyday activities, and are (therefore, because also of their seamlessness) vulnerable to fail wrt. proper, intended operation either due to malicious attacks by intruders, or due to 'acts of nature': earthquakes, typhoons, fire. Such failure of operation may have catastrophic consequences: loss of life or property, exposure of personal or company information or of state secrecy.

To safeguard against such consequences, to secure privacy, to maintain 'competitive edges', etc., it has become increasingly important to establish codes of practice for information security management, that is, to secure that

[1]This is an edited version of [36].

IT operations and data cannot be interfered with by un-authorised people or un-intended machinery. and not disrupted by 'acts of nature'.

Information security management has become, sorry to express it in this non-scientific manner, 'a hot topic'.

Yet the issue is not at all that clear. What really is an IT system? What is really meant by IT system security? Quite substantial amounts of resources are being spent today: monies, staff time in preparation, monitoring and control; and quite significant disruption of normal, otherwise very reasonable work practice are often incurred as a side-effect of ensuring IT system security.

It is therefore mandatory that the topic of 'information security management' be subject to a scientific study.

This then is the purpose of this chapter: to provide one such approach to a scientific study of 'information security management' while recognising that other approaches exists (but yet to be studied and reported).

The present study shall attempt to answer the questions: *what is an IT system? what is IT system security?* and *what is a code of practice for IT system security management?* — with these questions, in this chapter, being *only tentatively* answered in the, by now, classical style of (i) IT system domain modelling: the syntax and semantics of the IT system entities, functions, events and behaviours and (ii) of IT system security rules and regulations: their syntax and semantics relative to the domain model of IT systems

We are not aware of any attempts of formally understanding the issues of 'information security management' in the almost "holistic" sense of this presentation.

We venture to say that there is perhaps a whole new methodological (i.e., modelling) approach to emerge from this study.

As we show, we can apply this approach to such physical notions as building sites, buildings, their floors, rooms, etc.; building, room, etc. installations: wires, switches, pipes, valves, sensors, actuators, etc.; movable equipment: main frames, laptops, file cabinets, etc.; people; as well as to related conceptual notions: codes of practice, security rules, recordings of intrusions and their handling, etc. But we venture to claim that the approach can also be applied to similar systems: hospitals, factories, concert halls, hotels, etc. We know of no other modelling approach that can capture the depth and width as shown here.

Well, before being caught too optimistic, let's see how far we can get. Remember: it is still very much work in progress.

9.2 Our Methodological Approach

We choose the following sequence of analysis and synthesis actions: First we bring excerpts from the ISO Standard: INTERNATIONAL ISO/IEC STANDARD 17799: Information technology: security techniques — code of practice for information security management. On the basis of these rather cursory excerpts but also on the basis of a more comprehensive analysis — both of which we do not show — we postulate in five sections (Sects. 9.6–9.7) a domain model for IT systems.

Section 9.6 prepares for the formal model of IT systems given in Sect. 9.7. A formal model of the meaning of 'security rules and regulations' is then sketched (Sect. 9.8).

We end the chapter with some speculations as how to proceed with what has been presented in this chapter.

The formal model has two components: A formal model of system configurations: states and contexts; and a formal model of the "codes of practice for information security management". The former model is a conventional, software engineering model of "a system". Maybe there are some novel aspects that enable us to perform spatial (or diagrammatic) reasoning. Maybe existing work on spatial reasoning ought be consulted [4]. The latter model is a rather conventional model of the semantics of well formed formulas (*wff*s) in logic — without including modal operations — curiously absent, it seems, from the "ISO Code of Practice". The assumption being made here is that all "implementation guideline" statements of the "ISO Code of Practice" can be expressed in some (first ?) order predicate calculus.

This approach to the modelling of a "code of practice for information security management" is tentative. That is, it is an experiment. Maybe we succeed. Maybe we do not. The work reported here is thus of the following nature: it is experimental, it aims at understanding the domain of IT systems and of the related "code of practice for information security management". and of testing our principles and techniques of domain engineering with this "testing" being carried out in Sects. 9.6–9.7. If we get a formal model of the ISO (standard) "code of practice for information security management" that reveals that can be used to question this "code of practice", that can be used for "prediction", and on the basis of which we can implement computing and communication) systems support for this "practice" then we would claim the experiment for being successful.

9.3 An Example Set of IT System Codes of Practice

We quote extensively from INTERNATIONAL ISO/IEC STANDARD 17799: Information technology: security techniques — code of practice for information security management.

9.3.1 [6] Organisation of information security[2]

[2]The [bracketed] sections, subsections, etc., refer to respective sections, etc., in the ISO IT Security Management Code of Practice.

[6.1] Internal Organisation

[6.1.1] Management commitment to information security

Control:

Management should actively support security within the organization through clear direction, demonstrated commitment, explicit assignment, and acknowledgment of information security responsibilities.

Implementation guidance:

Management should:

1. ensure that information security goals are identified, meet the organizational requirements, and are integrated in relevant processes;
2. formulate, review, and approve information security policy;
3. review the effectiveness of the implementation of the information security policy;
4. provide clear direction and visible management support for security initiatives;
5. provide the resources needed for information security;
6. approve assignment of specific roles and responsibilities for information security across the organization;
7. initiate plans and programs to maintain information security awareness;
8. ensure that the implementation of information security controls is co-ordinated across the organization (see 6.1.2).

[6.1.2] Information security co-ordination

Control:

Information security activities should be co-ordinated by representatives from different parts of the organization with relevant roles and job functions.

Implementation guidance:

Typically, information security co-ordination should involve the co-operation and collaboration of managers, users, administrators, application designers, auditors and security personnel, and specialist skills in areas such as insurance, legal issues, human resources, IT and risk management.

This activity should:

1. ensure that security activities are executed in compliance with the information security policy;
2. identify how to handle non-compliances;
3. approve methodologies and processes for information security, e.g. risk assessment, information classification;
4. identify significant threat changes and exposure of information and information processing facilities to threats;
5. assess the adequacy and co-ordinate the implementation of information security controls;

6. effectively promote information security education, training and awareness throughout the organization;
7. evaluate information received from the monitoring and reviewing of information security incidents, and recommend appropriate actions in response to identified information security incidents.

[6.2] External parties

Objective: (1) To maintain the security of the organization's information and information processing facilities that are accessed, processed, communicated to, or managed by external parties. (2) The security of the organization's information and information processing facilities should not be reduced by the introduction of external party products or services. (3) Any access to the organization's information processing facilities and processing and communication of information by external parties should be controlled. (4) Where there is a business need for working with external parties that may require access to the organization's information and information processing facilities, or in obtaining or providing a product and service from or to an external party, a risk assessment should be carried out to determine security implications and control requirements. Controls should be agreed and defined in an agreement with the external party.

[6.2.1] Identification of risks related to external parties

Control:
The risks to the organization's information and information processing facilities from business processes involving external parties should be identified and appropriate controls implemented before granting access.

Implementation guidance:
Where there is a need to allow an external party access to the information processing facilities or information of an organization, a risk assessment (see also Section 4) should be carried out to identify any requirements for specific controls. The identification of risks related to external party access should take into account the following issues:

1. the information processing facilities an external party is required to access;
2. the type of access the external party will have to the information and information processing facilities, e.g.:
   (a) physical access, e.g. to offices, computer rooms, filing cabinets;
   (b) logical access, e.g. to an organization's databases, information systems;
   (c) network connectivity between the organization's and the external partyös network(s), e.g. permanent connection, remote access;
   (d) whether the access is taking place on-site or off-site;

3. the value and sensitivity of the information involved, and its criticality for business operations;
4. the controls necessary to protect information that is not intended to be accessible by external parties;
5. the external party personnel involved in handling the organization's information;
6. how the organization or personnel authorized to have access can be identified, the authorization verified, and how often this needs to be reconfirmed;
7. the different means and controls employed by the external party when storing, processing, communicating, sharing and exchanging information;
8. the impact of access not being available to the external party when required, and the external party entering or receiving inaccurate or misleading information;
9. practices and procedures to deal with information security incidents and potential damages, and the terms and conditions for the continuation of external party access in the case of an information security incident;
10. legal and regulatory requirements and other contractual obligations relevant to the external party that should be taken into account;
11. how the interests of any other stakeholders may be affected by the arrangements.

Access by external parties to the organization's information should not be provided until the appropriate controls have been implemented and, where feasible, a contract has been signed defining the terms and conditions for the connection or access and the working arrangement. Generally, all security requirements resulting from work with external parties or internal controls should be reflected by the agreement with the external party (see also 6.2.2 and 6.2.3).

It should be ensured that the external party is aware of their obligations, and accepts the responsibilities and liabilities involved in accessing, processing, communicating, or managing the organization's information and information processing facilities.

Other information:

Information might be put at risk by external parties with inadequate security management. Controls should be identified and applied to administer external party access to information processing facilities. For example, if there is a special need for confidentiality of the information, non-disclosure agreements might be used.

Organizations may face risks associated with inter-organizational processes, management, and communication if a high degree of outsourcing is applied, or where there are several external parties involved.

The controls 6.2.2 and 6.2.3 cover different external party arrangements, e.g. including:

1. service providers, such as ISPs, network providers, telephone services, maintenance and support services;
2. managed security services;
3. customers;
4. outsourcing of facilities and/or operations, e.g. IT systems, data collection services, call centre operations;
5. management and business consultants, and auditors;
6. developers and suppliers, e.g. of software products and IT systems;
7. cleaning, catering, and other outsourced support services;
8. temporary personnel, student placement, and other casual short-term appointments.

Such agreements can help to reduce the risks associated with external parties.

9.3.2 [7] Asset management

[7.1] Responsibility for assets

[7.1.1] Inventory of assets

Control: All assets should be clearly identified and an inventory of all important assets drawn up and maintained.

Implementation guidance:

An organization should identify all assets and document the importance of these assets. The asset inventory should include all information necessary in order to recover from a disaster, including type of asset, format, location, backup information, license information, and a business value. The inventory should not duplicate other inventories unnecessarily, but it should be ensured that the content is aligned.

In addition, ownership (see 7.1.2) and information classification (see 7.2) should be agreed and documented for each of the assets. Based on the importance of the asset, its business value and its security classification, levels of protection commensurate with the importance of the assets should be identified (more information on how to value assets to represent their importance can be found in ISO/IEC TR 13335-3).

Other information: There are many types of assets, including:

1. information: databases and data files, contracts and agreements, system documentation, research information, user manuals, training material, operational or support procedures, business continuity plans, fallback arrangements, audit trails, and archived information;
2. software assets: application software, system software, development tools, and utilities;

3. physical assets: computer equipment, communications equipment, removable media, and other equipment;
4. services: computing and communications services, general utilities, e.g. heating, lighting, power, and air-conditioning;
5. people, and their qualifications, skills, and experience;
6. intangibles, such as reputation and image of the organization.

Inventories of assets help to ensure that effective asset protection takes place, and may also be required for other business purposes, such as health and safety, insurance or financial (asset management) reasons. The process of compiling an inventory of assets is an important prerequisite of risk management (see also Section 4).

9.3.3 [8] Human resources security

[8.1] Prior to employment

(Explanation: The word 'employment' is meant here to cover all of the following different situations: employment of people (temporary or longer lasting), appointment of job roles, changing of job roles, assignment of contracts, and the termination of any of these arrangements.)

Objective: To ensure that employees, contractors and third party users understand their responsibilities, and are suitable for the roles they are considered for, and to reduce the risk of theft, fraud or misuse of facilities.

Security responsibilities should be addressed prior to employment in adequate job descriptions and in terms and conditions of employment.

All candidates for employment, contractors and third party users should be adequately screened, especially for sensitive jobs.

Employees, contractors and third party users of information processing facilities should sign an agreement on their security roles and responsibilities.

[8.1.1] Roles and responsibilities

Control:

Security roles and responsibilities of employees, contractors and third party users should be defined and documented in accordance with the organization's information security policy.

Implementation guidance:

Security roles and responsibilities should include the requirement to:

1. implement and act in accordance with the organization¡¯s information security policies (see 5.1);
2. protect assets from unauthorized access, disclosure, modification, destruction or interference;
3. execute particular security processes or activities;
4. ensure responsibility is assigned to the individual for actions taken;

5. report security events or potential events or other security risks to the organization.

Security roles and responsibilities should be defined and clearly communicated to job candidates during the pre-employment process.

9.3.4 [9] Physical and environmental security

[9.1] Secure areas

Objective: To prevent unauthorized physical access, damage, and interference to the organization's premises and information. Critical or sensitive information processing facilities should be housed in secure areas, protected by defined security perimeters, with appropriate security barriers and entry controls. They should be physically protected from unauthorized access, damage, and interference. The protection provided should be commensurate with the identified risks.

[9.1.1] Physical security perimeter

Control: Security perimeters (barriers such as walls, card controlled entry gates or manned reception desks) should be used to protect areas that contain information and information processing facilities.

Implementation guidance:

The following guidelines should be considered and implemented where appropriate for physical security perimeters:

1. security perimeters should be clearly defined, and the siting and strength of each of the perimeters should depend on the security requirements of the assets within the perimeter and the results of a risk assessment;
2. perimeters of a building or site containing information processing facilities should be physically sound (i.e. there should be no gaps in the perimeter or areas where a break-in could easily occur); the external walls of the site should be of solid construction and all external doors should be suitably protected against unauthorized access with control mechanisms, e.g. bars, alarms, locks etc; doors and windows should be locked when unattended and external protection should be considered for windows, particularly at ground level;
3. a manned reception area or other means to control physical access to the site or building should be in place; access to sites and buildings should be restricted to authorized personnel only;
4. physical barriers should, where applicable, be built to prevent unauthorized physical access and environmental contamination;
5. all fire doors on a security perimeter should be alarmed, monitored, and tested in conjunction with the walls to establish the required level of resistance in accordance to suitable regional, national, and international

standards; they should operate in accordance with local fire code in a failsafe manner;
6. suitable intruder detection systems should be installed to national, regional or international standards and regularly tested to cover all external doors and accessible windows; unoccupied areas should be alarmed at all times; cover should also be provided for other areas, e.g. computer room or communications rooms;
7. information processing facilities managed by the organization should be physically separated from those managed by third parties.

[9.1.2] Physical entry controls

Control: Secure areas should be protected by appropriate entry controls to ensure that only authorized personnel are allowed access.

Implementation guidance:

1. the date and time of entry and departure of visitors should be recorded, and all visitors should be supervised unless their access has been previously approved; they should only be granted access for specific, authorized purposes and should be issued with instructions on the security requirements of the area and on emergency procedures.
2. access to areas where sensitive information is processed or stored should be controlled and restricted to authorized persons only; authentication controls, e.g. access control card plus PIN, should be used to authorize and validate all access; an audit trail of all access should be securely maintained;
3. all employees, contractors and third party users and all visitors should be required to wear some form of visible identification and should immediately notify security personnel if they encounter un-escorted visitors and anyone not wearing visible identification;
4. third party support service personnel should be granted restricted access to secure areas or sensitive information processing facilities only when required; this access should be authorized and monitored;
5. access rights to secure areas should be regularly reviewed and updated, and revoked when necessary (see 8.3.3).

[9.1.3] Securing offices, rooms, and facilities

Control Physical security for offices, rooms, and facilities should be designed and applied.

Implementation guidance: The following guidelines should be considered to secure offices, rooms, and facilities:

1. account should be taken of relevant health and safety regulations and standards;
2. key facilities should be sited to avoid access by the public;

3. where applicable, buildings should be unobtrusive and give minimum indication of their purpose, with no obvious signs, outside or inside the building identifying the presence of information processing activities;
4. directories and internal telephone books identifying locations of sensitive information processing facilities should not be readily accessible by the public.

[9.1.4] Protecting against external and environmental threats

Control: Physical protection against damage from fire, flood, earthquake, explosion, civil unrest, and other forms of natural or man-made disaster should be designed and applied.

Implementation guidance:

Consideration should be given to any security threats presented by neighboring premises, e.g. a fire in a neighbouring building, water leaking from the roof or in floors below ground level or an explosion in the street.

1. hazardous or combustible materials should be stored at a safe distance from a secure area. Bulk supplies such as stationery should not be stored within a secure area;
2. fallback equipment and back-up media should be sited at a safe distance to avoid damage from a disaster affecting the main site;
3. appropriate fire fighting equipment should be provided and suitably placed.

[9.1.5] Working in secure areas

Control: Physical protection and guidelines for working in secure areas should be designed and applied.

Implementation guidance:

1. personnel should only be aware of the existence of, or activities within, a secure area on a need to know basis;
2. unsupervised working in secure areas should be avoided both for safety reasons and to prevent opportunities for malicious activities;
3. vacant secure areas should be physically locked and periodically checked;
4. photographic, video, audio or other recording equipment, such as cameras in mobile devices, should not be allowed, unless authorized;

The arrangements for working in secure areas include controls for the employees, contractors and third party users working in the secure area, as well as other third party activities taking place there.

[9.1.6] Public access, delivery, and loading areas

Control: Access points such as delivery and loading areas and other points where unauthorized persons may enter the premises should be controlled and, if possible, isolated from information processing facilities to avoid unauthorized access.

[9.2] Equipment security

Objective: To prevent loss, damage, theft or compromise of assets and interruption to the organization's activities. Equipment should be protected from physical and environmental threats. Protection of equipment (including that used off-site, and the removal of property) is necessary to reduce the risk of unauthorized access to information and to protect against loss or damage. This should also consider equipment siting and disposal. Special controls may be required to protect against physical threats, and to safeguard supporting facilities, such as the electrical supply and cabling infrastructure.

[9.2.1] Equipment siting and protection

Control: Equipment should be sited or protected to reduce the risks from environmental threats and hazards, and opportunities for unauthorized access.

[9.2.2] Supporting utilities

Control: Equipment should be protected from power failures and other disruptions caused by failures in supporting utilities.

[9.2.3] Cabling security

Control: Power and telecommunications cabling carrying data or supporting information services should be protected from interception or damage.

[9.2.4] Equipment maintenance

Control: Equipment should be correctly maintained to ensure its continued availability and integrity.

[9.2.5] Security of equipment off-premises

Control: Security should be applied to off-site equipment taking into account the different risks of working outside the organization's premises.

Implementation guidance: Regardless of ownership, the use of any information processing equipment outside the organization's premises s must be authorized by management.

1. equipment and media taken off the premises should not be left unattended in public places; portable computers should be carried as hand luggage and disguised where possible when travelling;
2. manufacturers' instructions for protecting equipment should be observed at all times, e.g. protection against exposure to strong electromagnetic fields;

3. home-working controls should be determined by a risk assessment and suitable controls applied as appropriate, e.g. lockable filing cabinets, clear desk policy, access controls for computers and secure communication with the office (see also ISO/IEC 18028 Network Security);
4. adequate insurance cover should be in place to protect equipment off-site.

Security risks, e.g. of damage, theft or eavesdropping, may vary considerably between locations and should be taken into account in determining the most appropriate controls.

9.3.5 [10] Communications and operations management

[10.1] Operational procedures and responsibilities

Objective: To ensure the correct and secure operation of information processing facilities. Responsibilities and procedures for the management and operation of all information processing facilities should be established. This includes the development of appropriate operating procedures. Segregation of duties should be implemented, where appropriate, to reduce the risk of negligent or deliberate system misuse.

[10.1.1] Documented operating procedures

Control: Operating procedures should be documented, maintained, and made available to all users who need them.

[10.1.2] Change management

Control: Changes to information processing facilities and systems should be controlled.

[10.1.4] Separation of development, test, and operational facilities

Control: Development, test, and operational facilities should be separated to reduce the risks of un-authorised access or changes to the operational system.

[10.4] Protection against malicious and mobile code

Objective: To protect the integrity of software and information. Precautions are required to prevent and detect the introduction of malicious code and unauthorized mobile code. Software and information processing facilities are vulnerable to the introduction of malicious code, such as computer viruses, network worms, Trojan horses, and logic bombs. Users should be made aware of the dangers of malicious code. Managers should, where appropriate, introduce controls to prevent, detect, and remove malicious code and control mobile code.

[10.4.1] Controls against malicious code

Control: Detection, prevention, and recovery controls to protect against malicious code and appropriate user awareness procedures should be implemented.

[10.5] Back-up

Objective: To maintain the integrity and availability of information and information processing facilities. Routine procedures should be established to implement the agreed back-up policy and strategy for taking back-up copies of data and rehearsing their timely restoration.

[10.5.1] Information back-up

Control: Back-up copies of information and software should be taken and tested regularly in accordance with the agreed backup policy.

[10.6] Network security management

Objective: To ensure the protection of information in networks and the protection of the supporting infrastructure.

The secure management of networks, which may span organizational boundaries, requires careful consideration to data-flow, legal implications, monitoring, and protection. Additional controls may also be required to protect sensitive information passing over public networks.

[10.6.1] Network controls

Control: Networks should be adequately managed and controlled, in order to be protected from threats, and to maintain security for the systems and applications using the network, including information in transit.

[10.7] Media handling

Objective: To prevent unauthorized disclosure, modification, removal or destruction of assets, and interruption to business activities.

Media should be controlled and physically protected.

Appropriate operating procedures should be established to protect documents, computer media (e.g. tapes, disks), input/output data and system documentation from unauthorized disclosure, modification, removal, and destruction.

[10.7.1] Management of removable media

Control: There should be procedures in place for the management of removable media.

[10.7.2] Disposal of media

Control: Media should be disposed-of securely and safely when no longer required, using formal procedures.

[10.7.3] Information handling procedures

Control: Procedures for the handling and storage of information should be established to protect this information from unauthorized disclosure or misuse.

[10.7.4] Security of system documentation

Control: System documentation should be protected against unauthorized access.

[10.8] Exchange of information

Objective: To maintain the security of information and software exchanged within an organization and with any external entity.

Exchanges of information and software between organizations should be based on a formal exchange policy, carried out in line with exchange agreements, and should be compliant with any relevant legislation (see clause 15).

Procedures and standards should be established to protect information and physical media containing information in transit.

[10.8.1] Information exchange policies and procedures

Control: Formal exchange policies, procedures, and controls should be in place to protect the exchange of information through the use of all types of communication facilities.

[10.8.3] Physical media in transit

Control: Media containing information should be protected against unauthorized access, misuse or corruption during transportation beyond an organization's physical boundaries.

Implementation guidance:

1. reliable transport or couriers should be used;
2. a list of authorized couriers should be agreed with management;
3. procedures to check the identification of couriers should be developed;
4. packaging should be sufficient to protect the contents from any physical damage likely to arise during transit and in accordance with any manufacturers' specifications (e.g. for software), for example protecting against any environmental factors that may reduce the media's restoration effectiveness such as exposure to heat, moisture or electromagnetic fields;

5. controls should be adopted, where necessary, to protect sensitive information from unauthorized disclosure or modification; examples include:
   (a) use of locked containers;
   (b) delivery by hand;
   (c) tamper-evident packaging (which reveals any attempt to gain access);
   (d) in exceptional cases, splitting of the consignment into more than one delivery and dispatch by different routes.

[10.8.4] Electronic messaging

Control: Information involved in electronic messaging should be appropriately protected.

Implementation guidance:

1. protecting messages from unauthorized access, modification or denial of service;
2. ensuring correct addressing and transportation of the message;
3. general reliability and availability of the service;
4. legal considerations, for example requirements for electronic signatures;
5. obtaining approval prior to using external public services such as instant messaging or file sharing;
6. stronger levels of authentication controlling access from publicly accessible networks.

[10.10] Monitoring

Objective: To detect unauthorized information processing activities.

Systems should be monitored and information security events should be recorded. Operator logs and fault logging should be used to ensure information system problems are identified.

An organization should comply with all relevant legal requirements applicable to its monitoring and logging activities.

System monitoring should be used to check the effectiveness of controls adopted and to verify conformity to an access policy model.

[10.10.1] Audit logging

Control: Audit logs recording user activities, exceptions, and information security events should be produced and kept for an agreed period to assist in future investigations and access control monitoring.

[10.10.2] Monitoring system use

Control: Procedures for monitoring use of information processing facilities should be established and the results of the monitoring activities reviewed regularly.

Implementation guidance: The level of monitoring required for individual facilities should be determined by a risk assessment. An organisation should comply with all relevant legal requirements applicable to its monitoring activities.

Areas that should be considered include:

1. authorized access, including detail such as:
   (a) the user ID;
   (b) the date and time of key events;
   (c) the types of events;
   (d) the files accessed;
   (e) the program/utilities used;
2. all privileged operations, such as:
   (a) use of privileged accounts, e.g. supervisor, root, administrator;
   (b) system start-up and stop;
   (c) I/O device attachment/detachment;
3. unauthorized access attempts, such as:
   (a) failed or rejected user actions;
   (b) failed or rejected actions involving data and other resources;
   (c) access policy violations and notifications for network gateways and firewalls;
   (d) alerts from proprietary intrusion detection systems;
4. system alerts or failures such as:
   (a) console alerts or messages;
   (b) system log exceptions;
   (c) network management alarms;
   (d) alarms raised by the access control system;
5. changes to, or attempts to change, system security settings and controls.

How often the results of monitoring activities are reviewed should depend on the risks involved. Risk factors that should be considered include the:

1. criticality of the application processes;
2. value, sensitivity, and criticality of the information involved;
3. past experience of system infiltration and misuse, and the frequency of vulnerabilities being exploited;
4. extent of system interconnection (particularly public networks);
5. logging facility being de-activated.

9.3.6 [11] Access control

[11.1] Business requirement for access control

Objective: To control access to information. Access to information, information processing facilities, and business processes should be controlled on the basis of business and security requirements. Access control rules should take account of policies for information dissemination and authorization.

[11.1.1] Access control policy

Control: An access control policy should be established, documented, and reviewed based on business and security requirements for access.

[11.2] User access management

Objective: To ensure authorized user access and to prevent unauthorized access to information systems. Formal procedures should be in place to control the allocation of access rights to information systems and services.

The procedures should cover all stages in the life-cycle of user access, from the initial registration of new users to the final de-registration of users who no longer require access to information systems and services. Special attention should be given, where appropriate, to the need to control the allocation of privileged access rights, which allow users to override system controls.

[11.2.1] User registration

Control: There should be a formal user registration and de-registration procedure in place for granting and revoking access to all information systems and services.

Implementation guidance:

1. using unique user IDs to enable users to be linked to and held responsible for their actions; the use of group IDs should only be permitted where they are necessary for business or operational reasons, and should be approved and documented;
2. checking that the user has authorization from the system owner for the use of the information system or service; separate approval for access rights from management may also be appropriate;
3. checking that the level of access granted is appropriate to the business purpose (see 11.1) and is consistent with organizational security policy, e.g. it does not compromise segregation of duties (see 10.1.3);
4. giving users a written statement of their access rights;
5. requiring users to sign statements indicating that they understand the conditions of access;

6. ensuring service providers do not provide access until authorization procedures have been completed;
7. maintaining a formal record of all persons registered to use the service;
8. immediately removing or blocking access rights of users who have changed roles or jobs or left the organization;
9. periodically checking for, and removing or blocking, redundant user IDs and accounts (see 11.2.4);
10. ensuring that redundant user IDs are not issued to other users.

Other information: Consideration should be given to establish user access roles based on business requirements that summarize a number of access rights into typical user access profiles. Access requests and reviews (see 11.2.4) are easier managed at the level of such roles than at the level of particular rights.

Consideration should be given to including clauses in personnel contracts and service contracts that specify sanctions if unauthorized access is attempted by personnel or service agents (see also 6.1.5, 8.1.3 and 8.2.3).

[11.2.2] Privilege management

Control: The allocation and use of privileges should be restricted and controlled.

[11.2.3] User password management

Control: The allocation of passwords should be controlled through a formal management process.

[11.2.4] Review of user access rights

Control: Management should review users' access rights at regular intervals using a formal process.

[11.4] Network access control

Objective: To prevent unauthorized access to networked services. Access to both internal and external networked services should be controlled. User access to networks and network services should not compromise the security of the network services by ensuring:

1. appropriate interfaces are in place between the organization's network and networks owned by other organizations, and public networks;
2. appropriate authentication mechanisms are applied for users and equipment;
3. control of user access to information services in enforced.

[11.4.1] Policy on use of network services

Control: Users should only be provided with access to the services that they have been specifically authorized to use.

[11.4.2] User authentication for external connections

Control: Appropriate authentication methods should be used to control access by remote users.

[11.4.3] Equipment identification in networks

Control: Automatic equipment identification should be considered as a means to authenticate connections from specific locations and equipment.

[11.4.4] Remote diagnostic and configuration port protection

Control: Physical and logical access to diagnostic and configuration ports should be controlled.

[11.4.5] Segregation in networks

Control: Groups of information services, users, and information systems should be segregated on networks.

[11.5] Operating system access control

Objective: To prevent unauthorized access to operating systems. Security facilities should be used to restrict access to operating systems to authorized users.

1. authenticating authorized users, in accordance with a defined access control policy;
2. recording successful and failed system authentication attempts;
3. recording the use of special system privileges;
4. issuing alarms when system security policies are breached;
5. providing appropriate means for authentication;
6. where appropriate, restricting the connection time of users.

[11.5.1] Secure log-on procedures

Control: Access to operating systems should be controlled by a secure log-on procedure.

9.3.7 [13] Information security incident management

[13.1] Reporting information security events and weaknesses

Objective: To ensure information security events and weaknesses associated with information systems are communicated in a manner allowing timely corrective action to be taken.

Formal event reporting and escalation procedures should be in place. All employees, contractors and third party users should be made aware of the procedures for reporting the different types of event and weakness that might have an impact on the security of organizational assets. They should be required to report any information security events and weaknesses as quickly as possible to the designated point of contact.

[13.1.1] Reporting information security events

Control: Information security events should be reported through appropriate management channels as quickly as possible.

[13.2] Management of information security incidents and improvements

Objective: To ensure a consistent and effective approach is applied to the management of information security incidents.

Responsibilities and procedures should be in place to handle information security events and weaknesses effectively once they have been reported. A process of continual improvement should be applied to the response to, monitoring, evaluating, and overall management of information security incidents.

Where evidence is required, it should be collected to ensure compliance with legal requirements.

[13.1.1] Reporting security weaknesses

Control:

All employees, contractors and third party users of information systems and services should be required to note and report any observed or suspected security weaknesses in systems or services.

9.4 The ISO Standard ISO/IEC 17799 Table-of-Contents

We bring the full table-of-contents of The ISO Standard ISO/IEC 17799 documents.

9.5 An Analysis of the ISO/IEC 17799 Code of Practice

We next analyse some of the ’codes of practice’ statements of Sect. 9.3 (Pages 225–243). Our analysis seeks to identify: (i) the entities, (ii) the predicates and functions, (iii) the events, and (iv) the behaviours referred to in these ’codes of practice’ statements.

You see, our problem with the ISO Standard, as well as with all the instantiations that we have studied, is that they take the domain of discourse for granted. They assume it. They never bother to carefully delineate, let alone describe it. Hence we have problem with *"what could be the semantics of these 'codes of practice' statements."*

For each of the 'Code of Practice' statements, which we repeat below, we "transliterate" this statement into predicate form. It all looks very "formal". But, of course, since we have not given, and will not, in this monograph, give, a complete axiom system for all the entities (i.e., predicate and (term) function arguments) and for all the postulated predicates and functions, the below transliterations are, at most, "pseudo-formal". We do, of course, eventually aim at properly formalising the system now hinted at.

9.5.1 [6.1.1] Management commitment to information security

The 'Code of Practice' Statement

Management should:

1. ensure that information security goals are identified, meet the organizational requirements, and are integrated in relevant processes;
2. formulate, review, and approve information security policy;
3. review the effectiveness of the implementation of the information security policy;
4. provide clear direction and visible management support for security initiatives;
5. provide the resources needed for information security;
6. approve assignment of specific roles and responsibilities for information security across the organization;
7. initiate plans and programs to maintain information security awareness;
8. ensure that the implementation of information security controls is co-ordinated across the organization (see 6.1.2).

A Predicate Term Interpretation

```
1. exists('information_security_goals')(system)
   ∧ exists('organizational_requirements')(system)
   ∧ does_meet(system('information_security_goals'),
                          system('organizational_requirements'))
   ∧ is_integrated(system('information_security_goals'),
                          system('system_processes'))
2. exists('information_security_policy')(system)
   ∧ is_reviewed(system('information_security_policy'))
   ∧ is_approved(system('information_security_policy'))
3. is_effective(system('information_security_policy'))
4. exists('security_initiatives')(system)
   ∧ exists('directives')(system)
   ∧ is_visible((system('security_initiatives'))('management_support'))
```

5. is_adequate(system('resources')),
   (resources(system('information_security_policy')))
6. exists('role_assignment')(system('information_security'))
   $\wedge$ exists('responsibilities')(system('information_security'))
7. is_aware('information_security')(system)
   $\supset$ exists('plans')(system('information_security'))
   $\wedge$ exists('programs')(system('information_security'))
8. exists('information_security_controls')(system)
   $\supset$ is_coordinated('information_security_controls')(system)

Some Comments

1. **The formal expression:**

   exists('information_security_goals')(system)
   $\wedge$ exists('organizational_requirements')(system)
   $\wedge$ does_meet(system('information_security_goals'),
   system('organizational_requirements'))
   $\wedge$ is_integrated(system('information_security_goals'),
   system('system_processes'))

   Comments:

   - exists *names a rather general predicate.*
   - *It applies to a name n and the "entire"* system.
   - *It is thus assumed that this entire* system *will posses* a document *named* n.
   - *Thus* system(n) *"selects" that* document.
   - does_meet *names a predicate.*
   - *It applies to two documents.*
   - system('system_processes') *"selects" the current* system processes *— or, possibly, the possibly infinite set of all potential* system processes.
   - is_integrated *names a predicate.*
   - is_integrated *applies to a* document *and the (...)* system processes *and checks (somehow) that the entities designated by the* document *are* integrated *in these* processes.
   - *Note that the first argument of* is_integrated *is a* document *whereas the second argument is a* dynamic system entity.
2. **The formal expression:**

   exists('information_security_policy')(system)
   $\wedge$ is_reviewed(system('information_security_policy'))
   $\wedge$ is_approved(system('information_security_policy'))

   Comments:

   - *The assumption here is that the* document
     system('information_security_policy')

possess at least the attributes of having been 'reviewed' and having been 'approved'.

- *This entails two other assumptions: that that* document *is subject to the two corresponding functions*
  - ⋆ review and
  - ⋆ approve.

3. **The formal expression:**

   is_effective(system('information_security_policy'))

   Comments:

   - is_effective *names a predicate.*
   - *It applies to a document*
   - *and somehow determines whether it is* effective.

4. **The formal expression:**

   exists('security_initiatives')(system)
   ∧ exists('directives')(system('security_initiatives'))
   ∧ has_property('management_support')(system('security_initiatives'))

   Comments:

   - *There must be a* document *named* 'security_initiatives',
   - *there must be a* document *named* 'directives',
   - *say, as a* sub-document, *in the* document, *d*, *named* 'security_initiatives', *and*
   - *there must be a obvious, i.e., "visible" property of d*
   - *namely that it has* 'management_support'.

5. **The formal expression:**

   is_adequate(system('resources')),
   (resources(system('information_security_policy')))

   Comments:

   - system('resources') *yields all* system resources.
   - resources(system('information_security_policy')) *yields a "catalogue" of resources, say by name, needed to fulfill the* 'information_security_policy'.
   - is_adequate*is a predicate.*
   - *It applies to a catalogue of "real"* resources,*by value, and to a "catalogue" of resources, by name, and yields truth if the former are sufficient to satisfy the latter.*

6. **The formal expression:**

   exists('role_assignment')(system('information_security'))
   ∧ exists('responsibilities')(system('information_security'))

   Comments:

   - approval *is here taken to be tantamount to the* existence *of the designated* assignments.

7. **The formal expression:**

   is_aware('information_security')(system)
   $\supset$ exists('plans')(system('information_security'))
   $\wedge$ exists('programs')(system('information_security'))

   Comments:

   - is_aware *is a rather "sweeping" predicate.*
   - *Its implementation is simple:*
     - ⋆ *one sends an e-mail to all staff to inquire "are you aware of plans and programs to maintain information security ?".*
     - ⋆ *If a significant percentage replies yes, then predicate yields true !*
   - *More "formally"* awareness *implies that the designated* plans *and* programs *(documents and [probably] software) are found (somewhere) in the* system.

8. **The formal expression:**

   exists('information_security_controls')(system)
   $\supset$ is_coordinated('information_security_controls')(system)

   Comments:

   - *For this 'code of practice' we have, if not "given up" then at least (again) resorted to some rather "sweeping" generalisations:*
     - ⋆ *First we have postulated that there is a* document*by the name* 'information_security_controls',
     - ⋆ *and that that* document *does indeed address the issues covered by its name.*
     - ⋆ *Then we have used the same name* ('information_security_controls') *as the name of a* concept
     - ⋆ *and postulated an again "sweeping" predicate,* is_coordinated*, which "tests" the* system *for being in compliance with this* concept.
   - *The implementation of* is_coordinated *could be like that of* is_aware *above (Item 7).*

9.5.2 [9.1.1] Physical security perimeter

The 'Code of Practice' Statement

The following guidelines should be considered and implemented where appropriate for physical security perimeters:

1. security perimeters should be clearly defined, and the siting and strength of each of the perimeters should depend on the security requirements of the assets within the perimeter and the results of a risk assessment.

   We leave the remaining items of the ISO document [9.1.1] Physical security perimeter further untreated.

A Predicate Term Interpretation

1. **The informal expression:**

   security perimeters should be clearly defined, and the siting and strength of each of the perimeters should depend on the security requirements of the assets within the perimeter and the results of a risk assessment;

   The formal expression:

   ```
   is_well_defined('security perimeter')(system) ∧
   let ra = risk_assessment(system), sr = security_requirements(system)
       sas = siting_and_strength(system) in is_commensurate((ra,sr),sas) end
   ```

   Comments:

   - *An overall comment is this:*
     - ⋆ *The informal 'code of practice' assumes quite a lot:*
       - ◇ *that there is a complete understanding of the physical plant, i.e., the land site, its borders to and bordering with other sites; the composition of buildings on this site; the one or more floors of each of these buildings; their floor plans; etc., etc.*
   - *Specific, predicate-related comments are:*
     - ⋆
     - ⋆
     - ⋆

   We leave the remaining items of the ISO document [9.1.1] Physical security perimeter further untreated.

9.6 The Phenomena of IT Systems

The observable, manifest phenomena are: *simple entities*[3], *functions, events and behaviours.* Besides phenomena, *"that which we can see, hear, touch, smell, and taste"* and (or) measure with physics- (incl. chemistry-) based instruments, there are concepts. We shall treat concepts later.

Our *treatment of phenomena and concepts* is in the form of rough sketches, that is, not systematic, as a narrative, and not formal — but will later be. Also, we shall not establish a proper terminology but ought to have. We leave that as an exercise to the reader.

[3]We distinguish simple entities from entities. The later are all the phenomena and concepts of the domain. Functions apply to and yield entities; events involve predicates over entities. Since behaviours are sets of traces of actions and events — where actions derive from the application of operations to arguments — also behaviours involve the full notion of entities.

9.6.1 Simple Entities

General

By a simple entity we shall understand something physical, something we can point to, something which occupies space, or something which is an abstraction, a concept, thereof. Simple entities might "end up", in a computing system, like data in a database, or data associated with variables in a program. Simple entities are the "things" to which we apply functions.

First Examples of Simple Entities

Examples of simple entities are (1) the fixed physical plant: (a-b-c-d-e-...) buildings: halls, stairwells, corridors, rooms, etc., and (f-g-h-i-...) the ground around buildings: roads, walkways, parking areas, etc., (1) the installable semi-fixed building parts: (a) electrical wiring, (b) water and sewage piper, (c) burglary alarm systems, (c) fire detection and fire extinguish systems, (..) etc.; (2) the installable and relocatable (IT security-related) equipment: (a) main frame computers, (b) servers, (c)data communication cabling, (..) etc.; (3) the movable equipment: (a) mostly laptops; (4) people: (a) staff, (b) hired consultants, (c) clients, (d) potential customers, (e) invited visitors and (f) intruders; and (5) registers: (a) books and (b) databases (possibly kept on potentially movable storage media).

We shall now conceptually examine these simple entities more systematically.

Atomicity and Compositionality

One can can abstract a simple entity either as an atomic simple entity or as a composite simple entity. We decide to model a simple entity as an atomic simple entity if it is decided that it has not sub-structuring, that is, if one can not meaningfully, that is, in the context of the purpose of the model, decompose it into sub-entities. And we decide to model a simple entity as a composite simple entity if it is decided that it has a meaningful sub-structuring, hence consists of sub-entities. Atomic simple entities have attributes, that is, can be characterised by a number of properties, but these properties, as a whole, cannot be separated. Examples will follow. Composite simple entities have (i) simple sub-entities, (ii) a mereology, i.e., something which tells us how the simple entities are related to one another, and (iii) attributes. We shall consider these three kinds as independent of one another.

Atomic Simple Entities

An atomic simple entity is a simple entity whose possible "parts" we have decided not to consider, that is, to abstract from.

In one context a simple entity may be considered atomic while in another context it may be considered composite. In the context of IT Systems we decide to model human beings as atomic; while in the context of surgery (health care) we may decide to model human beings as composite.

Examples of Atomic Simple Entities

We give two examples of atomic entities of IT systems.

The first example of an atomic entity is that of a laptop. Its attributes are: brand name, model, serial number, storage hierarchy capacity, clock cycle, ports, etc.

The second example of an atomic simple entity is that of a human being. Personal attributes are: Name, gender, birth date, where born, citizenship, etc.; height, weight, color of eyes, etc.; education; IT skills; and IT responsibilities and IT authorisations.

Composite Simple Entities

A composite simple entity is a simple entity whose possible "parts" (that is, the sub-entities) we have decided consider, that is, to focus on — as well as how (the mereology of how) these simple sub-entities are put together. Add to our analysis of composite simple entities some properties that are properties of the composite entity, not of the simple sub-entities. We shall refer to these properties as attributes of the composite simple entity.

Simple Sub-entities and Their Mereology

Thus we shall examine sub-entities as "free-standing" components of composite entities, and we shall introduce the concept of mereology (the study and conceptual (philosophical) knowledge of "parts and wholes") to deal with the "free-standedness"!

Examples of Composite Simple Entities

We give two examples of composite simple entities.

The first example is of a building complex:

- Sub-entities of a building complex: the ground area of the building complex, the roads external to the ground area, the roads internal to the ground area, and the buildings on the ground area.
- Mereology of the simple sub-entities of a single floor building: Some external roads are connected to some internal roads, some buildings are connected to some internal roads, and some buildings are connected to some other buildings.
- Attributes of a building complex: the name of the building complex, the address of the building complex, the legal ownership of the building complex, the acreage (etc.) of the building complex, etcetera.

The second example is of a single floor building: the simple sub-entities are the entrance/exit ways of the building, the corridors and the rooms (walls, doors, windows, etc., are considered part of these entities); the mereology of a single floor building outlines the general or specific adjacency of entrance/exit ways, corridors and rooms; and the attributes are those of the name, owner(s), position (within some ground area), building materials, etc.

Attributes

We thus associate properties with atomic as well as composite simple entities. Simple entities have at least one attribute. We have decided that it makes no sense to speak of attribute-less simple entities.[4] We shall model an attribute as having a name (an attribute, or type name) and a value. A simple entity may have more than one attribute. In our narrative of multiply-attributed entities we do not consider their structuring (i.e., the "mereology"). We have concluded that any such perceived structuring of multiply-attributed entity attributes is irrelevant.[5]

Shared Attributes

We introduce a modelling notion of shared attributes. Examples are: a wall separating two rooms (or diving a larger room into two smaller rooms), a door (of a wall), being shared between two rooms and a window between a room and "an outside". A shared attribute may in one model not be modelled as a shared attribute, but as a sub-entity. An example could be a door (or a window) of a wall.

Summary of Simple Entities of IT Systems

We summarise simple entities of IT Systems 'of interests',[6] helter-skelter, with no apparent consideration of whether atomic or composite, or whether simple sub-entities of other simple entities: Next is a semi-structured, yet incomplete list of simple entities of IT Systems of interest: (1) *physical plant:* an or the IT System building complex, building ground, road, building, room, corridor, etc.; (2) *installations:* wiring, water piping, sewage piping, burglary detector & alarm, fire detector & alarm, fire extinguisher, etc.; (3) *movable equipment:* main frame, server, chair, table, cabinet, laptop, etc.; (4) *person*; and (5) *register*. You will have noticed, that we have grouped the IT System simple entities into five classes. This is a choice. We could have chosen another decomposition of simple entities into such classes. We shall later motivate the above

[4]This is, of course, a conjecture. As such we are ready to one day admit its refutation. "Science only makes progress through refutations"!

[5]This is, of course, another conjecture. As such we are, also in this case, ready to one day admit its refutation.

[6]We single out the term 'of interest' to indicate that, in some other model of "basically the same domain", there could have been another choice of simple entities.

grouping. The above choice will determine our formal modelling. Whether our choice is a good or a not so good choice will become apparent only if we formalise a number of alternative choices — and then evaluate their merits, their elegance.

Discussion

We will not in this document list "all" the simple entities of an IT System. Instead we will, in our formalisation introduce abstract, i.e., conceptual classes of simple entities. We have treated the analysis & modelling notion of IT System entities from an abstract, generic point of view, for example outlining composite phenomena of building complexes and buildings generically. In any particular application of the ideas of this document to a specific IT System the applier would then have to instantiate the general notion of building (etc.) mereologies to become very concrete. The above analysis & modelling approach applies to the next issues as well: functions, events and behaviours.

9.6.2 Functions

General

By a function[7] we shall understand *something*, an abstract concept, which when *applied* to a grouping of one or more entities, i.e. and *argument yields* a *result*, a value, in the form of either a grouping of *entities* or of *attributes* or a combination thereof.

Functions on (1) Physical Plant

Examples of functions that apply to entities of class physical plant are: (a) create a building, (b) change building attributes, (c) remove a building, (d) subdivide building rooms, (e) change wall attributes,[8] (f) connect two building, (g) create a road, (h) change a road, (i) remove a road, (..) etc.

Functions on (2) Installations

Examples of functions that apply to entities of class installations are: (a..) install wiring (piping, fire detector or alarm or extinguisher, burglary detector or alarm), (b..) change, reroute, wiring (etc.), (c..) remove wiring (etc.), (d..) change attributes of the above (wiring, piping, fire detector or alarm or extinguisher, burglary detector or alarm), (..) etc. All of the above are wrt. some sub-entities of some building, etc.

[7]We shall, inter alia, use the term 'operation' in lieu of the term 'function'.

[8]— like inserting a door, removing a door, changing the attributes of the door [access rights], etc.

Functions on (3) Potentially Movable Equipment

Examples of functions that apply to entities of class potentially movable equipment are: (a) introduce (i.e., "create") such equipment, including placing it at some location, (b) moving mobile equipment from one location to another, (c) removing mobile equipment, (d) applying, for example a laptop or a main frame to a program, that is, invoking an IT Service, (e) changing attributes of mobile equipment, like installing, upgrading, or removing software or data, (..) etc.

Functions on (4) Persons

Examples of functions that apply to entities of class person are: (a..) hire, transfer, lay off or fire a staff, (b..) change attributes of a staff person: promote, demote, salary change, authorisation rights (privileges), etc., (c..) review or evaluate staff performance, (d..) allow a non-staff person to be admitted to a building or a room, or to perform some IT functions, etc., (..) etc.

Functions on (5) Registers

Examples of functions that apply to entities of class register are: (a) create a register, (b) update a register: (b.1) record the occurrence of a desirable or undesirable event, (b.2) evaluate a recorded event and so annotate the register, (b..) etc. (c) copy (part of) a register and (d) destroy a register.

Discussion I

As first presented above, functions are seen as mathematical abstractions. To apply a function to its arguments and obtain a result takes no time — time is not an issue. But in a real world performing the kind of functions that are exemplified above does take time. And, as presented above functions are "functional", that is, they are not like procedures or subroutines of conventional, imperative programming languages like Java and C#, "our" functions do not act upon storage variables and change the values of these — they are mathematical functions. To prepare for a treatment of functions whose application takes time and may be understood as "altering" some input argument we now introduce a notion of state.

States

One may consider any composition of entities as a state. We usually make the pragmatic distinction between contexts and states. Contexts are compositions of simple entities whose value change less often and states are compositions of simple entities whose value change more often. Contexts provide a setting for activities, while states are the target of these activities.

State-changing Functions

We say that state-changing functions when invoked are actions. Actions may change the state and may "return" a value to the actor, see next, who invokes (triggers, ...) the function.

Discussion II

When functions are applied, then they are usually applied at some location, and at some time, by some actor, a person or a machine, or whose invocation is triggered by some event — we may say that some "outside" agent "is at play" — and maybe with some arguments provided by the actor who also designates the context and state entities on, or to which the function is to be applied.

So actors are either persons, or are machines, or are "outside" agents. We shall now treat the notion of events.

9.6.3 Events

General

Events "happen". They "occur". They take place instantaneously. They are like "communications" from an "outside". They are not functions — although they may, "mysteriously" trigger the invocation of functions; and they are not entities — although they may convey values. Later, when dealing with behaviours, we shall treat events as (synchronisations and) communications between behaviours — including the, or an, "external" behaviour. The notion of event is closely related to the notion of behaviour.

Examples of IT System Events

We give a number of examples of undesirable IT System events.

(1) *Events related to the physical plant:* (a) earthquakes, (b) typhoons, (c) fire, (d) break-in by un-authorised persons, (..) etc.

(2) *Events related to physical plant installations:* (a) electricity power break-down, (b) broken water pipes, (c) vandalism to communication cables, (d) break-down of fire detector and fire extinguisher, (e) break-down of burglary detection and alarm system, (..) etc.

(3) *Events related to potentially movable equipments:* (a) un-authorised access to a mainframe or laptop, (b) disappearance (theft or otherwise) of a laptop or a data medium, (c) sudden appearance in an unexpected place of a laptop, (..) etc.

(4) *Events related to persons:* (a) un-authorised access to a room (of a building) by some person, (b) un-authorised access to a mainframe or laptop

by some person, (c) loss (theft or otherwise) of access entry card or password, (..) etc.

(5) *Events related to registers:* (a) the entries of a register are up for the annual review, (b) un-authorised access to (edit of, etc.) a register, (..) etc.

Event Identifier

By an event identifier we shall understand some unique way of identifying one set of events from another set. Examples of event identifiers: typhoon, earthquake, power break down, fire in building #A, water pipe breakage in building #B, etc.

Event Alphabet

By an event alphabet we shall understand a set of event identifiers. An example of an event alphabet is {typhoon, earthquake, power break down, fire in building #A, water pipe breakage in building #B, ...}

Synchronisation and Communication

We shall consider events as relating two (let us assume simple) behaviours where simple behaviours are seen as sequences of actions and events, in any order, and where events synchronise the progress of these two behaviours while possibly also communicating values between them.

Discussion

The above represent a greatly simplified notion of events (and behaviours). It will do for all of our present modelling. It is based on the process concept of CSP: Communicating Sequential Processes. Other notions of events and behaviours could have been used for example the Petri Net or the π-Calculus notion of processes.

9.6.4 Behaviours

General

By a behaviour we shall — somewhat circularly — understand a sequence of sets of actions and events.

Simple, Single-thread Behaviours

By a simple behaviour we shall understand a (linear) sequence of actions and events.

Composite, Multiple-thread Behaviours

By a composite behaviour we shall understand a set of simple or composite behaviours.

Communicating Behaviours

By a pair of communicating behaviours we shall understand two simple or composite behaviours such that an event in one of these two identifies an event in the other of these two.

Communications

Let

$$c_i :< a_{i_1}, ..., e_{ij}, ..., a_{i_m} >$$

and

$$c_j :< a_{j_1}, ..., e_{ij}, ..., a_{j_n} >$$

describe two behaviours $(\mathcal{C}_i, \mathcal{C}_j)$. The a_{i_k}'s and a_{k_ℓ}'s describe actions $(\mathcal{A}_{i_k}, \mathcal{A}_{j_\ell})$ internal to $\mathcal{C}_i$, and $\mathcal{C}_j$, respectively. e_{ij} describes an event $\mathcal{E}_{ij}$. Since e_{ij} occurs in both c_i and c_j event $\mathcal{E}_{ij}$ may occur in both $\mathcal{C}_i$ and $\mathcal{C}_j$. If $\mathcal{E}_{ij}$ occurs in both $\mathcal{C}_i$ and $\mathcal{C}_j$, then it occurs simultaneously in both behaviours.

Internal Communications

Let k designate a channel, e an expression, v an identifier, and let e_{ij} be of the "paired" forms

in ci: k!e, in cj: **let** v = k? **in** ... **end**

then, when event $\mathcal{E}_{ij}$ occurs between behaviours $\mathcal{C}_i$, $\mathcal{C}_j$, the following happens: e is evaluated in $\mathcal{C}_i$, the value is bound to v in $\mathcal{C}_j$, and the two behaviours proceed. We say that the *two behaviours* have been *synchronised* and that *a value* has been *communicated* from one to the other. We say that the communication has been internal between the two behaviours.

External Communications

If either behaviour $\mathcal{C}_i$ or $\mathcal{C}_j$ has been left out of our description (i.e., c_i or c_j has not been given), then we say that the communication has been external between the described behaviour and an "external world".

Discussion

We have presented a capsule view of behaviours (and events). There is more, much more, to say, but this shall suffice. The view presented is basically that of Hoare's CSP: Communicating Sequential Processes. It is the CSP view of behaviours that we shall assume in the following.

9.6.5 Discussion

We have presented a view of entities, functions, events and behaviours. We take these four concepts as forming, one could say, one coherent set of aspects of an ontology of descriptions. We shall next take a brief look at other sets of aspects of an ontology of descriptions.

9.7 A Formal Model of IT Systems

9.7.1 Ω: The "Grand" State

In the modelling of all observable entity phenomena: simple entities, operations (i.e., functions), events and behaviours, we make use of the notion of a "grand state" $\omega : \Omega$. The grand state includes basically all observable simple entities. We name by Ω the type of all "grand states". We usually name by ω a value in Ω, i.e., a "grand state". Usually functions performed on, i.e., actions within the IT system being modelled are of either of the following signatures:

```
type
  Ω, ARG, VAL
value
  val_f: ARG → Ω → VAL
  int_f: ARG → Ω → Ω
  elab_f: ARG → Ω → Ω × VAL
```

That is, these functions are either **eval**uation functions observing, extracting or calculating (i.e., computing) a value of some ω, or **int**erpretation functions "changing", updating ω into ω', or are **elab**oration functions observing, extracting or calculating (i.e., computing) a value of some ω while "changing", updating ω — the latter is then called a "side-effect".

We model the grand state as consisting of several subsystems (one could call them components):

- Φ: the plant,
- Θ: the installations,
- $\Phi\Theta$: the plant and installations,
- Σ: movable equipment,
- Π: personnel and
- $\Re$: registers.

We now discuss these.

Formally we shall consider Ω to be a sort equipped with observers for at least each of the major sub-systems.

Since we shall be modelling the plant and the more-or-less fixed installations as one (highly structured "component") sub-system, of sort $\Phi\Theta$,

value
  obs_$\Phi\Theta$: $\Omega \to \Phi\Theta$
  obs_Σ: $\Omega \to \Sigma$
  obs_Π: $\Omega \to \Pi$
  obs_$\Re$: $\Omega \to \Re$

Predicates applicable to $\Phi\Theta$ can then be defined to discriminate between plant components (or sub-systems) and installations. The reason for modelling the two otherwise somewhat distinguished sub-systems is that the highly intricate structuring of installations (such as pipes, wires and cables) follows the similarly highly intricate structuring of the plant.

9.7.2 $\Phi\Theta$: The Plant and Installations

We shall develop our model of the plant + its installations by "slowly" unfolding a notion of system diagrams and system graphs. The system diagrams are very much like architectural drawings, i.e., building and floor plans, whereas system graphs are just that: graphs with nodes and edges. Nodes correspond to rooms (or an installation) of a building whereas edges correspond to access to rooms (i.e., a door or a barrier) or access to installations (a water pipe crane, an electric wire adaptor, a sewage pipe drainage, etc.).

So our "pedantic unfolding" of how buildings are composed from rooms, and of how rooms may be considered "embedded" in "larger" rooms, or, rather, embedded in sub-parts of a building, e.g., floors or (east, center, north, etc.) wings of a building that pedantic development starts from basic, atomic entities and proceeds via their composition, to the general composition of composite entities (i.e., nodes) and the accesses from nodes (i.e., rooms, etc.) to nodes (i.e., adjacent rooms, etc.).

We develop the model for the plant + its installations by first developing two graphic languages: a language of system diagrams, and a language of system graphs. There are not many step in their development, but they are, as we have now said several times, a bit pedantic, so please bear with us.

Simple Composition Rules

41. **A simple atomic plant:**
    The simplest plant is one consisting of just one atomic component. See Fig. 9.1 on the following page.
    The sharp edged box (rectangle) in the system diagram is reminiscent of how one might draw a layout of a building, or a map of a collection

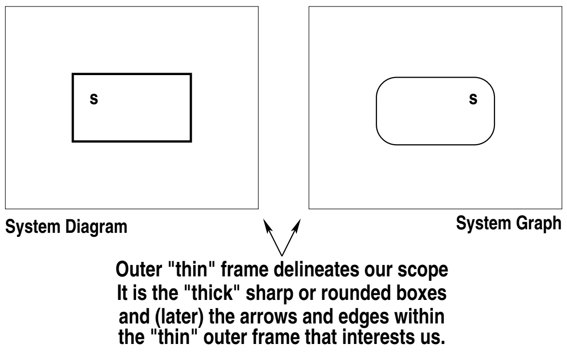

s
s
System Diagram
System Graph
Outer "thin" frame delineates our scope
It is the "thick" sharp or rounded boxes
and (later) the arrows and edges within
the "thin" outer frame that interests us.

Fig. 9.1. A simple atomic plant

of buildings (in this case only one), or a machine, or, for that matter a single human. The rounded corner box in the corresponding system graph is going to be our graphical notation for plants: a plant, a component, "is" a node.

42. **A simple composite, embedded plant:**
The simplest composite plant reflecting embeddedness consists of one composite component, s, which then has one simple atomic component, s_e, embedded within s.

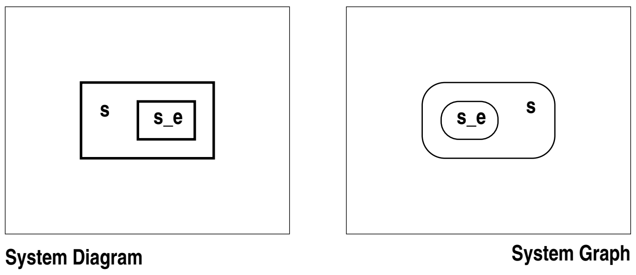


Fig. 9.2. A simple composite, embedded plant

Now we have a node within a node, as in hyper-graphs. Plant s appears not to be able to "access" sub-plant s_e — whatever we mean by 'access'. (We will elaborate on that later, but you can think of access as meaning: for a properly authorised human to "use" a plant, being able to perform the (one or more) function(s) that the plant may offer, being able to read, update, copy, etc., the information that the plant "embodies" or the functions it offer.)

43. **A simple composite, embedded plant with access:**

The simple embedded plant of Item 42 on the preceding page did not show the possibility of accessing sub-plant s_e from plant s. We modify Fig. 9.2 on the facing page into Fig. 9.3[A].

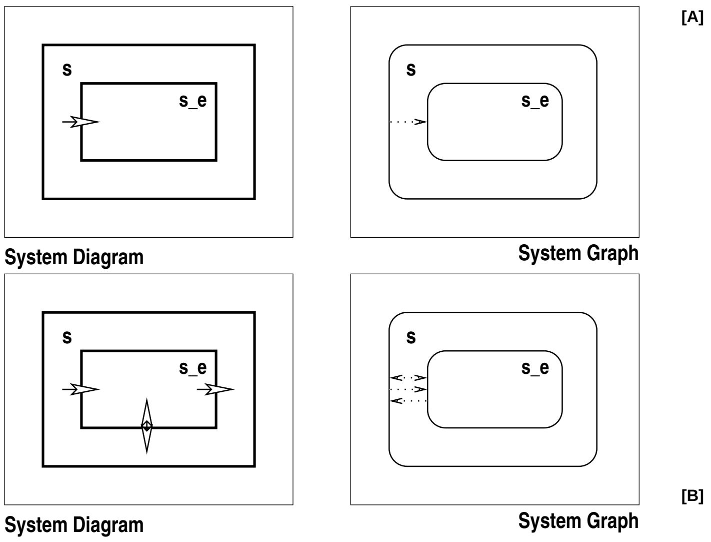


Fig. 9.3. A simple composite, embedded plant: [A] one access; [B] three accesses

We have in the system diagram of Fig. 9.3[A] (left) shown an "arrow" (mostly, really, an arrow-head) to indicate that one can "access" embedded component from "outer" components. The access is suggested to be directional, one way, in one direction, or in the opposite direction, or two-way, in both directions. The system diagram "arrow" is "dangling": it is not shown from where "within" plant s the arrow emanates and it is not shown to where "within" sub-plant s_e the arrow is incident. In the system graph of Fig. 9.3[A] (right) we show the "dangling arrow" notation of Fig. 9.3[A] (left): the system graph edges from nodes to sub-nodes are dashed.

In Fig. 9.3[B] we show three possibilities of access.

44. **A simple composite, disjoint plant:**
The simplest composite, non-embedded plant has the plant s consist of two adjacent, that is, two disjoint sub-plants si and sj. See Fig. 9.4 on the following page.
Sub-plants si and sj appear not be accessible from plant s and it also appears that one cannot access either of the sub-plants from the other.
45. **A simple composite, adjacent plant:**
We can juxtapose two disjoint sub-plants si and sj "right" next to one another, that is, adjacently, "sharing" some "wall". See Fig. 9.5.

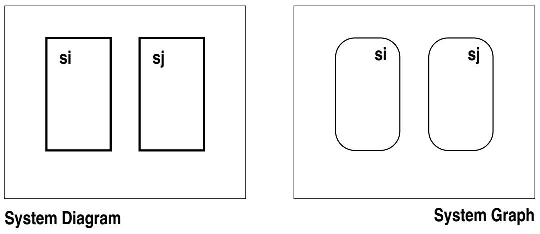


Fig. 9.4. A simple composite, disjoint plant

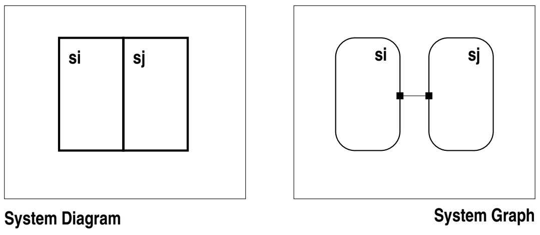


Fig. 9.5. A simple composite, adjacent plant

We shall soon see what that 'wall' means, that is, makes possible. As it stands now, in Fig. 9.5, there seems not to be access between the two sub-plants. Note the straight line between nodes *si* and *sj* of the system graph. It models the wall, i.e., adjacency (not access).

46. **A simple composite, disjoint and adjacent plant:**
We "insert" some access arrows in the wall of Fig. 9.5 to contain Fig. 9.6.

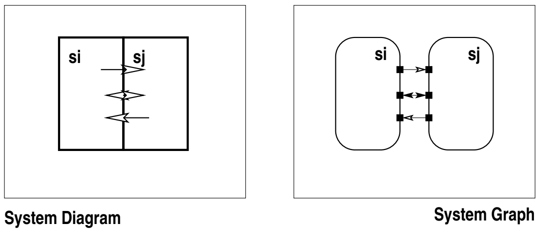


Fig. 9.6. A simple composite, disjoint and adjacent plant with access

The meaning is that now *si* and *sj* can access one another. We need only have shown one access arrow: either one-way from *si* to *sj*, or two-way *sj* between *si*, or one-way from *sj* to *si* — as shown, top-to-bottom in Fig. 9.6 on the preceding page. The (three) un-dotted (i.e., straight line) arrows of the system graph designate both adjacency and access direction.

47. **Embedded Adjacent Sub-plants with Access:**
Let us consider, see Fig. 9.7[A], a sub-plant s_{i_j} of a sub-plant s_i of plant s such that “activities” of s can directly access the “inner” sub-plant s_{i_j}. In the system diagram we show the sub-plant s_{i_j} “sharing” a wall” with sub-plant s_i, i.e., a wall between s and the two sub-plants (one, s_{i_j}, “within” the other, s_i).

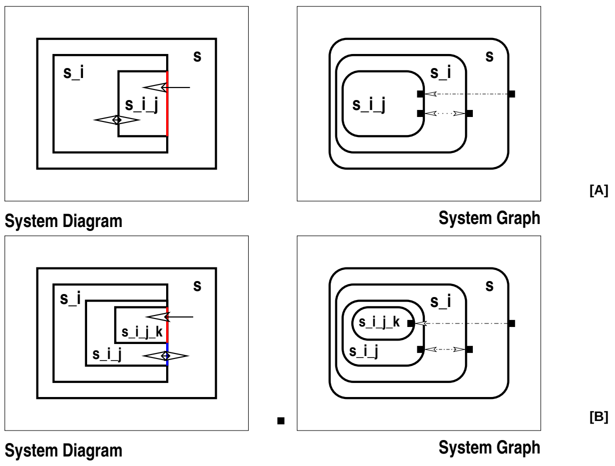


Fig. 9.7. [A] Doubly embedded plant, [B] triply embedded plant

In the system graph of Fig. 9.7[A] we show this not by “sharing” the contour of s_{i_j} with that of s_i but by a dash-dotted line from the contour of s through the contour of s_i to the contour of s_{i_j}.
Choosing this graphical rendition disambiguates any possible multiple interpretations as to which level of embedded sub-plants are being “connected”. See Fig. 9.7[B].

● ● ●

We have introduced the most basic rules for composing plants: embedding and juxtaposition. We have shown how one can transform a system diagram of boxes into a system graph of nodes. And we have introduced the most basic rules for designating access, that is, for composing (system diagram) component boxes and (system diagram) access arrows.

Generality of the Simple Composition Rules

There can be any number m of sub-plants $s_{e_1}, s_{e_2}, \ldots, s_{e_m}$ embedded in a plant s, and there can be any number of juxtaposed (i.e., adjacent) sub-plants

$s_{a1}, s_{a2}, \ldots, s_{am}$ in a plant s. Finally there can be any number of accesses (i.e., access arrows) between a plant s and an embedded sub-plant s_i of s and between any two adjacent plants s_{ai} and s_{aj} — even multiple occurrence of the same kind. What that means we shall cover later.

Composite (Combined) Composition Rules

We now analyse combinations of embedding, juxtaposing and access.

48. **Access between embedded sub-plants of adjacent plants:**
    Let si and sj be two disjoint, but adjacent plants. See system diagram of Fig. 9.8. Let plant si_a be a sub-plant of si, and let si_{a_p} be a sub-plant of si_a. Similarly for sub-plant sj_x of sj. The system diagram of Fig. 9.8 now illustrates all possible (in this case two-way) accesses between the two adjacent plants and all their respective sub-sub-plant. (Figure 9.8 does not illustrate accesses from "outer" plants to embedded sub-plants of neither si nor sj. This is left as an exercise for the reader to draw: Both the system diagram and the corresponding system graph.)

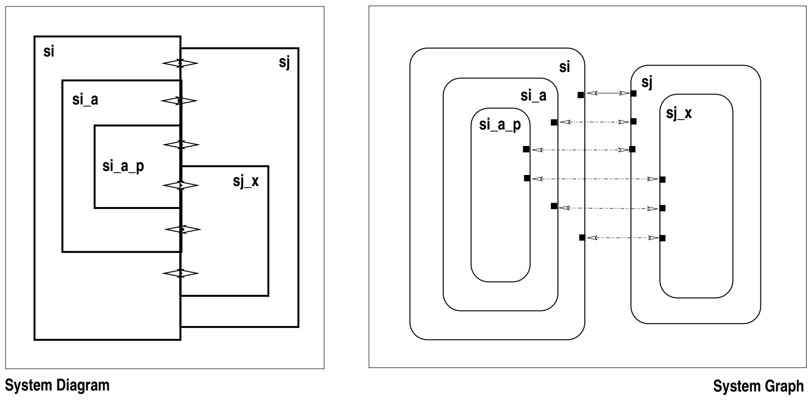


Fig. 9.8. Access paths

Note that the topmost edge from plant si to disjoint, but adjacent plant sj is a solid line two-way arrow. All other edges are "dash-dot" (– · – · – · –) two-way arrows. By an access path, a route, we mean a direct access that involves "transgressing" zero, one or more "walls", between plants. All of the above accesses are composite. We can model an access path as follows:

type
  AP = S × S
examples:

(si,sj), (sj,si), (si,sj_x), (sj_x,si), (sj,si_a), (sj_x,si_a_p)

Humans “transgress” access paths. Sometimes “transporting” plants. Each “transgression” amount to performing some function on the access.

49. **Access Routes:**
By an access route, r, we mean a sequence of one or more access paths such that if p_{k-1}, p_k is a pair of “adjacent” paths in r then the second state (s_i) of p_{k-1} is the same as the first state (s_j) of p_k, that is, rewriting r:

```
r: ⟨(s_1,s_2),(s_2,s_3),...,(s_j,s_j+1),(s_j+1,s_j_+2),...,(s_m−1,s_m)⟩
```

See Fig. 9.9.

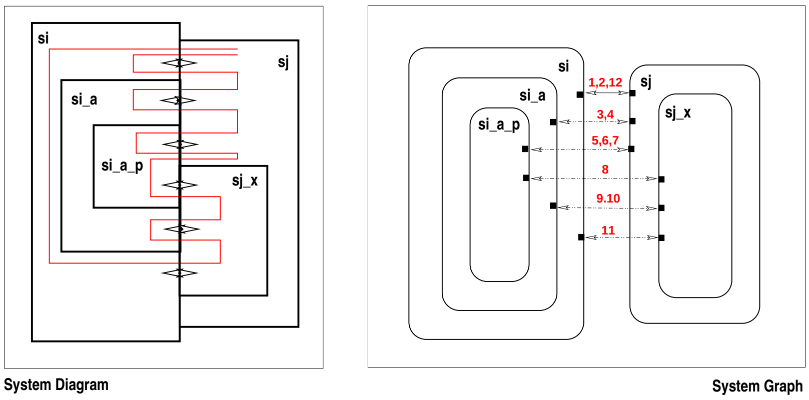


Fig. 9.9. An access *route*

The system diagram of Fig. 9.9 indicates the *route* while the system graph indicates the number of times the routes meanders its way through accesses (access points). Humans “travel” access routes. Sometimes “transporting” plants. ‘Travelling’ amounts to performing a sequence of functions on respective accesses.

Planar and Non-planar Graphs

You may have noticed that all our system graphs were shown as planar graphs. You may also have wondered about the two-dimensionality of our system diagrams. The plants that we deal with in humanly manifest physical plants, that is, plants of roads, terrain around buildings, buildings and their internal layout, equipment within buildings, the possible electrical of electronic (wired or wireless) communication “cabling”, etc., these plants and sub-plants are all three dimensional.

Is there a fourth dimension, or are there more than four dimensions? Is time a dimension? If the plants change their configurations of disjointness, adjacency or embeddedness, or if access paths change, is that something that is modelled in the time domain? We shall look at some of these issues now, and eventually at more of them. (Is it possible to eventually state that we have considered all such "dimensionality" issues?)

50. N **Adjacent Embedded Plants:**
Consider an n story building, floor stacked upon floor. Usually a staircase connects the floors. A system diagram would then show the building as the plant and the staircase plus n floors as $n+1$ sub-plants. To get (i.e., to "access") from one floor to another one would have to pass through two accesses, each access being between a floor and the staircase. We leave the design of the system diagram and the system graph as an exercise. Consider instead a building where for every floor there is a "bay" with a staircase to all the other floors such that only one access (one door) is necessary between any distinct pair of floors.

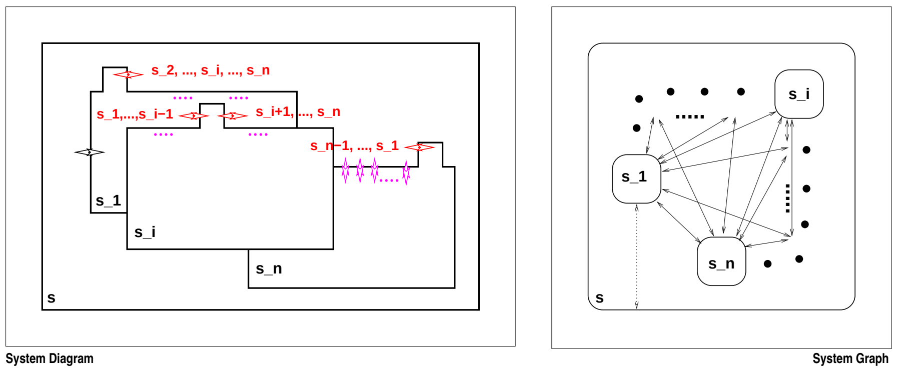


Fig. 9.10. n Adjacent embedded plants

The system diagram considers the building as "separate" from the floors and considers the floor as disjoint sub-plants with only floor #1 being adjacent to the building (i.e., its entrance hall).

The above construction shows that any three dimensional plant s can have any arbitrary number n of embedded sub-plants s_i of the plant be adjacent sub-plants. The two dimensional system diagram is inductive, cf. the use of "overlapping" floors and induction (...), hence it is schematic. Let us say that each horisontal floor plan is along dimensions X and Y, and that a vertical cut, along a vertical axis Z, through the building is along either dimensions X and ZS or Y and Z. Such a set of architectural plans or a proper isometric or perspective drawing of the building (or a set of such drawings together with floor plan drawings and a proper interpretation of those ensembles of drawings, would perhaps be the more proper way to show a three dimensional system diagram. There are similarly special diagrammatic languages for cabling (wiring), mechanical assembly, etc.

Conjecture

The essence of it all is that we can always map such three dimensional system diagrams onto a two dimensional system graph (albeit most often not a planar graph).

Examples of Plant Modelling

51. **Power supply cabling of machines:** See Fig. 9.11.

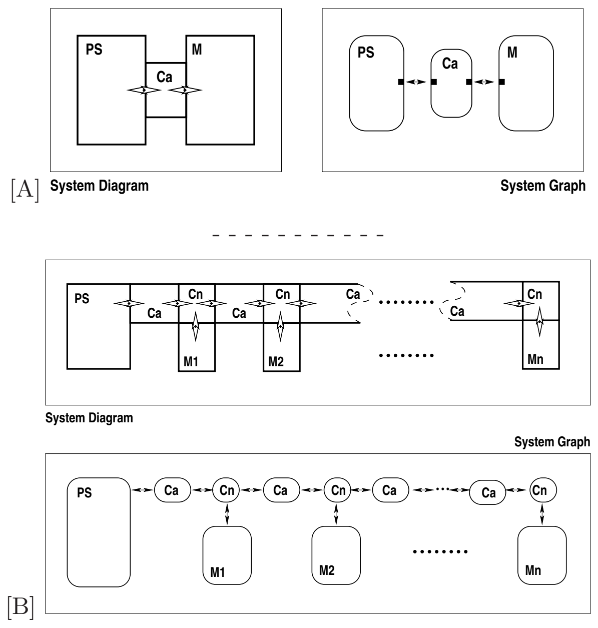


Fig. 9.11. Power supply cabling of machines: [A] One machine, [B] n Machines

9.7.3 A Formal Model of $\Phi\Theta$

The Syntax

52. Nodes have simple names (further undefined), and atomic (basic) components are further undefined.

53. A system graph G has a name, n:N, and otherwise consists of a basis part, b:B, a set of zero, one or more (disjoint) components (nodes), cs:C-**set**, and a set of zero, one or more edges, es:E-**set**.
54. A component, c:C, has a name, and consists of a basis part and a set of zero, one or mode components (ci) embedded in the defining component (c).
55. An edge connects two nodes, n1,n2:N, and has a set of one or more distinct access specifications, a:A.
56. An access specification identifies an access direction and a set of operations.
57. A direction is either from the first to the second node (n1,n2), or the reverse, or is two-way. If the direction is adj_wall then there is no access but the two nodes possess an adjacent wall (and the operation set is empty).
58. An operation is either a move, or a read, or a perform, or some other operation.
59. These operations are left further undefined.

```
type
52  N, B
53  G == mkG(sn:N,sb:B,cs:C-set,es:E-set)
54  C == mkC(sn:N,sb:B,scs:C-set)
55  E == mkE(s1:N,s2:N,sks:A-set)
56  A == mkA(sd:D,sas:O-set)
57  D == fst_snd | snd_fst | 2_way | adj_wall
58  O == Move | Read | Perform | ...
59  Move, Read, Perform, ...
```

The Syntactical Constraints

52. All nodes of a graph, whether embedded or juxtaposed, have distinct names.

52,54. To paraphrase the above: Any two disjoint components, si and sj, of the components $\{s1, s2, \ldots, si, \ldots, sj, \ldots, sm\}$ of a plant s have distinct names and these are distinct from the name of s. Any component si is embedded in s and any two components si and sj, are disjoint (within s).

54. A component, i.e., a plant (or sub-plant — which is the same), $si : S$, has a name, n_{si}.

54. If a plant, s of name n_s, consists of only one component, $s1 : S$, of name n_{s1}, then their names, n_s and n_{s1} will be made be different).

60. We decide to secure distinct of nodes by mandating that names, n_{i_1}, n_{i_2}, $\ldots$, n_{i_m}, of nodes of immediate sub-plants of a plant named n_i are distinct and that the name n_i can be uniquely “extracted” from each n_{i_j} for all j in the interval $1..m$.

60. That is, think of the immediate components, s_{i_1}, s_{i_2}, …, s_{i_m}, of s as being ordered as just listed, and the names being a bijection function, η of the name of the plant and the name index of the sub-plant:

```
type
    Idx
value
    η: N × Idx ↔ N
    η⁻¹: N ↔ N × Idx
axiom
    ∀ n:N, i:Idx • η∘⁻¹η(n,i) ≡ (n,i), i.e.: η⁻¹∘η ≡  λx.x ≡ η∘η⁻¹
```

55. An edge connects two nodes, n_α, and n_β. These nodes must be distinct. The two nodes stand in either of the following relations to one another:
    (a) Either they are of disjoint but (of course) adjacent plants (otherwise why have the edge unless to express adjacency?),
    (b) or one node is of a sub-plant embedded one or more levels within another (the "outer, surrounding") plant,
    (c) or they are sub-plant nodes, n_α, n_β, each embedded (one or more levels, i.e., $\ell\#a$, $\ell\#b$), within disjoint and adjacent plant n_i, n_j. The α, β indexes typically would be: $i_{i1_{i2_{\ldots_{i\ell\#a}}}}$ respectively $j_{j1_{j2_{\ldots_{j\ell\#b}}}}$. (The number of … in these past two index expressions are $\ell\#a - 3$, respectively $\ell\#b - 3$.)

Formalised Graph Well-formedness

```
value
    wf_G: G → Bool
    wf_G(mkG(n,_,cs,es)) ≡
        let ns = all_nodes(cs) in wf_Cs(n,cs) ∧ wf_Es({n}∪ ns,es) end

    wf_Cs: N × C-set → Bool
    wf_Cs(n,cs)
        let ns = {sn(c)|c:C•c ∈ cs} in
        let ixs={i|i:Idx,n′:N•n′ ∈ ns∧let (n″,j):(N×Idx)•η⁻¹(n′) in i=j end} in
        card cs = card ns = card ixs ∧ n ∉ ns ∧
        ∀ mkC(n′,_,cs′):C • mkC(n′,_,cs′) ∈ cs ⇒ wf_Cs(n′,cs′) end end

    wf_Es: N-set × E-set → Bool
    wf_Es(ns,es) ≡ ∀ mkE(n,n′,_):E • mkE(n,n′,_) ∈ es ⇒ {n,n′}⊆ns
```

Mereological Operations on *ΦΘ*

By a syntactic operation on a plant we mean an operation which changes its hypergraph representation. Humans perform such operations. Some operations on certain components or entities require authorisation.

61. Plants change dynamically.
62. One may
    (a) **adjoin a node to a plant** with the new node being disjoint to all other nodes of the plant,
    (b) **embed a node in a plant**, with the new node being immediately contained in some node of the plant,
    (c) **connect two nodes**, whether disjoint or arbitrarily contained.
    (d) **sever, i.e., remove, the edge between two nodes**, whether disjoint or arbitrarily contained.

    Etcetera.

We leave it as an exercise to formalise these operations.

Attribute Operations on $\Phi\Theta$

By an attribute operation on a plant we mean an operation which changes changes the attributes associated with nodes and edges. Humans (and foreseeable or unforeseen non-human events) perform such operations. Some operations on certain components or entities require authorisation.
We leave it as an exercise to conceive of and narrate and formalise such operations.

Semantic Operations on $\Phi\Theta$

By a semantics operation on a plant we mean an operation which invokes a function to be applied to the plant. Humans (and foreseeable or unforeseen non-human events) perform such operations. Some operations on certain components or entities require authorisation.
We leave it as an exercise to conceive of and narrate and formalise such operations.

9.7.4 $\Sigma\Pi\Re$: Movable Resources

Simple Entities

The movable resources, to recall, include:

- laptops,
- people and
- registers.

Modelling

We suggest to model a movable resource as a node in the system graph. At any one point two or more such nodes may be connected, i.e., are accessing one another, and/or are accessing nodes of the fixed (the $\phi\theta$ graph).

Operations

With movable resources we can associate a number of operations: (i) on laptops: (i.1–.2) (de-)register, (i.3) move, (i.4) query, (i.5–.6) (dis-)connect and (i.7) use; (ii) on people: (ii.1–.2) (de-)register, (ii.3) query, (ii.4–.5) (re) assign authorisations (ii.6) move, and (iii) on registers: (iii.1) (de-)catalog, (iii.2) update, (iii.3) query, (iii.4) search, and (iii.5) copy.

We leave it as an exercise to narrate and formalise these and other operations.

9.7.5 Discussion

We have sketched how one may narrate and formalise the simple entities and operations of an IT system. We have, and shall not cover the issue of events and behaviours of an IT system. But once a model of simple entities and operations is being established — true to scale, that is, with all the "bells & whistles" — one can then also model events and behaviours.

The simple entities, the operations, the events and the behaviours are all identifiable.

Their names must reflect the names used in the formal predicates of Sect. 9.5 (on Pages 249–250). These names identify IT System resources that are spatially related to one another as reflected in the $\Phi\Theta$ and $\Sigma\Pi\Re$. Each identifier of any phenomenon or concept can thus be mapped into its "position" in the system graph as modelled in Sect. 9.7.

9.8 A Formal Modal of IT Security Code of Practice

Very preliminary remarks: We model the "code[s] of practice" as well-formed formula (*wff*) in a first order predicate calculus. The ground (mostly non-Boolean valued) terms denote entities in Ω. Predicate symbols denote predicates as we identified them in the logical explication of the code[s] of practice. Function symbols denote functions as we identified them in the logical explication of the code[s] of practice. Evaluation of a *wff* now take place in the context of some $\omega \in \Omega$.

9.8.1 Ψ_{Syntax}

We claim that the formal expressions of Sect. 9.5 on page 248 can all be expressed as well-formed formulas (*wff*s) in a predicate calculus. Below we present an (example annotated) abstract syntax for WWFs.

Since this is standard knowledge we make no further comments at this place, but refer to Sect. 9.5.5 (pages 178–180) of [31].

type	**examples**
Cn, Vn, Pn, Fn, Tn	cn, vn, fn, pn
Term = TId | TAp	
TId :: Vn | Cn	cn, vn
TAp :: (Fn|Pn) Term*	pn(t1,t2,...,tm), fn(t1,t2,...,tm)
Atom = Aid | AAp	
AId :: Vn | **true** | **false**	vn, **true**, **false**
AAp :: Pn Term*	pn(t1,t2,...,tm)
wwf:WWF = Atom|NWff|AWff|OWff|IWff|EWff|QWff	
NWff :: WFF	$\sim$wff
AWff :: WFF WFF	wff $\wedge$ wff$'$
OWff :: WFF WFF	wff $\vee$ wff$'$
IWff :: WFF WFF	wff $\Rightarrow$ wff$'$
EWff :: WFF WFF	wff = wff$'$
QWff :: Quan Vn Tn WFF	$\forall$ wff, $\exists$ wff$'$
Quan == all | exist	

9.8.2 $\Psi_{\text{Semantics}}$

By the semantics of a language, WFF, of *wff*s we mean an interpretation of the *wff*s in some context. The context assigns meaning to all symbols: The meaning of a predicate symbol is a predicate function of an arity commensurate with the number of terms following the predicate symbol. The meaning of a function symbol is a function of an arity commensurate with the number of terms following the function symbol. The meaning of a variable name is given by its typed binding in a quantified expression. The meaning of a constant name is given by the instantiation of a given plant (i.e., by some ω). The meaning of a type name is the set of all values of that type. And so forth.

All this is standard knowledge we make no further comments at this place, but refer to Sect. 9.5.7 (pages 181–184) of [31].

There is, however, a small technicality. It has to do with the context in which the *wff*s are interpreted. We normally see this context as a map from constant and variable identifiers, predicate and function symbols, etc., to their meaning. So, from the instantiated ω of the IT system being studied we prepare a context which maps all possible component and access (edge) names to their meaning (the designated physical artifact, including person, or the concept identified) — this was, amongst others, a reason for insisting on unique component and access names. The predicate and function symbols *wff*s of Sect. 9.5 on page 248 are likewise bound in an initial context to their meaning. Pls. observe that some of these predicate and function symbols may not denote computable functions — so we treat them as oracles.

The Context

```
type
  i$\Omega$ = (Cn|Vn|Pn|Fn|...) $\xrightarrow[m]{}$ VAL
  VAL = (VAL$^*$ $\overset{\sim}{\to}$ VAL) | Bool | Int | ...
value
  c$\omega$: $\Omega$ $\overset{\sim}{\to}$ i$\Omega$
```

The Meaning Functions

```
value
  M: WFF $\to$ i$\Omega$ $\to$ Bool
  M(wff)i$\omega$ $\equiv$
    case wff of
      mkNWff(wff$'$) $\to$ $\sim$M(wff$'$)i$\omega$,
      mkAWff(wff$'$,wff$''$) $\to$ M(wff$'$)i$\omega$ $\wedge$ M(wff$''$)i$\omega$,
      mkOWff(wff$'$,wff$''$) $\to$ M(wff$'$)i$\omega$ $\vee$ M(wff$''$)i$\omega$,
      mkIWff(wff$'$,wff$''$) $\to$ M(wff$'$)i$\omega$ $\Rightarrow$ M(wff$''$)i$\omega$,
      mkEWff(wff$'$,wff$''$) $\to$ M(wff$'$)i$\omega$ = M(wff$''$)i$\omega$,
      mkQWff(all,v,t,wff$''$) $\to$ $\forall$ u $\in$ i$\omega$(t) $\bullet$ M(wff$''$)(i$\omega$ $\dagger$ [v$\mapsto$u],
      mkQWff(exist,v,t,wff$''$) $\to$ $\exists$ u $\in$ i$\omega$(t) $\bullet$ M(wff$''$)(i$\omega$ $\dagger$ [v$\mapsto$u],
      _ $\to$ A(wff)i$\omega$
  A: Atom $\to$ i$\Omega$ $\to$ Bool
  A(mkAId(v))i$\omega$ $\equiv$ i$\omega$(v)
  A(mkAId(true))i$\omega$ $\equiv$ true
  A(mkAId(false))i$\omega$ $\equiv$ false

  A(mkAAp(nm,lt))i$\omega$ $\equiv$ i$\omega$(pn)($\langle$V(lt(i))i$\omega$|i in lt$\rangle$)

  V: Term $\to$ i$\Omega$ $\to$ VAL
  ...
```

The definition of the Term e*V*aluation function follows, as do the predicate and function symbol meanings, from the instantiated ω under study.

9.9 Making Use of The Formalisations

We have three formalisations:

- A formalisation of the IT System "Code of Practice" predicates of Sect. 9.5;
- a formalisation of IT Systems, Sect. 9.7; and
- a formalisation of "Code of Practice" predicate evaluation, Sect. 9.8

Now, how are we going to bring these three formalisations together.

- First we have to complete the formalisation of the IT System "Code of Practice" predicates, Sect. 9.5.
- Then we have to complete the formalisation of IT Systems, Sect. 9.7: that is, to define all the term functions and the auxiliar predicate used, but not otherwise defined in Sect. 9.5.
- The former have to be instantiated to a given, specific IT System. Section 9.9.1 will sketch how.
- The specific (client) IT System-instantiated predicates now serve as input to the predicate evaluator of Sect. 9.8. That "input" process will be discussed in Sect. 9.9.2.

9.9.1 Instatiating IT Security Predicates for Evaluation

The reader will have noticed that the predicates presented in Sect. 9.5 were generic: that is, applied to any IT System. The reader will also have noticed that there did not appear to be any explicit universal or existensial quantifiers in the predicates presented in Sect. 9.5. Some of the semantic functions of Sect. 9.8 were fully defined, notably the overall predicate evaluator, M, but the term evaluator, V, was not. To define that function would require that we first completed the formalisation of the IT System "Code of Practice" predicates, Sect. 9.5, identifying all the auxiliary functions whose arguments specifically designated fixed and movable resourses and that we also complete the formalisation of IT Systems, Sect. 9.7. Many of the predicates of Sect. 9.5, as was commented, are not computable, and many of the auxiliary functions will, most likely turn out to be not only non-deterministic but also necessarily underfined ("loosely defined").

Thus there is a "hidden" universal quantifier that ranges over all IT Systems, and, thus, for each of these, is bound to a specifically instantiated $\omega_\imath$ and, hence, to a specific $i\omega_\imath$. Now, each of the identified constant and variable names, $cn : CN, vn : VN$, ranges over that which they are intended to denote: specific physical facilities: plant, installations and movable resources. Thus the "hidden" universal or existensial quantifiers must be made specific and their range must be enunciated for every specific embedding. That is: any one instance of a predicate applies to a (cn or vn) value of $i\omega_\imath$. Thus the IT System "Code of Practice" predicates of Sect. 9.5, shall have to be instantiated for all resources immediately contained in that "layer" ("embedding"),

k, of $i\omega$. That value may, for example, be a room (i.e., a plant, etc.) which has embedded rooms (plants, etc.). Thus the IT System "Code of Practice" predicates of Sect. 9.5, shall have to be instantiated for these, layer $k+1$ embedded facilities, fixed or movable if applicable. And so forth.

9.9.2 Evaluation

Testing for IT Security Dynamically

Thus we can define a function M and apply it to any instantiated state $\omega_\imath$:

$$M(\mathit{wff}_\imath)(\omega_\imath)$$

where $\mathit{wff}_\imath$ is any conjugated ($\wedge$) and instantiated subset of "code[s] of practice". If the resulting value is `ff` the instantiated subset "code[s] of practice" have been violated. If the resulting value is `tt` the instantiated subset "code[s] of practice" have not been violated.

Testing for IT Security Statically

If we evaluate

$$M(\mathit{wff}_\imath)(\omega_\imath)$$

for any (valid) instantiated $\omega_\imath$ then we are, in a sense testing whether the given set of instantiated $\mathit{wff}_\imath$s constitutes a relative complete and consistent "code of practice".

A Caveat

Of course: we cannot mechanically evaluate $M(\mathit{wff\_\imath})(\omega_\imath)$. Most of the predicates and term functions mentioned in a formalisation of the ISO Code of Practice are not computable. So what do we do ?

Interactive Evaluation

One can devise the evaluation process such that whenever the evaluator encounters a partially defined function then the process interacts with IT System Security stakeholders present the state of evaluation to them and requests their advice as to which course the ongoing evaluation should take. This interactive evaluation process can be refined to allow for "search trees" of evaluation: that is, the interactive evaluator keeps track of non-deterministic and multiple input choices, and pursues these in turn.

Conformance

How do we know that the ISO Code of Practice instantiation is commensurate with the un-instantiated version of the ISO Code of Practice axioms ? Well, since the un-instantiated, formal version is not computable we shall have to prove the correctness of the refinement, i.e., the instantiations. And that proof cannot be mechanised. It is a classical mathematical (logic) proof: a lot of brain-power and a lot of writing.

9.10 Closing

9.10.1 What is IT Security ?

The International ISO/IEC Standard 17799: Information technology: security techniques — code of practice for information security management does not provide any briefer characterisation of what is meant by IT System security than its approximately 135 pages of detailed, operational description.

This may be acceptable. An IT system is a very operational "affair". It embodies few abstractions. What we would like to see in the direction of a characterisation of "what IT System Security" is, is illustrated next.

When is an IT System Secure ?

An IT System is secure when an un-authorised user, after periods of trying to "enter" the system (1) *cannot find out what it is doing (i.e., protecting),* (2) *cannot find out how it is doing (whatever it is doing),* (3) *and does not know that she, the user, does not know* (1–2) *!*

The third part is introspective[9] wrt. the first two parts.

This "definition" is, of course, highly debatable. But it makes a point: namely that one cannot pursue the issue of IT System Security without having a proper, not too long, and certainly not an approximately 135 page long, and definitely not an implicit "definition" such as the ISO/IEC Standard 17799. One must do far better than that. Whether our definition is a feasible one, or, with some preamble definitions could be made feasible we shall leave as an open question

9.10.2 What Have We Achieved?

We have "achieved" the following: (i) indicated, while providing formal "arguments" for, how IT System Codes of Practice rules could be formalised; (ii) indicated, while providing formal "arguments" for, how the context in which these IT System Codes of Practice rules must be understood; and (iii) indicated, while providing formal "arguments" for, how the [(i)] rules can be interpreted in their contexts [(ii)].

[9]cf. introspective logic of belief ...

9.10.3 Issues of Contention

But the "formalised indications" (of Sect. 9.10.2) are merely sketches. And not all issues have been resolved. There are many "dangling", i.e., unresolved issues: (i) completion of formalisation of IT System Codes of Practice; (ii) completion of formalisation of IT System facilities and resources; (iii) completion of semantic interpretation functions; and (iv) clarification of exactly how the free identifiers in the formalisation of the predicates and the auxiliary term functions of (i) shall be bound to the entities of (ii).

9.10.4 Future Work

The (i–iv) of the previous paragraph (Sect. 9.10.3) points out to a serious experimental research activity. It is hoped that work on (i–iv) (Sect. 9.10.3) will lead to the identification of a number of "lifted" predicates and functions that can vastly simply the masses of the formal predicates, (i), that are now so detailed and specific. Let us indicate what we mean by a first set of "lifted" predicates (we refer to Sect. 9.4): Chapters 9 and 10 of the ISO Standard appears to address very similar issues: Chap. 9 mostly related to computers and Chap. 10 mostly related to communications. (One could imagine that IBMs representative in the IT System Security effort had focused their contributions wrt. Chap. 9 and that CISCO's representative in the IT System Security effort had focused their contributions wrt. Chap. 10, respectively.) The technical/scientific (read: engineering/research) questions is therefore: would it be possible and beneficial to "lift" a number of relevant predicates of Chaps. 9 and 10 such that the specific predicates of these two chapters could be simpler expressed in terms of a number of parametrised predicates and auxiliary term functions. These parametrised predicates and auxiliary term functions are what we mean by "lifting".

Similar R&D issues can be raised wrt. a great number of 'Code of Practice' predicates and auxiliary term functions from usually two or more chapters.

In other words: Quite, I think, an exciting PhD topic!

10

A Family of Script Languages Licenses and Contracts — Incomplete Sketch[1]

Drs. Arimoto Yasuhito, Chen Xiaoyi and Xiang Jianwen were co-partners in this study

Caveat

This chapter is incomplete. Its basis, [62], is even more so. We leave a number of formal semantic function definitions undefined.

Summary

Classical digital rights license languages, [3, 9, 10, 72, 74–76, 113, 117, 141, 154, 166, 167, 177, 188, 189, 206, 207, 220] applied to the electronic "downloading", payment and rendering (playing) of artistic works (for example music, literature readings and movies). In this chapter we generalise such applications languages and we extend the concept of licensing to also cover work authorisation (work commitment and promises) in health care and in public government. The digital works for these two new application domains are patient medical records, public government documents (Sects. 10.2–10.4.3) and bus transport contracts (Sect. 10.6).

Digital rights licensing for artistic works seeks to safeguard against piracy and to ensure proper payments for the rights to render these works. Health care and public government license languages seek to ensure transparent and professional (accurate and timely) health care, respectively 'good governance'. Bus transport contract languages seeks to ensure timely and reliable bus services by an evolving set of transport companies.

Proper mathematical definition of licensing languages seeks to ensure smooth and correct computerised management of licenses and contracts.

In this chapter we shall motivate and exemplify four license languages, the pragmatics and syntax of four (Sects. 10.2–10.6) of them as well as the formal semantics of one of them (Sect. 10.6).

[1]This is an edited version of [62].
Section 10.6 (Pages 309–326) is new and authored only by Dines Bjørner.

10.1 Introduction

10.1.1 Computing Science cum Programming Methodology

- This chapter is not about [so-called Theoretical] Computer Science:
  - ⋆ The study & knowledge of concepts that can ∃ “inside” computers.
  - ⋆ Establishing computational models.
  - ⋆ Proving foundational lemmas.
- This chapter is about Computing Science
  - ⋆ The study & knowledge of how to construct “those” things !
  - ⋆ No proving foundational lemmas as in TCS.
  - ⋆ Instead establishing method principles, techniques and tools
  - ⋆ for formal specification and design calculi,
  - ⋆ and verifying, model checking and formally testing properties.

10.1.2 Caveats

This document constitutes a comprehended set of R&D development notes. It is not a report, let alone a JAIST report, and it is certainly not a publishable science and/or technology paper. This document is to serve as a basis for further work on the design, pragmatics, semantics and syntax of license languages, for upcoming work on understanding permissions and obligations, for upcoming work on studying possible understandings of license languages in terms of game theory, transaction costs, and possibly other issues such as technological feasibilities. This document is grossly incomplete.

We paraphrase the above. We do so since it has shown difficult for some to understand that this is not a paper anywhere close to submission for publications, let alone an internal JAIST report. To repeat: these are working notes. They are being “constantly” revised[2].

The formal semantics given “late” in the chapter (Sect. 10.6) is a standard, near “classical” way of (i) securing that the author of that formalisation has understood the design of the language. (ii) That CSP + RSL[3] definition can be used to write users reference manuals for constructing, issuing and acting upon licenses, and (ii) as a basis for implementing trustworthy interpreters for licenses and contracts and for license contract uses; that is for possibly provably correct implement a distributed license and contract management (monitoring and control) system.

If you are looking for “deeper” results, results that span any family of license languages adhering to the basic semantic principles developed in this document, then this is not the document to read. Well, you had better first read this document, or the reports and paper(s) that are planned to emanate

[2]Well, they have not been worked on since late 2006. I hope, during the summer of 2009, to be able to completely revise this chapter into a publishable paper.

[3]`CSP`: [137, 138, 218, 222], `RSL`: [31–33, 44, 101, 104, 106]

from this document. Then you may have to do the research that may lead to generic results. It is to be expected from such theoretical computer science work that a mathematical notation — invented explicitly for the purpose — will then "redefine" a suitably commensurate (congruent) and perhaps "vastly" generic sub-language, and the desired generic results are then proved to hold of that special notation "semantics".

10.1.3 On Licenses

License:
a right or permission granted in accordance with law by a competent authority
to engage in some business or occupation,
to do some act, or to engage in some transaction
which but for such license would be unlawful

Merriam Webster Online [232]

The concepts of licenses and licensing express relations between (i) *actors* (licensors (the authority) and licensees), (ii) *entities* (artistic works, hospital patients, public administration, citizen documents) and bus transport contracts and (iii) *functions* (on entities), and as performed by actors. By issuing a license to a licensee, a licensor wishes to express and enforce certain permissions and obligations: which functions on which entities the licensee is allowed (is licensed, is permitted) to perform. In this chapter we shall consider four[4] kinds of entities: (i) digital recordings of artistic and intellectual nature: music, movies, readings ("audio books"), and the like, (ii) patients in a hospital as represented also by their patient medical records, (iii) documents related to public government, and (iv) busses, time tables and road nets (of a bus transport system).

The *permissions* and *obligations* issues are, (1) for the owner (agent) of some intellectual property to be paid (an *obligation*) by users when they perform *permitted* operations (rendering, copying, editing, sub-licensing) on their works; (2) for the patient to be professionally treated — by medical staff who are basically *obliged* to try to cure the patient; (3) for public administrators and citizens to enjoy good governance: transparency in law making (national parliaments and local prefectures and city councils), in law enforcement (i.e., the daily administration of laws), and law interpretation (the judiciary) — by agents who are basically *obliged* to produce certain documents while being *permitted* to consult (i.e., read, perhaps copy) other documents; and (4) for bus passengers to enjoy reliable bus schedules — offered by bus transport companies on contract to, say public transport authorities and on sub-contract to other such bus transport companies where these transport companies are *obliged* to honour a contracted schedule.

[4]During our 2006 study we only studied and in [62] we only reported on the license languages related to (i–iii) below. The bus transport contract language, related to (iv), emerged during late 2008.

10.1.4 What Kind of Science Is This?

It is experimental computing science: The study and knowledge of how to design and construct software that is right, i.e., correct, and the right software, i.e., what the user wants. To study methods for getting the right software is interesting. To study methods for getting the software right is interesting. Domain development helps us getting the right software. Deriving requirements from domain descriptions likewise. Designing software from such requirements helps us get the software right. Understanding a domain and then designing license languages from such an understanding is new. We claim that computer-supported management of properly designed license languages is a hallmark of the e-Society.

10.1.5 What Kind of Contributions?

The experimental nature of the project being reported on is as follows: Postulate four domains. Describe these informally and formally. Postulate the possibility of license languages (LLs) that somehow relate to activities of respective domains. Design these – experimentally. Try discover similarities and differences between the four LLs ($\mathrm{LL}_{\mathsf{DRM}}$, $\mathrm{LL}_{\mathsf{HHLL}}$, $\mathrm{LL}_{\mathsf{PALL}}$, $\mathrm{CL}_{\mathsf{BUS}}$). Formalise the common aspects: $\mathcal{C}_{\mathsf{LL}}$. Formalise the three LLs — while trying to "parameterise" the $\mathcal{C}_{\mathsf{LL}}$ to achieve $\mathrm{LL}_{\mathsf{DRM}}$, $\mathrm{LL}_{\mathsf{HHLL}}$, $\mathrm{LL}_{\mathsf{PALL}}$, $\mathrm{CL}_{\mathsf{BUS}}$. This investigation is bound to tell us something, we hope.

The above outlines our ultimate goals. In reality, this chapter brings us only part of the way towards such a goal. To do the study as outlined in this section we first need complete the formal semantics of all the four languages.

10.2 Pragmatics of Three License Languages

- *By* **pragmatics** *we understand the study and practice of the factors that govern our choice of language in social interaction and the effects of our choice on others.*

In this section we shall rough-sketch-describe pragmatic aspects of the three domains of (1) production, distribution and consumption of artistic works, (2) the hospitalisation of patient, i.e., hospital health care and (3) the handling of law-based document in public government. The emphasis is on the pragmatics of the terms, i.e., the language used in these three domains. We leave the discussion of the bussing contract language till Sect. 10.6.

10.2.1 The Performing Arts: Producers and Consumers

The intrinsic entities of the performing arts are the artistic works: drama or opera performances, music performances, readings of poems, short stories, novels, or jokes, movies, documentaries, newsreels, etc. We shall limit our span to the scope of electronic renditions of these artistic works: videos, CDs or other. In this paper we shall not touch upon the technical issues of "downloading"(whether "streaming" or copying, or other). That and other issues should be analysed in [245].

Operations on Digital Works

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

License Agreement and Obligation

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

The Artistic Electronic Works: Two Assumptions

Two, related assumptions underlie the pragmatics of the electronics of the artistic works. The first assumption is that the format, the electronic representation of the artistic works is proprietary, that is, that the producer still owns that format. Either the format is publicly known or it is not, that is, it is somehow "secret". In either case we "derive" the second assumption (from the fulfilment of the first). The second assumption is that the consumer is not allowed to, or cannot operate[5] on the works by own means (software, machines). The second assumption implies that acceptance of a license results in the consumer receiving software that supports the consumer in performing all operations on licensed works, their copies and edited versions: rendering, copying, editing and sub-licensing.

[5]render, copy and edit

Protection of the Artistic Electronic Works

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

10.2.2 Hospital Health Care: Patients and Medical Staff

Citizens go to hospitals in order to be treated for some calamity (disease or other), and by doing so these citizens become patients. At hospitals patients, in a sense, issue a request to be treated with the aim of full or partial restitution. This request is directed at medical staff, that is, the patient authorises medical staff to perform a set of actions upon the patient. One could claim, as we shall, that the patient issues a license.

Hospital Health Care: Patients and Patient Medical Records

So patients and their attendant patient medical records (PMRs) are the main entities, the "works" of this domain. We shall treat them synonymously: PMRs as surrogates for patients. Typical actions on patients — and hence on PMRs — involve admitting patients, interviewing patients, analysing patients, diagnosing patients, planning treatment for patients, actually treating patients, and, under normal circumstance, to finally release patients.

Hospital Health Care: Medical Staff

Medical staff may request ('refer' to) other medical staff to perform some of these actions. One can conceive of describing action sequences (and 'referrals') in the form of hospitalisation (not treatment) plans. We shall call such scripts for licenses.

Professional Health Care

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

10.2.3 Public Government and the Citizens

The Three Branches of Government

By public government we shall, following Charles de Secondat, baron de Montesquieu (1689–1755)[6], understand a composition of three powers: the law-making (legislative), the law-enforcing and the law-interpreting parts of public

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

government. Typically national parliament and local (province and city) councils are part of law-making government. Law-enforcing government is called the executive (the administration). And law-interpreting government is called the judiciary [system] (including lawyers etc.).

Documents

A crucial means of expressing public administration is through *documents*.[7] We shall therefore provide a brief domain analysis of a concept of documents. (This document domain description also applies to patient medical records and, by some "light" interpretation, also to artistic works — insofar as they also are documents.)

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

Document Attributes

With documents one can associate, as attributes of documents, the *actors* who created, edited, read, copied, distributed (and to whom distributed),shared, performed calculations and shredded documents.

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

Actor Attributes and Licenses

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

Document Tracing

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

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

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

10.3 The Semantic Intents of Licensor and Licensee Actions

10.3.1 Overview

There are two parties to a license: the *licensor* and the *licensee*. And there is a common agreement concerning a shared "item" between them, namely: the *license* and the *work item:* the artistic work, the patient, the document.

The licensor gives the licensee permission, or mandates the licensee to be obligated to perform certain actions on designated "items".

The licensee performs, or does not perform permitted and/or obligated actions

And the licensee may perform actions not permitted or not obligated.

The license shall serve to ensure that only permitted actions are carried out, and to ensure that all obligated actions are carried out.

Breach of the above, that is, breach of the contracted license may or shall result in revocation of the license.

10.3.2 Licenses and Actions

Licenses

Conceptually a licensor o (for owner) may issue a license named ℓ to licensee u (for user) to perform some actions. The license may syntactically appear as follows:

ℓ : **licensor** o **licenses licensee** u **to**
 perform actions {a1,a2,...,an} **on work item** w

Actions

And, conceptually, licensee (u) may perform actions which can be expressed as follows:

ℓ:a(w); ℓ:a′(w); ...; ℓ:a″(w); ...; ℓ:a‴(w)

These actions $(a, a', ..., a'', ..., a''')$ may be in the set {a1,a2,...,an}, mentioned in the license, or they may not be in that set. In the latter case we have a breach of license ℓ.

Any one licensee may have licensed several licenses $\ell_1, \ell_2, \ldots, \ell_n$. And such a licensee may, in an interleaved fashion, perform actions referring to different licenses:

$$\ell_i : a_i(w); \ell_j : a'_j(w); ...; \ell_k : a''_k(w); ...; \ell_n : a'''_n(w)$$

Two Languages

Thus there is *the language of licenses* and *the language of actions*.

We advise the reader to take note of the distinction between the permitted or obligated actions enumerated in a license and the license name labelled actions actually requested by a licensee.

10.3.3 Sub-licensing, Scheme I

A licensee u may wish to delegate some of the work associated with performing some licensed actions to a colleague (or customer). If, for example the license originally stated:

ℓ : **licensor** o **licenses licensee** u
  to perform actions {a1,a2,...,an} **on work item** w

the licensee (u) may wish a colleague u' to perform a subset of the actions, for example

{ai,aj,...,ak} $\subseteq$ {a1,a2,...,an}

Therefore u would like if the above license

ℓ : **licensor** o **licenses licensee** u
  to perform actions {a1,a2,...,an} **on work item** w

instead was formulated as:

ℓ : **licensor** o **licenses licensee** u
  to perform actions {a1,a2,...,an} **on work item** w
  allowing sub-licensing of actions {ai,aj,...,ak}

where

{ai,aj,...,ak} $\subseteq$ {a1,a2,...,an}

Now licensee u can perform the action

ℓ : **license actions** {a$'$,a$''$,a$'''$} **to** u$'$

The above is an action designator. Its practical meaning is that a license is issued by u:

$\eta(\ell$,u,t): **licensor** u **licenses licensee** u$'$
  to perform actions {a$'$,a$''$,a$'''$} **on work item** w

The above license can be easily "assembled" from the action including the action named license: the context determines who (namely u) is issuing the license, and who or which is the work item. η is a function which applies to license name, agent identifications and time and yields unique new license names.

10.3.4 Sub-licensing, Scheme II

The subset relation

{ai,aj,...,ak} $\subseteq$ {a1,a2,...,an}

mentioned in the sub-licensing part of license

ℓ : **licensor** o **licenses licensee** u
  to perform actions {a1,a2,...,an} **on work item** w
  allowing sub-licensing of actions {ai,aj,...,ak}

may be omitted. In fact one could relax the relation completely and allow any actions {ai,aj,...,ak} whether in {a1,a2,...,an} or not ! That is, the original licensor may mandate that a licensee allow a sub-licensee to perform operations that the licensee is not allowed to perform. Examples are: a licensee may break the shrink-wrap around some licensed software package — an action which may not be performed by the licensor; a medical nurse (i.e., a licensee) may perform actions on patients not allowed performed by the licensor (say, a medical doctor); and a civil servant (say, an archivist) may copy, distribute or shred documents, actions that may not be allowed by the licensor (i.e., the manager of that civil servant), while that civil servant (the archivist) is not allowed to create or read documents.

10.3.5 Multiple Licenses

Consider the following scenario: A licensee u is performing actions a_p, a_q, $\ldots$, a_r, on work item ω, and has licensed u' to perform actions $a_i, a_j, \ldots, a_k$, also on work item ω. The action whereby u licenses u' occurs at some time. At that time u has performed none or some of the actions $a_p, a_q, \ldots, a_r$ (on work item ω), but maybe not all. What is going to happen? Can u and u' go on, in parallel, performing actions on the same work item (ω) ? Our decision is yes, and they can do so in an interleaved manner, not concurrently but alternating, i.e., not accessing the same work item simultaneously.

10.4 Syntax and Informal Semantics

We distinguish between the pragmatics, the semantics and the syntax of languages. Leading textbooks on (formal) semantics of programming languages are [82, 114, 215, 221, 236, 241].

We have already covered the concept of pragmatics and illustrated its application to some issues of license language design.

We shall now illustrate the use of syntax and semantics in license language design.

- *By* **syntax** *we mean (i) the ways in which words are arranged to show meaning (cf. semantics) within and between sentences, and (ii) rules for forming syntactically correct sentences.*
- *By* **semantics** *we mean the study and knowledge [incl. specification] of meaning in language [79].*
- By **informal** *semantics we mean a semantics which is expressed in concise natural language, for example, as here, English.*

10.4.1 General Artistic License Language

We refer to the abstract syntax formalised below (that is, formulas 0.–14.). The work on the specific form of the syntax has been facilitated by the work reported by Xiang JenWen [245].[8]

Licenses and Actions

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

Licenses

Syntax

We first present an abstract syntax of the language of licenses, then we annotate this abstract syntax, and finally we present an informal semantics of that language of licenses.

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

[8]As this work, [245], has yet to be completed the syntax and annotations given here may change.

Syntax Annotations

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

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

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

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

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

copy $\langle$s1,s2,s7$\rangle$ v

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

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

(8.) The license command

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

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

Informal Semantics

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

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

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

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

Actions

Syntax

```
type
9.  V
10. Act = Ln × (Rndr|Copy|Edit|License)
11. Rndr    == mkR(sel:S*,wrk:(W|V))
12. Copy    == mkC(sel:S*,wrk:(W|V),into:V)
13. Edit    == mkE(wrks:V*,into:V)
14. License == mkL(ln:Ln,licensee:Nm,wrk:V,cmds:Cmd-set,fees:PF)
```

Annotations and Informal Semantics:

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

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

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

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

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

14: Sub-Licensing The licensee can further sub-license copied and/or edited work. The licensee must give the license being assembled a unique name. And the licensee must choose to whom to license this work. A sub-license, like does a license, authorises which actions can be performed, and then with which one of a set of alternative selection sequences. No payment function is explicitly mentioned. It is to be semi-automatically derived (from the originally licensed payment fee function and the licensee's payment demands) by means of functionalities provided as part of the licensed payment fee function.

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

10.4.2 Hospital Health Care License Language

We refer to the abstract syntax formalised below (that is, formulas 1.–5.). The work on the specific form of the syntax has been facilitated by the work reported in [8].[9]

Licenses and Actions

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

A Notion of License Execution State

In the context of the Artistic License Language licensees could basically perform licensed actions in any sequence and as often as they so desired. There were, of course, some obvious constraints. Operations on local works could not be done before these had been created — say by copying. Editing could only be done on local works and hence required a prior action of, for example, copying a licensed work. In the context of hospital health care most of the actions can only be performed if the patient has reached a suitable state in the hospitalisation. We refer to Fig. 10.1 on the following page for an idealised hospitalisation plan.

We therefore suggest to join to the licensed commands an indicator which prescribe the (set of) state(s) of the hospitalisation plan in which the command action may be performed.

Two or more medical staff may now be licensed to perform different (or even same !) actions in same or different states. If licensed to perform same action(s) in same state(s) — well that may be "bad license programming" if and only if it is bad medical practice ! One cannot design a language and prevent it being misused!

Licenses

type
0. Ln, Mn, Pn
1. License = Ln × Lic
2. Lic == mkLic(staff1:Mn,mandate:ML,pat:Pn)
3. ML == mkML(staff2:Mn,to_perform_acts:CoL-**set**)
4 CoL = Cmd | ML | Alt

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

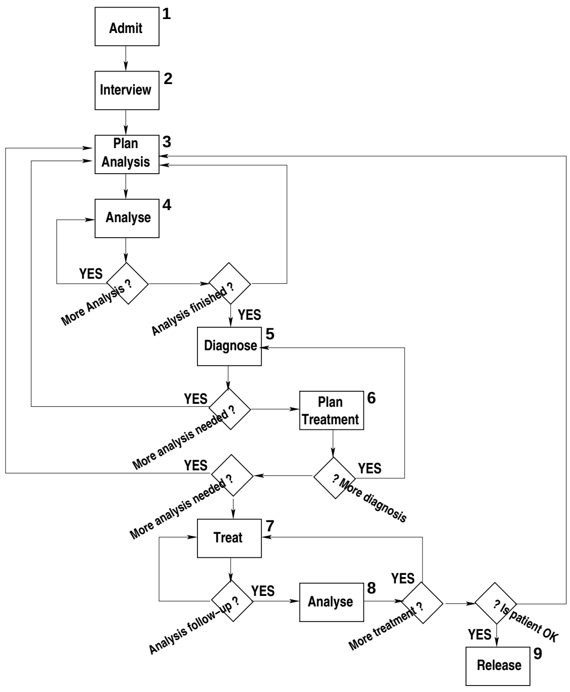


Fig. 10.1. An example hospitalisation plan. States: {1,2,3,4,5,6,7,8,9}

5. Cmd == mkCmd(σs:Σ-**set**,stmt:Stmt)
6 Alt == mkAlt(cmds:Cmd-**set**)
7. Stmt = **admit** | **interview** | **plan-analysis** | **do-analysis**
   | **diagnose** | **plan-treatment** | **treat** | **transfer** | **release**

The above syntax is correct RSL [31–33,44,101,104,106]. But it is decorated! The subtypes {|**boldface keyword**|} are inserted for readability.

Syntax Annotations

(0.) Licenses, medical staff and patients have names.

(1.) Licenses further consist of license bodies (Lic).

(2.) A license body names the licensee (Mn), the patient (Pn), and,

(3.) through the "mandated" licence part (ML), it names the licensor (Mn) and which set of commands (C) or (o) implicit licenses (L, for CoL) the licensor is mandated to issue.

(4.) An explicit command or licensing (CoL) is either a command (Cmd), or a sub-license (ML) or an alternative.

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

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

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

(7.) A statement is either an admit, an interview, a plan analysis, an analysis, a diagnose, a plan treatment, a treatment, a transfer, or a release directive Information given in the patient medical report for the designated state inform medical staff as to the details of analysis, what to base a diagnosis on, of treatment, etc.

Actions

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

Syntax Annotations

(8.) Each action actually attempted by a medical staff refers to the license, and hence the patient name.

(9.) Actions are either of an admit, an interview, a plan analysis, an analysis, a diagnose, a plan treatment, a treatment, a transfer, or a release actions. Each individual action is only allowed in a state σ if the action directive appears in the named license and the patient (medical record) designates state σ.

(10.) Or an action can be a sub-licensing action. Either the sub-licensing action that the licensee is attempting is explicitly mandated by the license (4. ML), or is an alternative one thus implicitly mandated (6.). The full sub-license, as defined in (1.–3.) is compiled from contextual information.

Informal Semantics

An informal, rough-sketch semantics (here abbreviated) would state that a prescribed action is only performed if the patient, cum patient medical record is in an appropriate state; and that the patient is being treated according to the action performed; that records of this treatment and its (partially) analysed outcome is introduced into the patient medical record. The next state of the patient, cum patient medical record, depends on the outcome of the treatment[10]; and hence the patient medical record carries with it, i.e., embodies a, or the, hospitalisation plan in effect for the patient, and a reference to the current state of the patient.

[10]Cf. the diamond-shaped decision boxes in Fig. 10.1 on the preceding page.

10.4.3 Public Administration License Language

We refer to the abstract syntax formalised below (that is, formulas 1.–15.). The work on the specific form of the syntax has been facilitated by the work reported in [26,46,73].[11]

Licenses and Actions

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

Licenses

License Classes

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

Licensed Operations

```
type
7.  Dn, DCn, UDI
8.  Crea  == mkCr(dn:Dn,doc_class:DCn,based_on:UDI-set)
9.  Edit  == mkEd(doc:UDI,based_on:UDI-set)
10.  Read  == mkRd(doc:UDI)
11.  Copy  == mkCp(doc:UDI)
12. Licn  == mkLi(kind:LiTy)
13. LiTy  == grant | extend | restrict | withdraw
14.  Shar  == mkSh(doc:UDI,with:An-set)
15.  Rvok  == mkRv(doc:UDI,from:An-set)
16.  Rlea  == mkRl(dn:Dn)
17.  Rtur  == mkRt(dn:Dn)
18.  Calc  == mkCa(fcts:CFn-set,docs:UDI-set)
19.  Shrd  == mkSh(doc:UDI)
```

[11]As part this work, [73], has yet to be completed the syntax and annotations given here may change.

Syntax & Informal Semantics Annotations

(0.) The are names of licenses (Ln), actors (An), documents (UDI), document classes (DCn) and calculation functions (Cfn).

(1.) There are four kinds of licenses: granting, extending, restricting and withdrawing.

(2.) Actors (licensors) grant licenses to other actors (licensees). An actor is constrained to always grant distinctly named licenses. No two actors grant identically named licenses.[12] A set of operations on (named) documents are granted.

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

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

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

(8.) Creation results in an initially void document which is

not necessarily uniquely named (dn:Dn) (but that name is uniquely associated with the unique document identifier created when the document is created[13]) typed by a document class name (dcn:DCn) and possibly based on one or more identified documents (over which the licensee (at least) has reading rights). We can presently omit consideration of the document class concept. "based on" means that the initially void document contains references to those (zero, one or more) documents.[14] The "based on" documents are moved from licensor to licensee.

(9.) Editing a document may be based on "inspiration" from, that is, with reference to a number of other documents (over which the licensee (at least) has reading rights). What this "be based on" means is simply that the edited document contains those references. (They can therefore be traced.) The "based on" documents are moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

(10.) Reading a document only changes its "having been read" status (etc.) — as per [26]. The read document, if not the result of a copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

(11.) Copying a document increases the document population by exactly one document. All previously existing documents remain unchanged except that the document which served as a master for the copy has been so marked. The copied document is like the master document except that the copied

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

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

[14]They can therefore be traced (etc.) — as per [26].

document is marked to be a copy (etc.) — as per [26]. The master document, if not the result of a create or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions.

(12.) A licensee can sub-license (sL) certain operations to be performed by other actors.

(13.) The granting, extending, restricting or withdrawing permissions, cannot name a license (the user has to do that), do not need to refer to the licensor (the licensee issuing the sub-license), and leaves it open to the licensor to freely choose a licensee. One could, instead, for example, constrain the licensor to choose from a certain class of actors. The licensor (the licensee issuing the sub-license) must choose a unique license name.

(14.) A document can be shared between two or more actors. One of these is the licensee, the others are implicitly given read authorisations. (One could think of extending, instead the licensing actions with a **shared** attribute.) The shared document, if not the result of a create and edit or copy, is moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. Sharing a document does not move nor copy it.

(15.) Sharing documents can be revoked. That is, the reading rights are removed.

(16.) The release operation: if a licensor has authorised a licensee to create a document (and that document, when created got the unique document identifier udi:UDI) then that licensee can release the created, and possibly edited document (by that identification) to the licensor, say, for comments. The licensor thus obtains the master copy.

(17.) The return operation: if a licensor has authorised a licensee to create a document (and that document, when created got the unique document identifier udi:UDI) then that licensee can return the created, and possibly edited document (by that identification) to the licensor — “for good”! The licensee relinquishes all control over that document.

(18.) Two or more documents can be subjected to any one of a set of permitted calculation functions. These documents, if not the result of a creates and edits or copies, are moved from licensor to licensee — if not already so moved as the result of the specification of other authorised actions. Observe that there can be many calculation permissions, over overlapping documents and functions.

(19.) A document can be shredded. It seems pointless to shred a document if that was the only right granted wrt. document.

Actions

20. Action = Ln × Clause
21. Clause = Cre | Edt | Rea | Cop | Lic | Sha | Rvk | Rel | Ret | Cal | Shr
22. Cre == mkCre(dcn:DCn,based_on_docs:UID-**set**)
23. Edt == mkEdt(uid:UID,based_on_docs:UID-**set**)
24. Rea == mkRea(uid:UID)

```
25. Cop == mkCop(uid:UID)
26. Lic == mkLic(license:L)
27. Sha == mkSha(uid:UID,with:An-set)
28. Rvk == mkRvk(uid:UID,from:An-set)
29. Rel == mkRel(dn:Dn,uid:UID)
30. Ret == mkRet(dn:Dn,uid:UID)
31. Cal == mkCal(fct:Cfn,over_docs:UID-set)
32. Shr == mkShr(uid:UID)
```

Preliminary Remarks

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

Syntax & Informal Semantics Annotations

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

(21.) Actions clauses are either create, edit, read, copy, sub-license, share, revoke, release, return, calculate or shred.

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

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

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

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

(26.) A licensee may issue licenses of the kind permitted. The licensee decides whether to do so or not. The licensee decides to whom, over which, if any, documents, and for which operations. The licensee looks after a proper ordering of licensing commands: first grant, then sequences of zero, one or more either extensions or restrictions, and finally, perhaps, a withdrawal.

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

(28.) Sharing may (conditionally) be revoked, partially or fully, that is, wrt. original “sharers”.

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

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

(31.) A license may (conditionally) apply any of a licensed set of calculation functions to “downloaded” (edited, copied, etc.) documents, or can (unconditionally) apply any of a licensed set of calculation functions to created (etc.) documents. The result of a calculation is a document. The licensee obtains all operation authorisations to this document (— as for created documents).

(32.) A license may (conditionally) shred a “downloaded” (etc.) document.

10.4.4 Discussion

Comparisons

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

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

Differences

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

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

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

10.5 Formal Semantics

By formal semantics we understand a definition expressed in a formal language, that is, a language with a mathematical syntax, a mathematical semantics, and a consistent and relative complete proof system. We shall deploy the CSP [137,138,218,222] Specification Language embedded in a Landin–like notation of **let** clauses[15]. We hope someone will complement our definition with a commensurate CafeOBJ [89,90,99,100] definition.

10.5.1 A Model of Common Aspects

Actors: Behaviours and Processes

We see the system as a set of actors. An actor is either a licensor, or a licensee, or, usually, such as we have envisaged our license languages, both. To each actor we associate a behaviour — and we model actor behaviours as CSP processes. So the system is then modelled as a set of concurrent behaviours, that is, parallel ($\|$) CSP processes. Actors are uniquely identified (Aid).

System States

With each actor behaviour we associate a state ($\omega : \Omega$). "Globally" initial such state components are modelled as maps from actor identifiers to states. We shall later analyse these states into major components.

type
  Aid, Ω
  Ωs = Aid $\overrightarrow{m}$ Ω

System Processes

Actor processes communicate with one another over channels. There is a set of actor identifier indexed channels. Potential licensees request licenses. Licensors issue licenses in response to requests. Work items are communicated over these channels. As are payments and reports on use of licensed operations on

[15]— known since the very early 1960's

licensed work items. An actor is either pro-active, requesting licenses, sending payment or reports, or re-active: responding to license requests, sending work items. An actor non-deterministically ($\lceil\rceil$) alternates between these activities.

```
type
    M = Lic | Pay | Rpt | ...
channel
    {a[i]|i:Aid} (Aid×M)
value
    actor: j:Aid × Ω → in,out {a[i]|i:Aid•i≠j} Unit
    actor(j)(ω) ≡ let ω′ = pro_act(j)(ω)⌈⌉re_act(j)(ω) in actor(j)(ω′) end

    system: Ωs → Unit
    system(ωs) ≡ ‖ {actor(i)(ωs(i))|i:Aid}
```

Actor Processes

We have identified two kinds of actor processes: pro-active, during which the actor, by own initiative, (as a prospective licensee) may request a license from a prospective licensor, or, (as an actual licensee) as the result of performing licensed actions sends payments or reports (or other) to the licensor. and re-active, during which the actor, in response to requests (as a licensor) sends a requesting actor a license (whereby the requester becomes a licensee), or "downloads" (access to) requested works or functions. functions.

The Pro-active Actor Behaviour In the pro-active behaviour an actor, at will, i.e., non-deterministically internal choice ($\lceil\rceil$), decides to either request a license (rl) or to perform some action (op). In the former case the actor inquires (l_iq) an own state to find out from which licensor (aid) which kind of license requirements (l_rq) is to be requested. This licensor and these requirements are duly noted in the state. After sending the request the actor continues being an actor in the duly noted state. In the latter case (op) there may be many "next" actions to do (act). The actor inquires (a_iq) an own state to find out which action (designated by op_i) is "next". The actor them performs (act) the designated operation. It is here, for simplification assumed that all operation completions imply a "completion" message (a payment, a report, or other) to the operation licensing actor. So such a message is sent and the operation updated local state is yielded — whereby the pro-active actor "resumes" being an actor as from that state.

```
type
    M = Lic | Pay | Rpt | ...
channel
    {a[i]|i:Aid} (Aid×M)
value
    pro_act: j:Aid → Ω → in,out {a[i]|i:Aid•j≠i}  Ω
```

```
pro_act(j)($\omega$) $\equiv$
    **let** what_to_do = rl $\sqcap$ op **in**
    **case** what_to_do **of**
        rl $\rightarrow$ **let** (k,l_rq,$\omega'$)=iq_l_$\Omega$($\omega$) **in**
                a(aid)!(j,l_rq);$\omega'$ **end**
        op $\rightarrow$ **let** op_i=iq_a_$\Omega$($\omega$) **in**
                **let** (k,m,$\omega'$)=act(op_i)($\omega$) **in**
                a(k)!(j,m) ; $\omega'$ **end** **end**
    **end** **end**
```

The Re-active Actor Behaviour In the re-active behaviour an actor (j), is willing to engage in communication with other actors. This is formalised by a non-deterministic external choice ($\lbrack\!\rbrack$) between either of a set ({...}) of (zero, or more) other actors (k:Aid\{j}) who are trying to contact the re-active actor. The communicated message reveals the identity (k) of the requesting, i.e., the pro-active actor,[16] The message, m, reveals what action (act(m)) the re-active actor is requested to perform. The actor does so/ This results in a reply message m′ and a state change. The reply message is sent to the requesting actor; and the re-active actor yields the requested action-updated state — whereby the re-active actor "resumes" being an actor as from that state.

```
**type**
    M = Lic | Pay | Rpt | ...
**channel**
    {a[i]|i:Aid} (Aid$\times$M)
**value**
    re_act: j:Aid $\rightarrow$ $\Omega$ $\rightarrow$ **in**,**out** {a[i]|i:Aid•j$\neq$i} $\Omega$
    re_act(j)($\omega$)$\equiv$
        **let** (k,m)=$\lbrack\!\rbrack${a(k)?|k:Aid} **in**
        **let** (m′,$\omega'$)=act(m)($\omega$) **in**
        a(k)!(j,m′);$\omega'$ **end** **end**
```

Functions

We first list (and "read") the signatures of the two auxiliary (*iq_l_Ω*, *iq_a_Ω*) and one elaboration (*act*) function assumed in the definition of the pro- and re-active actor processes. After that we discuss the former and suggest definitions of the latter.

[16]Do not get confused by the two k's on either side of the let clause. The left k is yielded by the (input) communication a(k)?. The defining scope of the right side k, as in a(k), is just the right-hand side of the left clause.

Auxiliary Function Signatures

The inquire license function (iq_l_Ω) inspects the actor's state to ("eureka") determine which most desirable licensor (Aid) offers which one kind of desired license requirements (License_Requirements). The inquire action function (iq_a_Ω) inspects the actor's state to (somewhat "eureka") determine which action is "next" in tine to be performed. That action is being designated (Action_Designator).

type
    License_Requirements,Action_Designation
value
    iq_l_Ω: $\Omega \rightarrow$ Aid $\times$ License_Requirements $\times$ Ω
    iq_a_Ω: $\Omega \rightarrow$ Action_Designator

By 'eureka'[17] is meant that the inquiry is internal non-deterministic that is, is not influenced by an outside, could have any one of very many outcomes, and can thus only be rather loosely defined.

Elaboration Function Signature

The action performing function (act) "finds" the designated operation in the current state, applies it in the current state, and yields ("read" backwards) a possibly new state ($\omega : \Omega$), a message (M) to be sent to the licensor (Aid) who authorised the operation and may need or which to have a payment, a report, or some such thing "back"!

type
    Action_Designation
value
    act: Action_Designation $\rightarrow \Omega \rightarrow$ Aid $\times$ M $\times$ Ω

Discussion of Auxiliary Functions

The auxiliary functions are usually not computable functions. The actors are not robots. And it is not necessary to further define these functions beyond stating their signatures as they are usually performed by human actors. The signature of the inquire license function expresses a possible change to the inquired state. One would think of the inquiring actor somehow noting down, remembering, as it were, which inquiries were attempted or had been made. The signature of the inquire actions function does not express such a state change. But it could be expressed as well.

[17] "Eureka" used to express triumph on a discovery, heuristics

Schema Definitions of Elaboration Functions

10.6 A Transport Contract Language

10.6.1 Narrative

Preparations

In a number of steps (‘A Synopsis’, ‘A Pragmatics and Semantics Analysis’, and ‘Contracted Operations, An Overview’) we arrive at a sound basis from which to formulate the narrative. We shall, however, forego such a detailed narrative. Instead we leave that detailed narrative to the reader. (The detailed narrative can be “derived” from the formalisation.)

A Synopsis

Contracts obligate transport companies to deliver bus traffic according to a timetable. The timetable is part of the contract. A contractor may subcontract (other) transport companies to deliver bus traffic according to timetables that are sub-parts of their own timetable. Contractors are either public transport authorities or contracted transport companies. Contracted transport companies may cancel a subset of bus rides provided the total amount of cancellations per 24 hours for each bus line does not exceed a contracted upper limit[18]. The cancellation rights are spelled out in the contract[19]. A subcontractor cannot increase a contracted upper limit for cancellations above what the sub-contractor was told (in its contract) by its contractor[20]. Etcetera.

A Pragmatics and Semantics Analysis

The “works” of the bus transport contracts are two: the timetables and, implicitly, the designated (and obligated) bus traffic. A bus timetable appears to define one or more bus lines, with each bus line giving rise to one or more bus rides. We assume a timetable description along the lines of Appendix G. Nothing is (otherwise) said about regularity of bus rides. It appears that bus ride cancellations must be reported back to the contractor. And we assume that cancellations by a sub-contractor is further reported back also to the subcontractor’s contractor. Hence eventually that the public transport authority is notified.

Nothing is said, in the contracts, such as we shall model them, about passenger fees for bus rides nor of percentages of profits (i.e., royalties) to be paid back from a sub-contractor to the contractor. So we shall not bother, in this example, about transport costs nor transport subsidies. But will leave that necessary aspect as an exercise.

[18]We do not treat this aspect further in this chapter.
[19]See Footnote 18.
[20]See Footnote 18.

The opposite of cancellations appears to be 'insertion' of extra bus rides, that is, bus rides not listed in the time table, but, perhaps, mandated by special events[21] We assume that such insertions must also be reported back to the contractor.

We assume concepts of acceptable and unacceptable bus ride delays. Details of delay acceptability may be given in contracts, but we ignore further descriptions of delay acceptability. but assume that unacceptable bus ride delays are also to be (iteratively) reported back to contractors.

We finally assume that sub-contractors cannot (otherwise) change timetables. (A timetable change can only occur after, or at, the expiration of a license.) Thus we find that contracts have definite period of validity. (Expired contracts may be replaced by new contracts, possibly with new timetables.)

Contracted Operations, An Overview

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

10.6.2 A Formalisation

Syntax

We treat separately, the syntax of contracts (for a schematised example see Page 310) and the syntax of the actions implied by contracts.

Contracts

A concrete example contract can be 'schematised':

cid: **contractor** cor **contracts sub-contractor** cee
  to perform operations
    {"start","cancel","insert","subcontract"}
  with respect to timetable tt.

We assume a context (a global state) in which all contract actions (including contracting) takes place and in which the implicit transport net (see Appendix B) is defined.

63. contracts, contractors and sub-contractors have unique identifiers CId, CNm, CNm.

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

64. A contract has a unique identification, names the contractor and the sub-contractor (and we assume the contractor and sub-contractor names to be distinct). A contract also specifies a contract body.
65. A contract body stipulates a timetable and the set of operations that are mandated or allowed by the contractor.
66. An Operation is either a `"start"` (i.e., start a bus ride), a bus ride `"cancel"`lation, a bus ride `"insert"`, or a `"subcontrat"`ing operation.

type
63. CId, CNm
64. Contract = CId × CNm × CNm × Body
65. Body = Op-**set** × TT
66. Op == ″`start`″ | ″`cancel`″ | ″`insert`″ | ″`subcontract`″

An abstract example contract:

(cid,cnm_i,cnm_j,({″`start`″,″`cancel`″,″`insert`″,″`sublicense`″},tt))

Actions

Example actions can be schematised:

(a) cid: **conduct bus ride** (blid,bid) **to start at time** t
(b) cid: **cancel bus ride** (blid,bid) **at time** t
(c) cid: **insert bus ride like** (blid,bid) **at time** t

The schematised license (Page 310) shown earlier is almost like an action; here is the action form:

(d) cid: **sub-contractor** cnm′ **is granted a contract** cid′
to perform operations {”conduct”,”cancel”,”insert”,sublicense”}
with respect to timetable tt′.

All actions are being performed by a sub-contractor in a context which defines that sub-contractor cnm, the relevant net, say n, the base contract, referred here to by cid (from which this is a sublicense), and a timetable tt of which tt′ is a subset. contract name cnm′ is new and is to be unique. The subcontracting action can (thus) be simply transformed into a contract as shown on Page 310.

type
Action = CNm × CId × (SubCon | SmpAct) × Time
SmpAct = Start | Cancel | Insert
Conduct == mkSta(s_blid:BLId,s_bid:BId)
Cancel == mkCan(s_blid:BLId,s_bid:BId)
Insert = mkIns(s_blid:BLId,s_bid:BId)
SubCon == mkCon(s_cid:CId,s_cnm:CNm,s_body:(s_ops:Op-**set**,s_tt:TT))

examples:

(a) (cnm,cid,mkSta(blid,id),t)
(b) (cnm,cid,mkCan(blid,id),t)
(c) (cnm,cid,mkIns(blid,id),t)
(d) (cnm,cid,
        mkCon(cid′,
            ({″conduct″,″cancel″,″insert″,″sublicense″},tt′),t))

where: cid′ = generate_CId(cid,cnm,t) See Item/Line 69

We observe that the essential information given in the start, cancel and insert action prescriptions is the same; and that the RSL record-constructors (mkSta, mkCan, mkIns) make them distinct.

Contract Identification

67. There is a "root" contract name, rcid.
68. There is a "root" contractor name, rcnm.

value
67 rcid:CId
68 rcnm:CNm

All other contract names are derived from the root name. Any contractor can at most generate one contract name per time unit. Any, but the root, sub-contractor obtains contracts from other sub-contractors, i.e., the contractor. Eventually all sub-contractors, hence contract identifications can be referred back to the root contractor.

69. Such a contract name generator is a function which given a contract identifier, a sub-contractor name and the time at which the new contract identifier is generated, yields the unique new contract identifier.
70. From any but the root contract identifier one can observe the contract identifier, the sub-contractor name and the time that "went into" its creation.

value
69 gen_CId: CId $\times$ CNm $\times$ Time $\to$ CId
70 obs_CId: CId $\stackrel{\sim}{\to}$ CIdL [**pre** obs_CId(cid):cid$\neq$rcid]
70 obs_CNm: CId $\stackrel{\sim}{\to}$ CNm [**pre** obs_CNm(cid):cid$\neq$rcid]
70 obs_Time: CId $\stackrel{\sim}{\to}$ Time [**pre** obs_Time(cid):cid$\neq$rcid]

71. All contract names are unique.

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

72. Thus a contract name defines a trace of license name, sub-contractor name and time triple, “all the way back” to “creation”.

```
type
   CIdCNmTTrace = TraceTriple*
   TraceTriple == mkTrTr(CId,CNm,s_t:Time)
value
72 contract_trace: CId → LCIdCNmTTrace
72 contract_trace(cid) ≡
72   case cid of
72     rcid → ⟨⟩,
72     _ → contract_trace(obs_LicNm(cid))^⟨obs_TraceTriple(cid)⟩
72   end

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

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

Well, there is a theorem to be proven once we have outlined the full formal model of this contract language: namely that time entries in contract name traces increase with increasing indices.

```
theorem
   ∀ licn:LicNm •
      ∀ trace:LicNmLeeNmTimeTrace • trace ∈ license_trace(licn) ⇒
            ∀ i:Nat • {i,i+1}⊆inds trace ⇒ s_t(trace(i))<s_t(trace(i+1))
```

Semantics

Execution State

Local and Global States Each sub-contractor has an own local state and has access to a global state. All sub-contractors access the same global state. The global state is the bus traffic on the net. There is, in addition, a notion of running-state. It is a meta-state notion. The running state “is made up” from the fact that there are n sub-contractors, each communicating, as contractors, over channels with other sub-contractors. The global state is distinct from sub-contractor to sub-contractor – no sharing of local states between sub-contractors. We now examine, in some detail, what the states consist of.

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

type
133. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI) 420

183. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos))) 425
184. BPos = atHub | onLnk | atBS
185. atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
186. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
187. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)

We shall consider BusTraffic (with its Net) to reflect the global state.

Local sub-contractor contract States: Semantic Types A sub-contractor state contains, as a state component, the zero, one or more contracts that the sub-contractor has received and that the sub-contractor has sublicensed.

type
  Body = Op-**set** × TT
  LicΣ = RcvLicΣ×SubLicΣ×LorBusΣ
  RcvLicΣ = LorNm $\overrightarrow{m}$ (LicNm $\overrightarrow{m}$ (Body×TT))
  SubLicΣ = LeeNm $\overrightarrow{m}$ (LicNm $\overrightarrow{m}$ Body)
  LorBusΣ ... [see below and Page 315] ...

(Recall that LorNm and LeeNm are the same.)

In RecvLics we have that LorNm is the name of the contractor by whom the contract has been granted, LicNm is the name of the contract assigned by the contractor to that license, Body is the body of that license, and TT is that part of the timetable of the Body which has not (yet) been sublicensed.

In DespLics we have that LeeNm is the name of the sub-contractor to whom the contract has been despatched, the first (left-to-right) LicNm is the name of the contract on which that sublicense is based , the second (left-to-right) LicNm is the name of the sublicense, and License is the contract named by the second LicNm.

Local sub-contractor Bus States: Semantic Types The sub-contractor state further contains a bus status state component which records which buses are free, FreeBusΣ, that is, available for dispatch, and where "garaged", which are in active use, ActvBusΣ, and on which bus ride, and a bus history for that bus ride, and histories of all past bus rides, BusHistΣ. A trace of a bus ride is a list of zero, one or more pairs of times and bus stops. A bus history, BusHistory, associates a bus trace to a quadruple of bus line identifiers, bus ride identifiers, contract names and sub-contractor name.[22]

type

BusNo
$\mathrm{Bus}\Sigma = \mathrm{FreeBuses}\Sigma \times \mathrm{ActvBuses}\Sigma \times \mathrm{BusHists}\Sigma$
$\mathrm{FreeBuses}\Sigma = \mathrm{BusStop} \underset{m}{\rightarrow} \mathrm{BusNo}\text{-}\mathbf{set}$
$\mathrm{ActvBuses}\Sigma = \mathrm{BusNo} \underset{m}{\rightarrow} \mathrm{BusInfo}$
$\mathrm{BusInfo} = \mathrm{BLId}\times\mathrm{BId}\times\mathrm{LicNm}\times\mathrm{LeeNm}\times\mathrm{BusTrace}$
$\mathrm{BusHists}\Sigma = \mathrm{Bno} \underset{m}{\rightarrow} \mathrm{BusInfo}^*$
$\mathrm{BusTrace} = (\mathrm{Time}\times\mathrm{BusStop})^*$
$\mathrm{LorBus}\Sigma = \mathrm{LeeNm} \underset{m}{\rightarrow} (\mathrm{LicNm} \underset{m}{\rightarrow} ((\mathrm{BLId}\times\mathrm{BId}) \underset{m}{\rightarrow} (\mathrm{BNo}\times\mathrm{BusTrace})))$

A bus is identified by its unique number (i.e., registration) plate (BusNo). We could model a bus by further attributes: its capacity, etc., for for the sake of modelling contracts this is enough. The two components are modified whenever a bus is commissioned into action or returned from duty, that is, twice per bus ride.

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

73. For simplicity we assume a constant net — constant, that is, only with respect to the set of identifiers links and hubs. These links and hubs obviously change state over time.
74. We also assume a constant set, leens, of sub-contractors. In reality sub-contractors, that is, transport companies, come and go, are established and go out of business. But assuming constancy does not materially invalidate our model. Its emphasis is on contracts and their implied actions — and these are unchanged wrt. constancy or variability of contract holders.
75. There is an initial bus traffic, tr.
76. There is an initial time, t_0, which is equal to or larger than the start of the bus traffic tr.
77. To maintain the bus traffic "spelled out", in total, by timetable tt one needs a number of buses.
78. The various bus companies (that is, sub-contractors) each have a number of buses. Each bus, independent of ownership, has a unique (car number plate) bus number (BusNo).
    These buses have distinct bus (number [registration] plate) numbers.
79. We leave it to the reader to define a function which ascertain the minimum number of buses needed to implement traffic tr.

[22] In this way one can, from the bus history component ascertain for any bus which for whom (sub-contractor), with respect to which license, it carried out a further bus line and bus ride identified tour and its trace.

value
73. net : N,
74. leens : LeeNm-**set**,
75. tr : BusTraffic, **axiom** wf_Traffic(tr)(net)
76. t_0 : T • $t_0 \geq$ **min dom** tr,
77. min_no_of_buses : **Nat** • necessary_no_of_buses(itt),
78. busnos : BusNo-**set** • **card** busnos $\geq$ min_no_of_buses

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

80. To "bootstrap" the whole contract system we need a distinguished contractor, named init_leen, whose only license originates with a "ghost" contractor, named root_leen (o, for outside [the system]).
81. The initial, i.e., the distinguished, contract has a name, root_licn.
82. The initial contract can only perform the `"sublicense"` operation.
83. The initial contract has a timetable, tt.
84. The initial contract can thus be made up from the above.

value
80. root_leen,init_ln : LeeNm • root_leen $\notin$ leens $\wedge$ initi_leen $\in$ leens,
81. root_licn : LicNm
82. iops : Op-**set** = {″`sublicense`″},
83. itt : TT,
84. init_lic:License = (root_licn,root_leen,(iops,itt),init_leen)

Initial sub-contractor contract States

type
  InitLicΣs = LeeNm $\overrightarrow{m}$ LicΣ
value
  ilσ:LicΣ=([init_leen $\mapsto$ [root_leen $\mapsto$ [iln $\mapsto$ init_lic]]]
    $\cup$ [leen $\mapsto$ [ ] | leen:LeeNm • leen $\in$ leenms$\backslash${init_leen}],[ ],[ ])

Initial sub-contractor Bus States

85. Initially each sub-contractor possesses a number of buses.
86. No two sub-contractors share buses.
87. We assume an initial assignment of buses to bus stops of the free buses state component and for respective contracts.
88. We do not prescribe a "satisfiable and practical" such initial assignment (ibσs).
89. But we can constrain ibσs.
90. The sub-contractor names of initial assignments must match those of initial bus assignments, allbuses.
91. Active bus states must be empty.

92. No two free bus states must share buses.
93. All bus histories are void.

type
85. AllBuses′ = LeeNm $\overrightarrow{m}$ BusNo-**set**
86. AllBuses = {|ab:AllBuses′•∀ {bs,bs′}⊆**rng** ab∧bns≠bns′⇒bns ∩ bns′={}|}
87. InitBus*Σ*s = LeeNm $\overrightarrow{m}$ Bus*Σ*
value
86. allbuses:Allbuses • **dom** allbuses = leenms ∪ {root_leen} ∧ ∪ **rng** allbuses = busnos

```
87. ibσs:InitBusΣs
88. wf_InitBusΣs: InitBusΣs → Bool
89. wf_InitBusΣs(iσs) ≡
90.   dom iσs = leenms ∧
91.   ∀ (_,abσ,_):BusΣ•(_,abσ,_) ∈ rng iσs ⇒ abσ=[ ] ∧
92.   ∀ (fbiσ,abiσ),(fbjσ,abjσ):BusΣ •
92.     {(fbiσ,abiσ),(fbjσ,abjσ)}⊆rng iσs
92.       ⇒ (fbiσ,actiσ)≠(fbjσ,actjσ)
92.         ⇒ rng fbiσ ∩ rng fbjσ = {}
93.           ∧ actiσ=[ ]=actjσ
```

Communication Channels The running state is a meta notion. It reflects the channels over which contracts are issued; messages about committed, cancelled and inserted bus rides are communicated, and fund transfers take place.

Sub-Contractor↔Sub-Contractor Channels Consider each sub-contractor (same as contractor) to be modelled as a behaviour. Each sub-contractor (licensor) behaviour has a unique name, the LeeNm. Each sub-contractor can potentially communicate with every other sub-contractor. We model each such communication potential by a channel. For n sub-contractors there are thus $n \times (n-1)$ channels.

channel { l_to_l[fi,ti] | fi:LeeNm,ti:LeeNm • {fi,ti}⊆leens ∧ fi≠ti } LLMSG
type LLMSG = ...

We explain the declaration: **channel** { l_to_l[fi,ti] | fi:LeeNm, ti:LeeNm • fi≠ti } LLMSG. It prescribes $n \times (n-1)$ channels (where n is the cardinality of the sub-contractor name sets). Each channel is prescribed to be capable of communicating messages of type MSG. The square brackets [...] defines l_to_l (sub-contractor-to-sub-contractor) as an array.

We shall later detail the BusRideNote, CancelNote, InsertNote and FundXfer message types.

Sub-Contractor↔Bus Channels Each sub-contractor has a set of buses. That set may vary. So we allow for any sub-contractor to potentially communicate with any bus. In reality only the buses allocated and scheduled by a sub-contractor can be "reached" by that sub-contractor.

```
channel { l_to_b[l,b] | l:LeeNm,b:BNo • l ∈ leens ∧ b ∈ busnos } LBMSG
type LBMSG = ...
```

Sub-Contractor↔Time Channels Whenever a sub-contractor wishes to perform a contract operation that sub-contractor needs know the time. There is just one, the global time, modelled as one behaviour: time_clock.

```
channel { l_to_t[l] | l:LeeNm • l ∈ leens } LTMSG
type LTMSG = ...
```

Bus↔Traffic Channels Each bus is able, at any (known) time to ascertain where in the traffic it is. We model bus behaviours as processes, one for each bus. And we model global bus traffic as a single, separate behaviour.

```
channel { b_to_tr[b] | b:BusNo • b ∈ busnos } LTrMSG
type
   BTrMSG == reqBusAndPos(s_bno:BNo,s_t:Time) | (Bus×BusPos)
```

Buses↔Time Channel Each bus needs to know what time it is.

```
channel { b_to_t[b] | b:BNo • b ∈ busnos } BTMSG
type
   BTMSG ...
```

Local sub-contractor Bus States: Update Functions

```
value
   update_BusΣ: Bno×(T×BusStop) → ActBusΣ → ActBusΣ
   update_BusΣ(bno,(t,bs))(actσ) ≡
      let (blid,bid,licn,leen,trace) = actσ(bno) in
      actσ†[bno↦(licn,leen,blid,bid,trace⌢⟨(t,bs)⟩)] end
      pre bno ∈ dom actσ
```

```
value
   update_FreeΣ_ActΣ:
      BNo×BusStop→BusΣ→BusΣ
   update_FreeΣ_ActΣ(bno,bs)(freeσ,actvσ) ≡
      let (_,_,_,_,trace) = actσ(b) in
      let freeσ′ = freeσ†[bs ↦ (freeσ(bs))∪{b}] in
      (freeσ′,actσ\{b}) end end
      pre bno ∉ freeσ(bs) ∧ bno ∈ dom actσ
```

```
value
  update_LorBusΣ:
    LorNm×lin:LicNm×len:LeeNm×(BLId×BId)×(BNo×Trace)→LorBusΣ
      → out {l_to_l[len,lorn]|lon:LorNm•lon ∈ leens\{len}} LorΣ
  update_LorBusΣ(lon,lin,len,(bli,bi),(bno,tr))(lbσ) ≡
    l_to_l[len,lon]!Licensor_BusHistΣMsg(bno,bli,bi,lin,len,tr) ;
    lbσ†[len↦(lbσ(len))†[lin↦((lbσ(len))(lin))†[(bli,bi)↦(bno,trace)]]]
    pre len ∈ dom lbσ ∧ lin ∈ dom (lbσ(len))
```

```
value
  update_ActΣ_FreeΣ:
    LeeNm×LicNm×BusStop×(BLId×BId)→BusΣ→BusΣ×BNo
  update_ActΣ_FreeΣ(leen,licn,bs,(blid,bid))(freeσ,actvσ) ≡
    let bno:Bno • bno ∈ freeσ(bs) in
    ((freeσ\{bno},actvσ ∪ [bno↦(blid,bid,licnm,leenm,⟨⟩)]),bno) end
    pre bs ∈ dom freeσ ∧ bno ∈ freeσ(bs) ∧ bno ∉ dom actvσ ∧ ...
```

Run-time Environment So we shall be modelling the transport contract domain as follows: As for behaviours we have this to say. There will be n sub-contractors. One sub-contractor will be initialised to one given license. You may think of this sub-contractor being the transport authority. Each sub-contractor is modelled, in RSL, as a CSP-like process. With each sub-contractor, l_i, there will be a number, b_i, of buses. That number may vary from sub-contractor to sub-contractor. There will be b_i channels of communication between a sub-contractor and that sub-contractor's buses, for each sub-contractor. There is one global process, the traffic. There is one channel of communication between a sub-contractor and the traffic. Thus there are n such channels.

As for operations, including behaviour interactions we assume the following. All operations of all processes are to be thought of as instantaneous, that is, taking nil time ! Most such operations are the result of channel communications either just one-way notifications, or inquiry requests. Both the former (the one-way notifications) and the latter (inquiry requests) must not be indefinitely barred from receipt, otherwise holding up the notifier. The latter (inquiry requests) should lead to rather immediate responses, thus must not lead to dead-locks.

The System Behaviour

The system behaviour starts by establishing a number of licenseholder and bus_ride behaviours and the single time_clock and bus_traffic behaviours

```
value
  system: Unit → Unit
  system() ≡
```

```
licenseholder(init_leen)(il$\sigma$(init_leen),ib$\sigma$(init_leen))
|| (|| {licenseholder(leen)(il$\sigma$(leen),ib$\sigma$(leen))
      | leen:LeeNm•leen ∈ leens\{init_leen}})
|| (|| {bus_ride(b,leen)(root_lorn,"nil")
      | leen:LeeNm,b:BusNo •leen ∈ dom allbuses ∧ b ∈ allbuses(leen)})
|| time_clock($t_0$) || bus_traffic(tr)
```

The initial licenseholder behaviour states are individually initialised with basically empty license states and by means of the global state entity bus states. The initial bus behaviours need no initial state other than their bus registration number, a "nil" route prescription, and their allocation to contract holders as noted in their bus states.

Only a designated licenseholder behaviour is initialised to a single, received license.

Semantic Elaboration Functions

The Licenseholder Behaviour

94. The licenseholder behaviour is a sequential, but internally non-deterministic behaviour.
95. It internally non-deterministically (⌈⌉) alternates between
    (a) performing the licensed operations (on the net and with buses),
    (b) receiving information about the whereabouts of these buses, and informing contractors of its (and its subsub-contractors') handling of the contracts (i.e., the bus traffic), and
    (c) negotiating new, or renewing old contracts.

```
94. licenseholder: LeeNm → (Lic$\Sigma$×Bus$\Sigma$) → Unit
95. licenseholder(leen)(lic$\sigma$,bus$\sigma$) ≡
95.   licenseholder(leen)((lic_ops⌈⌉bus_mon⌈⌉neg_licenses)(leen)(lic$\sigma$,bus$\sigma$))
```

The Bus Behaviour

96. Buses ply the network following a timed bus route description.
    A timed bus route description is a list of timed bus stop visits.
97. A timed bus stop visit is a pair: a time and a bus stop.
98. Given a bus route and a bus schedule one can construct a timed bus route description.
    (a) The first result element is the first bus stop and origin departure time.
    (b) Intermediate result elements are pairs of respective intermediate schedule elements and intermediate bus route elements.
    (c) The last result element is the last bus stop and final destination arrival time.
99. Bus behaviours start with a "nil" bus route description.

```
type
96. TBR = TBSV*
97. TBSV = Time × BusStop
value
98. conTBR: BusRoute × BusSched → TBR
98. conTBR((dt,til,at),(bs1,bsl,bsn)) ≡
98(a))    ⟨(dt,bs1)⟩
98(b))    ⌢ ⟨(til[i],bsl[i])|i:Nat•i:⟨1..len til⟩⟩
98(c))    ⌢ ⟨(at,bsn)⟩
          pre: len til = len bsl
type
99. BRD == "nil" | TBR
```

100. The bus behaviour is here abstracted to only communicate with some contract holder, time and traffic,
101. The bus repeatedly observes the time, t, and its position, po, in the traffic.
102. There are now four case distinctions to be made.
103. If the bus is idle (and a a bus stop) then it waits for a next route, brd′ on which to engage.
104. If the bus is at the destination of its journey then it so informs its owner (i.e., the sub-contractor) and resumes being idle.
105. If the bus is ‘en route’, at a bus stop, then it so informs its owner and continues the journey.
106. In all other cases the bus continues its journey

```
value
100. bus_ride: leen:LeeNm × bno:Bno → (LicNm × BRD) →
100.      in,out l_to_b[leen,bno], in,out b_to_tr[bno], in b_to_t[bno] Unit
100. bus_ride(leen,bno)(licn,brd) ≡
101.    let t = b_to_t[bno]? in
101.    let (bus,pos) = (b_to_tr[bno]!reqBusAndPos(bno,t) ; b_to_tr[bno]?) in
102.    case (brd,pos) of
103.      ("nil",mkAtBS(_,_,_,_)) →
103.           let (licn,brd′) = (l_to_b[leen,bno]!reqBusRid(pos);l_to_b[leen,bno]?) in
103.           bus_ride(leen,bno)(licn,brd′) end
104.      (⟨(at,pos)⟩,mkAtBS(_,_,_,_)) →
104s           l_to_b[l,b]!BusΣMsg(t,pos);
104           l_to_b[l,b]!BusHistΣMsg(licn,bno);
104           l_to_b[l,b]!FreeΣ_ActΣMsg(licn,bno) ;
104            bus_ride(leen,bno)(ilicn,"nil"),
105.      (⟨(t,pos),(t′,bs′)⟩⌢brd′,mkAtBS(_,_,_,_)) →
105s           l_to_b[l,b]!BusΣMsg(t,pos) ;
105            bus_ride(licn,bno)(⟨(t′,bs′)⟩⌢brd′),
106.      _ → bus_ride(leen,bno)(licn,brd) end end end
```

In formula line 101 of bus_ride we obtained the bus. But we did not use “that” bus ! We we may wish to record, somehow, number of passengers alighting and boarding at bus stops, bus fees paid, one way or another, etc. The bus, which is a time-dependent entity, gives us that information. Thus we can revise formula lines 104s and 105s:

```
Simple:   104s  l_to_b[l,b]!Bus$\Sigma$Msg(pos);
Revised: 104r  l_to_b[l,b]!Bus$\Sigma$Msg(pos,bus_info(bus));

Simple:   105s  l_to_b[l,b]!Bus$\Sigma$Msg(pos);
Revised: 105r  l_to_b[l,b]!Bus$\Sigma$Msg(pos,bus_info(bus));
```

```
type
   Bus_Info = Passengers × Passengers × Cash × ...
value
   bus_info: Bus → Bus_Info
   bus_info(bus) ≡ (obs_alighted(bus),obs_boarded(bus),obs_till(bus),...)
```

It is time to discuss our description (here we choose the bus_ride behaviour) in the light of our claim of modeling “the domain”. These are our comments:

- First one should recognise, i.e., be reminded, that the narrative and formal descriptions are always abstractions. That is, they leave out few or many things. We, you and I, shall never be able to describe everything there is to describe about even the simplest entity, operation, event or behaviour.

The Global Time Behaviour

107. The time_clock is a never ending behaviour — started at some time t_0.
108. The time can be inquired at any moment by any of the licenseholder behaviours and by any of the bus behaviours.
109. At any moment the time_clock behaviour may not be inquired.
110. After a skip of the clock or an inquiry the time_clock behaviour continues, non-deterministically either maintaining the time or advancing the clock!

```
value
107. time_clock: T →
107.       in,out {l_to_t[leen] | leen:LeeNm • leen ∈ leenms}
107.       in,out {b_to_t[bno] | bno:BusNo • bno ∈ busnos}  Unit
107.  time_clock:(t) ≡
109.    (skip ⌈⌉
108.    (⌈⌉{l_to_t[leen]? ; l_to_t[leen]!t | leen:LeeNm•leen ∈ leens})
108.    ⌈⌉ (⌈⌉{b_to_t[bno]? ; b_to_t[bno]!t | bno:BusNo•bno ∈ busnos})) ;
110.    (time_clock:(t) ⌈⌉ time_clock(t+$\delta_t$))
```

The Bus Traffic Behaviour

111. There is a single bus_traffic behaviour. It is, "mysteriously", given a constant argument, "the" traffic, tr.
112. At any moment it is ready to inform of the position, bps(b), of a bus, b, assumed to be in the traffic at time t.
113. The request for a bus position comes from some bus.
114. The bus positions are part of the traffic at time t.
115. The bus_traffic behaviour, after informing of a bus position reverts to "itself".

```
value
111. bus_traffic: TR → in,out {b_to_tr[bno]|bno:BusNo•bno ∈ busnos} Unit
111. bus_traffic(tr) ≡
113.   ⌷ { let reqBusAndPos(bno,time) = b_to_tr[b]? in assert b=bno
112.       if time ∉ dom tr then chaos else
114.       let (_,bps) = tr(t) in
112.       if bno ∉ dom tr(t) then chaos else
112.       b_to_tr[bno]!bps(bno) end end end end | b:BusNo•b ∈ busnos} ;
115.   bus_traffic(tr)
```

License Operations

116. The lic_ops function models the contract holder choosing between and performing licensed operations.
We remind the reader of the four actions that licensed operations may give rise to; cf. the abstract syntax of actions, Page 311.
117. To perform any licensed operation the sub-contractor needs to know the time and
118. must choose amongst the four kinds of operations that are licensed. The choice function, which we do not define, makes a basically non-deterministic choice among licensed alternatives. The choice yields the contract number of a received contract and, based on its set of licensed operations, it yields either a simple action or a sub-contracting action.
119. Thus there is a case distinction amongst four alternatives.
120. This case distinction is expressed in the four lines identified by: 120.
121. All the auxiliary functions, besides the action arguments, require the same state arguments.

```
value
116. lic_ops: LeeNm → (LicΣ×BusΣ) → (LicΣ×BusΣ)
116. lic_ops(leen)(licσ,busσ) ≡
117.   let t = (time_channel(leen)!req_Time;time_channel(leen)?) in
118.   let (licn,act) = choice(licσ)(busσ)(t) in
119.   (case act of
120.      mkCon(blid,bid)  → cndct(licn,leenm,t,act),
120.     mkCan(blid,bid)   → cancl(licn,leenm,t,act),
120.     mkIns(blid,bid)   → insrt(licn,leenm,t,act),
120.     mkLic(leenm′,bo) → sublic(licn,leenm,t,act) end)(licσ,busσ) end end

        cndct,cancl,insert: SmpAct→(LicΣ×BusΣ)→(LicΣ×BusΣ)
        sublic: SubLic→(LicΣ×BusΣ)→(LicΣ×BusΣ)
```

Bus Monitoring Like for the bus_ride behaviour we decompose the bus_monitoring behaviour into two behaviours. The local_bus_monitoring behaviour monitors the buses that are commissioned by the sub-contractor. The licensor_bus_monitoring behaviour monitors the buses that are commissioned by sub-contractors sub-contractd by the contractor.

```
value
   bus_mon: l:LeeNm → (LicΣ×BusΣ)
               → in {l_to_b[l,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
   bus_mon(l)(licσ,busσ) ≡
      local_bus_mon(l)(licσ,busσ) ⌈⌉ licensor_bus_mon(l)(licσ,busσ)
```

122. The local_bus_monitoring function models all the interaction between a contract holder and its despatched buses.
123. We show only the communications from buses to contract holders.
124. Etcetera.

```
122. local_bus_mon: leen:LeeNm → (LicΣ×BusΣ)
123.       → in {l_to_b[leen,b]|b:BNo•b ∈ allbuses(l)} (LicΣ×BusΣ)
122. local_bus_mon(leen)(licσ:(rlσ,slσ,lbσ),busσ:(fbσ,abσ)) ≡
124.   let (bno,msg) = ⌈⌉{(b,l_to_b[l,b]?)|b:BNo•b ∈ allbuses(leen)} in
124.   let (blid,bid,licn,lorn,trace) = abσ(bno) in
124.   case msg of
124.     BusΣMsg(t,bs) →
124.       let abσ′ = update_BusΣ(bno)(licn,leen,blid,bid)(t,bs)(abσ) in
124.       (licσ,(fbσ,abσ′,histσ)) end,
124.     BusHistΣMsg(licn,bno) →
124.       let lbσ′ = update_LorBusΣ
124.          (obs_LorNm(licn),licn,leen,(blid,bid),(b,trace))(lbσ)  in
124.       l_to_l[leen,obs_LorNm(licn)] !
124.          Licensor_BusHistΣMsg(licn,leen,bno,blid,bid,tr);
124.       ((rlσ,slσ,lbσ′),busσ) end
124.     FreeΣ_ActΣMsg(licn,bno) →
124.       let (fbσ′,abσ′) = update_FreeΣ_ActΣ(bno,bs)(fbσ,abσ) in
124.       (licσ,(fbσ′,abσ′)) end
124.   end end end
```

125. Reader is to provide the narrative!

```
125. licensor_bus_mon: lorn:LorNm → (Lic$\Sigma$×Bus$\Sigma$)
125.   → **in** {l_to_l[lorn,leen]|leen:LeeNm•leen ∈ leenms\{lorn}}
125.      (Lic$\Sigma$×Bus$\Sigma$)
125. licensor_bus_mon(lorn)(lic$\sigma$,bus$\sigma$) ≡
125.     **let** (rl$\sigma$,sl$\sigma$,lbh$\sigma$) = lic$\sigma$ **in**
125.     **let** (leen,Licensor_BusHist$\Sigma$Msg(licn,leen″,bno,blid,bid,tr))
125.       = ⌷{(leen′,l_to_l[lorn,leen′]?)|leen′:LeeNm•leen′ ∈ leenms\{lorn}} **in**
125.     **let** lbh$\sigma$′ =
125.       update_BusHist$\Sigma$
125.       (obs_LorNm(licn),licn,leen″,(blid,bid),(bno,trace))(lbh$\sigma$) **in**
125.     l_to_l[leenm,obs_LorNm(licnm)] !
125.       Licensor_BusHist$\Sigma$Msg(b,blid,bid,lin,lee,tr);
125.     ((rl$\sigma$,sl$\sigma$,lbh$\sigma$′),bus$\sigma$)
125.     **end end end**
```

The Conduct Bus Ride Action

126. The conduct bus ride action prescribed by (ln,mkCon(bli,bi,t′) takes place in a context and shall have the following effect:
   (a) The action is performed by contractor li and at time t. This is known from the context.
   (b) First it is checked that the timetable in the contract named ln does indeed provide a journey, j, indexed by bli and (then) bi, and that that journey starts (approximately) at time t′ which is the same as or later than t.
   (c) Being so the action results in the contractor, whose name is "embedded" in ln, receiving notification of the bus ride commitment.
   (d) Then a bus, selected from a pool of available buses at the bust stop of origin of journey j, is given j as its journey script, whereupon that bus, as a behaviour separate from that of sub-contractor li, commences its ride.
   (e) The bus is to report back to sub-contractor li the times at which it stops at en route bus stops as well as the number (and kind) of passengers alighting and boarding the bus at these stops.
   (f) Finally the bus reaches its destination, as prescribed in j, and this is reported back to sub-contractor li.
   (g) Finally sub-contractor li, upon receiving this 'end-of-journey' notification, records the bus as no longer in actions but available at the destination bus stop.

The Cancel Bus Ride Action

127. The cancel bus ride action prescribed by (ln,mkCan(bli,bi,t′) takes place in a context and shall have the following effect:

(a) The action is performed by contractor li and at time t. This is known from the context.
(b) First a check like that prescribed in Item 126(b)) is performed.
(c) If the check is OK, then the action results in the contractor, whose name is “embedded” in ln, receiving notification of the bus ride cancellation.
That’s all !

The Insert Bus Ride Action

128. The insert bus ride action prescribed by (ln,mkIns(bli,bi,t′) takes place in a context and shall have the following effect:
(a) The action is performed by contractor li and at time t. This is known from the context.
(b) First a check like that prescribed in Item 126(b)) is performed.
(c) If the check is OK, then the action results in the contractor, whose name is “embedded” in ln, receiving notification of the new bus ride commitment.
(d) The rest of the effect is like that prescribed in Items 126(d))–126(g)).

The Contracting Action

129. The subcontracting action prescribed by (ln,mkLic(li′,(pe′,ops′,tt′))) takes place in a context and shall have the following effect:
(a) The action is performed by contractor li and at time t. This is known from the context.
(b) First it is checked that timetable tt is a subset of the timetable contained in, and that the operations ops are a subset of those granted by, the contract named ln.
(c) Being so the action gives rise to a contract of the form (ln′,li,(pe′,ops′,-tt′),li′). ln′ is a unique new contract name computed on the basis of ln, li, and t. li′ is a sub-contractor name chosen by contractor li. tt′ is a timetable chosen by contractor li. ops′ is a set of operations likewise chosen by contractor li.
(d) This contract is communicated by contractor li to sub-contractor li′.
(e) The receipt of that contract is recorded in the license state.
(f) The fact that the contractor has sublicensed part (or all) of its obligation to conduct bus rides is recorded in the modified component of its received contracts.

10.6.3 Discussion

10.7 Conclusion

It really is too early — in the development of the topic of this chapter — to conclude!

10.7.1 Achievements

What Did We Wish to Achieve?

Or rather, at this early, incomplete stage, what do we wish to achieve? In a first round we wish to achieve the following: an understanding of different kinds of license languages; an understanding of obligations and permissions (yet to be "designed" more explicitly into the three languages; a formalisation of both common aspects of the license systems (as a "vastly" distributed set of very many actors acting on even more licenses "competing" for resources, etc.), as well as of each individual language.

What Have We Achieved?

We think we have achieved what we set out to achieve.

What Do We Now Wish to Achieve?

First we would like to complete the full formalisation of each of the four languages: three license languages and one contract language. Based on those four formalisations we hope to be able to identify some common, better formalised, i.e., parametrised, license and contract concepts and thus to "lift" the four sets of syntaxes, well-formedness predicates and semantic functions into one set of parametrised functions and syntaxes. We think that given such four, widely separate examples and their parametrised "lifting" we can offer better contract and license language design, parametrised formalisations and common parametrised implementation software designs.

Part IV

Example Appendices

Appendices A–E (Pages 331–411) were respective appendices of JAIST Technical Memorandum [28]. Relative to the purposes of issuing [28] these appendices were meant as examples — of a sufficient “spread” — intended to “convince” management, i.e., the readers of [28], that domain engineering does indeed address their concerns.

A

Business Processes (BP) and BP Reengineering

Summary

By a business process we understand a behaviour, usually involving several actors, that is, staff of the domain enterprise(s), whose purpose it is to fulfil a business objective: a transaction, a production, a service.

By business process reengineering we understand a design that proposes changes to the current business processes of the domain enterprise(s).

We refer to Sect. 1.1.5 on page 10 for a first mentioning of the concept of business processes and business process reengineering.

We repeat that section:

Crucial elements in software engineering and in providing services to IT clients is that of identifying the business processes and suggesting the revision of business processes. With carefully worked-out domain descriptions the pursuit of business process engineering and reengineering takes on a far more professional rôle. We therefore claim that pursuing serious domain engineering helps consultancy firms better advise their clients.

A.1 Business Process Engineering

We rough-sketch a number of business process 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.

A.1.1 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 Cf. Fig. A.1.

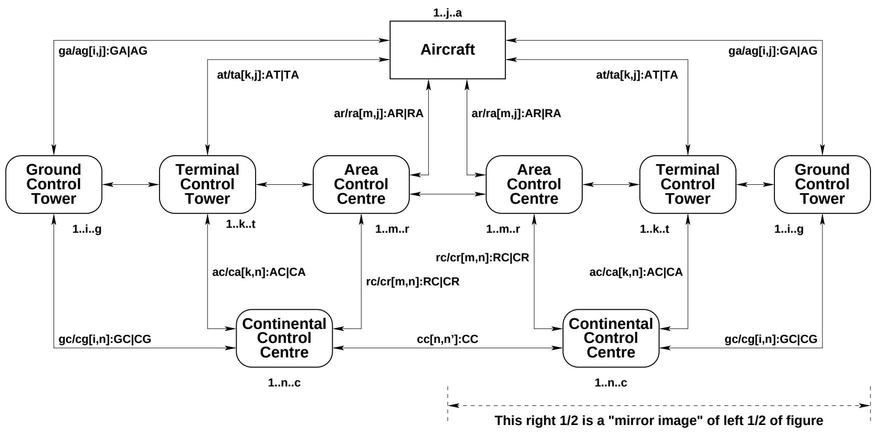


Fig. A.1. An air traffic behavioural system abstraction

We describe each of these behaviours separately:

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

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

(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.

A.1.2 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. A.2 on the next page). 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.

A.1.3 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. A.3 (Page 335)). 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

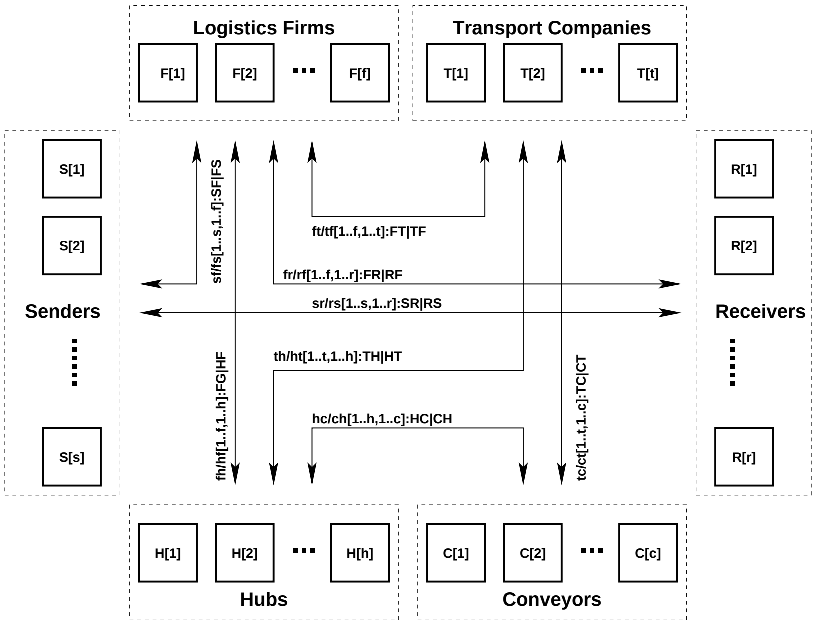


Fig. A.2. A freight logistics behavioural system abstraction

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.

A.1.4 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.

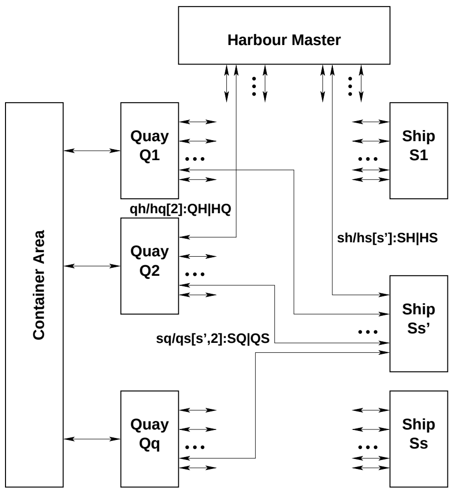


Fig. A.3. A harbour behavioural system abstraction

(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.

A.2 Business Process Reengineering Requirements

Characterisation 1 (Business Process Reengineering) By *business process reengineering* we understand the reformulation of previously adopted business process descriptions, together with additional business process engineering work.∎

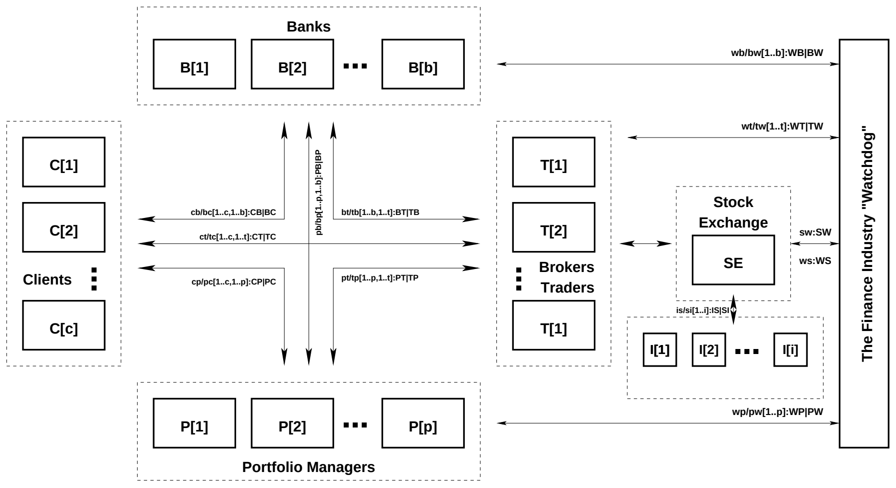


Fig. A.4. A financial behavioural system abstraction

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*.

A.2.1 Michael Hammer’s Ideas on BPR

Michael Hammer, a guru of the business process reengineering “movement”, states [119]:

1. *Understand a method of reengineering before you do it for serious.*

So this is what this appendix is all about!

2. *One can only reengineer processes.*

Clearly Hammer utters an untenable dogma!

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. We think it is very dangerous to not do a careful analysis of the business processes, yes, indeed of their entire domain!

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.*

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 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.

[1] A 'stretch goal' is a goal, an objective, for which, if one wishes to achieve that goal, one has to stretch oneself.

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?

A.2.2 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.

A.2.3 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.*

A.2.4 BPR and the Requirements Document

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.

Intrinsics Review and Replacement

Characterisation 2 (Intrinsics Review and Replacement) By *intrinsics review and replacement* we understand an evaluation as to whether current intrinsics stays or goes, and as to whether newer intrinsics need to be introduced.■

Example A.1 *Intrinsics Replacement:* A railway net owner changes its business from owning, operating and maintaining railway nets (lines, stations and signals) to operating trains. Hence the more detailed state changing notions of rail units need no longer be part of that new company's intrinsics while the notions of trains and passengers need be introduced as relevant intrinsics. ■

Replacement of intrinsics usually point to dramatic changes of the business and are usually not done in connection with subsequent and related software requirements development.

Support Technology Review and Replacement

Characterisation 3 (Support Technology Review and Replacement) By *support technology review and replacement* we understand an evaluation as to whether current support technology as used in the enterprise is adequate, and as to whether other (newer) support technology can better perform the desired services.■

Example A.2 *Support Technology Review and Replacement:* Currently the main information flow of an enterprise is taken care of by printed paper, copying machines and physical distribution. All such documents, whether originals (masters), copies, or annotated versions of originals or copies, are subject to confidentiality. As part of a computerised system for handling the future information flow, it is specified, by some domain requirements, that document confidentiality is to be taken care of by encryption, public and private keys, and digital signatures. However, it is realised that there can be a need for taking physical, not just electronic, copies of documents. The following business process reengineering proposal is therefore considered: Specially made printing paper and printing and copying machines are to be procured, and so are printers and copiers whose use requires the insertion of special signature cards which, when used, check that the person printing or copying is the person identified on the card, and that that person may print the desired document. All copiers will refuse to copy such copied documents — hence the special paper. Such paper copies can thus be read at, but not carried outside the premises (of the printers and copiers). And such printers and copiers can register who printed, respectively who tried to copy, which documents. Thus people are now responsible for the security (whereabouts) of possible paper 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. ■

Management and Organisation Reengineering

Characterisation 4 (Management and Organisation Reengineering) By *management and organisation reengineering* we understand an evaluation as to whether current management principles and organisation structures as used in the enterprise are adequate, and as to whether other management principles and organisation structures can better monitor and control the enterprise.■

Example A.3 *Management and Organisation Reengineering:* A rather complete computerisation of the procurement practices of a company is being contemplated. Previously procurement was manifested in the following physically separate as well as design-wise 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. ■

Rules and Regulations Reengineering

Characterisation 5 (Rules and Regs. Reengineering) By *rules and regulations reengineering* we understand an evaluation as to whether current rules and regulations as used in the enterprise are adequate, and as to whether other rules and regulations can better guide and regulate the enterprise.■

Here it should be remembered that rules and regulations principally stipulate business engineering processes. That is, they are — i.e., were — usually not computerised.

Example A.4 *Rules and Regulations Reengineering:* Assume now, due to reengineered support technologies, that interlock signalling can be made magnitudes safer than before, without interlocking. 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. ■

Human Behaviour Reengineering

Characterisation 6 (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 A.5 *Human Behaviour Reengineering:* A company has experienced certain lax attitudes among members of a certain category of staff. The progress of certain work procedures therefore is reengineered, implying that members of another category of staff are henceforth expected to follow up on the progress of "that" work.

In a subsequent domain requirements stage the above reengineering leads to a number of requirements for computerised monitoring of the two groups of staff. ■

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 7 (Script Reengineering) By *script reengineering* we understand evaluation as to whether the way in which rules and regulations are scripted and made known (i.e., posted) to stakeholders in and of the enterprise is adequate, and as to whether other ways of scripting and posting are more suitable for the enterprise.■

Example A.6 *Script Reengineering:* 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 called for, one that is first to be done in the basic business processes of a bank offering such services to its customers. ■

A.2.5 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.

General

Business process reengineering is based on the premise that corporations must change their way of operating, and, hence, must "reinvent" themselves. Some enterprises are "vertically" structured along functions, products or geographical regions. Others are "horizontally" structured along coherent business processes. In either case adjustments may need to be made as the business (i.e., products, sales, markets, etc.) changes.

B

Towards a Domain Model of Transportation

Summary

In this appendix we show a fragment of a domain model of transportation nets. The presentation follows some of our description dogmas: Precise, enumerated narratives: short sentences, carefully ordered and structured — followed by formalisations whose lines are related back to the enumerated narratives.

The entire appendix, Pages 343–369, covers only entities of the transportation domain. Thus there is no coverage of operations, events and behaviours. This, in the ears of most good programmers and of some algebraic specification language afficionados, is contrary to their 'beliefs'. Usually a data type is understood in terms of the operations upon data values. But for such very concrete phenomena as transport[ation] nets, such as we do indeed observe them, "out there", in the actual domain, for such entities we can spend a long time and quite some space just trying to understand these concrete phenomena.

For the uninitiated reader we annotate the RSL formula, that is, explain what these mean.

B.1 Net Topology

We conceptualise as segments the physically manifest phenomena of roads (between adjacent street intersections), rail tracks (between adjacent train stations), air-lanes (between adjacent airports) and shipping lanes (between adjacent harbours). We likewise conceptualise as junctions the physically manifest phenomena of street intersections, train stations, airports and harbours.

B.1.1 Nets, Segments and Junctions

1. Nets consists of one or more segments and two or more junctions.

```
type
   N, S, J
value
   obs_Ss: N → S-set
   obs_Js: N → J-set
axiom
   ∀ n:N • card obs_Ss(n) ≥ 1 ∧ card obs_Js(n) ≥ 2
```

Annotations:

- N, S, J are considered abstract types, i.e., sorts. N, S and J are type names, i.e., names of types of values. Values of type N are nets, values of type S are segments and values of type J are junctions.
- One can observe from nets, n, their (one or more) segments (obs_Ss(n)) and their (two or more) junctions (obs_Js(n)); n is a value of type N.
- Functions have names, obs_Ss, and obs_Cs, and functions, f, have signatures, f: A → B (not illustrated), where A and B are type names. A designates the definition set of f and B the range set.
- A-**set** is a type expression. It denotes the type whose values are finite, possibly empty set of A values.
- These observer functions are postulated.
- They cannot be formally defined.
- They are "defined" once a net has been pointed out[1]
- The axiom expresses that in any net there is at lest one segment and at least two junctions. ▪

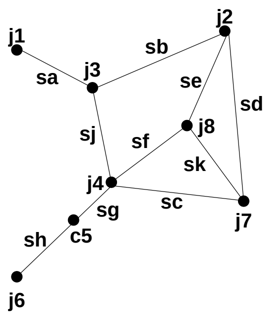


Fig. B.1. A simple net of segments and junctions

Applying the observer functions to the net of Fig. B.1 yields:

[1]Take the transportation net Europe. By inspecting it, and by deciding which segments and which associated junctions to focus on (i.e., "the interesting ones") we know which are all the interesting roads, rail tracks, air-lanes and shipping lanes, respectively the interesting (associated) street intersections, trains stations, airports and harbours.

obs_Ss(n) = {sa,sb,sc,sd,se,sf,sg,sh,sj,sk}
obs_Js(n) = {j1,j2,j3,j4,j5,j6,j7,j8}

Nets, segments and junctions are physically manifest, i.e., are phenomena.

B.1.2 Segment and Junction Identifications

2. We now assume that segments and junctions have unique identifications.

   type
   Si, Ji
   value
   obs_Si: S → Si
   obs_Ji: J → Ji

   Segment and junction identifications are mental concepts.
3. No two segments have the same segment identifier. And no two junctions have the same junction identifier.

   axiom
   ∀ n:N • **card** obs_Ss(n) ≡ **card** {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
   ∀ n:N • **card** obs_Js(n) ≡ **card** {obs_Ji(c)|j:J • j ∈ obs_Js(n)}

   Annotations:
   - **card** set expresses the cardinality of the set set, i.e., its number of distinct elements.
   - {f(a)|a:A • p(a)} expresses the set of all those B elements f(a) where a is of type A and has property p(a) [where we do not further state f, A and B. p is a predicate, i.e., a function, here from A into truth values of type **Bool**, for Boolean].
   - The axioms now express that the number of segments in n is the same as the number of segment identifiers of n — which is a circumscription for: No two segments have the same segment identifier.
   - Similar for junctions. ▪

   The constraints that limit identification of segments and junctions can be physically motivated: Think of the geographic (x, y, z co-ordinate) point spaces "occupied" by a segment or by a junction. They must necessarily be distinct for otherwise physically distinct segments and junctions. Segments may thus cross each other without the crossing point (in x, y space) being a junction, but, for example, one segment may, at the crossing point be physically above the other segment (tunnels, bridges, etc.).

B.1.3 Segment and Junction Reference Identifications

4. Segments are delimited by two distinct junctions. From a segment one can also observe, obs_Cis, the identifications of the delimiting junctions.

```
type
    Jip = {|{ji,ji′}:Ji-set • ji≠ji′|}
value
    obs_Jis: S → Jip
```

Annotations:

- {|a:A • p(a)|} is a subtype expression. It expresses a subset of type A, namely those A values which enjoys property p(a) [p is a predicate, i.e., a function, here from A into truth values in the type **Bool**]. In the above p(a) is ji≠ji′.
- In this case Jip is the subtype of Ji-**set** whose values are exactly 2 element sets of Ji elements. ■

5. Any junction has a finite, but non-zero number of segments connected to it. From a junction one can also observe, obs_Sis, the identifications of the connected segments.

```
type
    Si1 = {|sis:Si-set•card sis ≥1|}
value
    obs_Sis: J → Si1
```

Annotations:

- Si1 is the type whose values are non-empty, but still finite sets of Si values. ■

One cannot from a segment alone observe the connected junctions. One can only refer to them. Similarly: one cannot from a junction alone observe the connected segments. One can only refer to them. The identifications serve the role of being referents.

6. In any net, if s is a segment connected to connectors identified by ji and ji′, respectively, then there must exist connectors j and j′ which have these identifications and such that the identification si of s is observable from both j and j′.

```
axiom
    ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒
        let {ji,ji′} = obs_Jis(s) in
        ∃! j,j′:J • {j,j′}⊆obs_Js(n) ∧ j≠j′ ∧
            obs_Si(s) ∈ obs_Sis(c) ∩ obs_Sis(c′) end
```

Annotations:

- We read the above axiom:
  - ⋆ for all nets n and for all segments s in n
  - ⋆ let ji and ji′be the two distinct junction identifications observable from s, then
  - ⋆ exists exactly two distinct junctions, j and j′ of the net, such that
  - ⋆ the segment identification of s is in both the sets of segment identifications observable from j and j′.

■

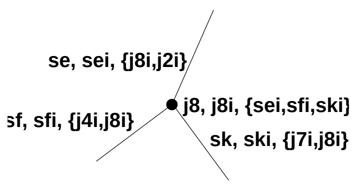


Fig. B.2. One junction and its connected segments

Figure B.2 illustrates the relation between observed identifications of segments and junctions.
The above constraints take on the mantle of being laws of nets: If segments and junctions otherwise have distinct identifications, then the above must follow as a law of man-made artifacts.

7. Vice-versa: In any net, if j is a junction connecting segments identified by si, si′, . . . , si′′ then there must exist segments s, s′, . . . , s′′ which have these identifications and such that the identification ji of j is observable from all s, s′, . . . , s′′.

```
axiom
  ∀ n:N, j:J • j ∈ obs_Js(n) ⇒
    let sis = obs_Sis(c), ji = obs_Ji(j) in
    ∃! ss:S-set • ss⊆obs_Ss(n) ∧ card ss=card sis ∧
    sis = {|obs_Si(s)|s:S•s ∈ ss|} end
```

Annotations:
- Let us read the above axiom:
  - ⋆ for all nets, n, and all junctions, j, of that net
  - ⋆ let sis be the set of segment identifications observed from j, and let ji be the junction identifier of j, then
  - ⋆ there exists a unique set, ss, of segments of n with as many segments as there are segment identifications in sis, and such that
  - ⋆ sis is exactly the set of segment identifications of segments in ss.

■

B.1.4 Paths and Routes

8. By a path we shall understand a triplet of a junction identification, a segment identification and a junction identification.

```
type
  P = Ji × Si × Ji
value
```

```
    paths: N → P-**set**
    paths(n) ≡
        {(ji,si,ji′) | s:S,ji,ji′:Ji,si:Si •
                s ∈ obs_Ss(n)∧{ji,ji′} ∈ obs_Jis(s)∧si=obs_Si(s)}
```

Annotations:

- Paths are modelled as Cartesians.
- One can generate all the paths of a net.
- It is the set of path triplets, two for each segment of the net and such that the pair of junction identifications, ji and ji′, observable from a segment is at either "end" of the triplet, and such that the segment identification is common to the two triplets (and in the "middle"). ∎

Paths, and as we shall see next, routes are mental concepts.

9. By a route of a net we shall understand a list, i.e., a sequence of paths as follows:
   - A sequence of just one path of the net is a route.
   - If r and r′ are routes of the net such that the last junction identification, ji, of the last path, (_,_,ji) of r and the first junction identification, ji′, of the first path (ji′,_,_) of r′ are the same, i.e., ji=ji′, then $r\widehat{\ }r'$ is a route.
   - Only routes that can be generated by uses of the first (the basis) and the second (the induction) clause above qualify as proper routes of a net.

```
type
    R = {|r:P*•wf_R(r)|}
value
    wf_R: P* → Bool
    wf_R(r) ≡
        ∀ i:Nat • {i,i+1}⊆inds(r) ⇒
            let (_,_,ji)=r(i), (ji′,_,_)=r(i+1) in ji = ji′ end

    routes: N → R-infset
    routes(n) ≡
        let rs = {⟨p⟩|p:P•p ∈ paths(n)}
                ∪ {r⌢r′|r,r′:R•{r,r′}⊆rs∧wf_R(r⌢r′)} in
        rs end
```

Annotations:

- Routes are well-formed sequences of paths.
- A sequence of paths is a well-formed route if adjacent path elements of the route share junction identification.
- Give a net we can compute all its routes as follows:
  - ⋆ let rs be the set of routes to be computed. It consists first of all the single path routes of the net.

⋆ Then rs also contains the concatenation of all pairs of routes, r and r′, such that these are members of rs and such that their concatenation is a well-formed route.
⋆ If the net is circular then the set rs is an infinite set of routes. The least fix point of the recursive equation in rs is the solution to the “routes” computation.

■

B.1.5 Segment and Junction Identifications of Routes

10. For future purposes we need be able to identify various segment and junction identifications as well as various segments and junctions of a route.

```
value
    xtr_Jis: R → Ci-set, xtr_Sis: R → Si-set
    xtr_Jis(r) ≡ case r of ⟨⟩ → {}, ⟨(ji,_,ji′)⟩⌢r′ → {ji,ji′}∪ xtr_Jis(r′) end
    xtr_Sis(r) ≡ case r of ⟨⟩ → {}, ⟨(_,si,_)⟩⌢r′ → {si}∪ xtr_Sis(r′) end

    xtr_Ss: N × Ji → S-set
    xtr_Ss(n,ji) ≡ {s|s:S•s ∈ obs_Ss(n) ∧ ji ∈ obs_Jis(s)}

    xtr_C: N × Ji → C, xtr_S: N × Si → S
    xtr_C(n,ji) ≡ let j:J • j ∈ obs_Js(n) ∧ ji=obs_Ji(j) in j end
    xtr_S(n,si) ≡ let s:S • s ∈ obs_Ss(n) ∧ si=obs_Si(s) in s end

    first_Ji: R →̃ Ji, last_Ji: R →̃ Ji
    first_Ji(r) ≡ case r of ⟨⟩ → chaos, ⟨(ji,_,_)⟩⌢r′ → ji end
    last_Ji(r) ≡ case r of ⟨⟩ → chaos, r′⌢⟨(_,_,ji)⟩ → ji end

    first_Si: R →̃ Si, last_Si: R →̃ Si
    first_Si(r) ≡ case r of ⟨⟩ → chaos, ⟨(_,si,_)⟩⌢r′ → si end
    last_Si(r) ≡ case r of ⟨⟩ → chaos, r′⌢⟨(_,si,_)⟩ → si end

    first_J: R × N →̃ J, last_J: R × N →̃ J
    first_J(r,n) ≡ xtr_J(first_Ji(r),n)
    last_J(r,n) ≡ xtr_J(last_Ji(r),n)

    first_S: R × N →̃ S, last_S: R × N →̃ S
    first_S(r,n) ≡ xtr_S(first_Si(r),n)
    last_S(r,n) ≡ xtr_S(last_Si(r),n)
```

Annotations:

- Given a route one can extract the set of all its junction identifications.
  - ⋆ If the route is empty, then the set is empty.

  - ⋆ If the route is not empty than it consists of at least one path and the set of junction identifications is the pair of junction identifications of the path together with set of junction identifications of the remaining route.
  - ⋆ Possible double "counting up" of route adjacent junction identifications "collapse", in the resulting set into one junction identification. (Similarly for cyclic routes.)
- Given a route one can similarly extract the set of all its segment identifications.
- Given a net and a junction identification one can extract all the segments connected to the identified junction.
- Given a net and a junction identification one can extract the identified junction.
- Given a net and a segment identification one can extract the identified segment.
- Given a route one can extract the first junction identification of the route.
  - ⋆ This extraction should not be applied to empty routes.
  - ⋆ A non-empty route can always be thought of as its first path and the remaining route. The first junction identification of the route is the first junction identification of that (first) path.
- Given a route one can similarly extract the last junction identification of the route.
- Given a route one can similarly extract the first segment identification of the route.
- Given a route one can similarly extract the last segment identification of the route.
- And similarly for extracting the first and last junctions, respectively first and last segments of a route. ■

B.1.6 Circular and Pendular Routes

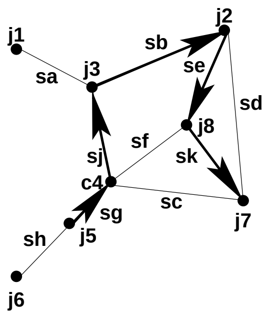
11. A route is circular if the same junction identification either occurs more than twice in the route, or if it occurs as both the first and the last junction identification of the route. Given a net we can compute the set of all non-circular routes by omitting from the above pairs of routes, r and r′, where the two paths share more than one junction identification.

```
non_circular_routes: N → R-**set**
non_circular_routes(n) ≡
  **let** rs = {⟨p⟩|p:P•p ∈ paths(n)}
          ∪ {r⌢r′|r,r′:R•{r,r′}⊆rs∧wf_R(r⌢r′)∧non_circular(r,r′)} **in**
  rs **end**
non_circular: R×R → **Bool**
non_circular(r,r′) ≡ **card** xtr_Jis(r) ∩ xtr_Jis(r′) =1
```

Annotations:

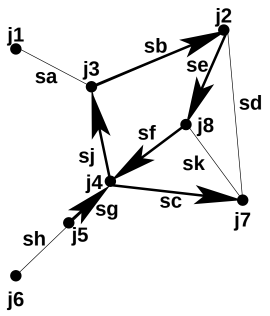
- To express the finite set of all non-circular routes
  - ⋆ is to re-express the set of all routes
  - ⋆ except constrained by the further predicate: non_circular.
- An otherwise well-formed route consisting of a first part r and a remaining part r′
  - ⋆ is non-circular if the two parts share at most one junction identification.

∎



<(j5i,sgi,j4i),(j4i,sji,j3i),(j3i,sbi,j2i),(j2i,sei,j8i)>

Fig. B.3. A route, graphically and as an expression



<(j5i,sgi,j4i),(j4i,sji,j3i),(j3i,sbi,j2i),(j2i,sei,j8i),(j8i,sfi,j4i),(j4i,sci,j7i)>

Fig. B.4. A circular route, graphically and as an expression

12. Let a path be (ji_f, si, ji_t), then (ji_t, si, ji_f) is a *reverse path.* That is: the two junction identifications of a path are reversed in the reverse path. A route, rr, is the reverse route of a route r if the ith path of rr is the

reverse path of the $n-i+1$'st path of r where n is the length of the route r, i.e., its number of paths. A route is a *pendular* route if it is of an even length and the second half (which is a route) is the reverse of the first half route.

```
value
    reverse: P → P
    reverse(jif,si,jit) ≡ (jit,si,jif)

    reverse: R → R
    reverse(r) ≡
        case r of
            ⟨⟩ → ⟨⟩,
            ⟨(jif,si,jit)⟩⌢r′ → reverse(r′)⌢⟨(jit,si,jif)⟩
        end

    reverse(r) ≡ ⟨reverse(r(i))|i in [n..1]⟩

    pendular: R → R
    pendular(r) ≡ r⌢reverse(r)

    is_pendular(r) ≡ ∃ r′,r″:R • r′⌢r″ = r ∧ r″=reverse(r′)
```

Annotations:

- The reverse of a path is a path with the same segment identification, but with reverse junction identifications.
- The reverse of a route, r, is
  - ⋆ the empty route if r is empty, and otherwise
  - ⋆ it is the reverse route of all of r except the first path of r concatenate (juxtaposed) with the singleton route of the reverse path of the first path of r.
- Given a route, r, we can construct a pendular route whose first half is the route r and whose last half is the reverse route of r.
- A (an even length) route is a pendular route if it can be expressed as the concatenation of two (equal length) routes, r′ and r″ such that r″ is the reverse of r′, that is, if its second half is the reverse of its first half. ▪

B.1.7 Connected Nets

13. A net is connected if for any two junctions of the net there is a route between them.

```
value
    is_connected: N → Bool
    is_connected(n) ≡
```

```
$\forall$ j,j$'$:J • {j,j$'$}$\subseteq$obs_Js(n) $\wedge$ j$\neq$j$'$ $\Rightarrow$
    **let** (ji,ji$'$) = (obs_Ji(j),obs_Ji(j$'$)) **in**
    $\exists$ r:R • r $\in$ routes(n) $\wedge$
        first_Ji(r) = ji $\wedge$ last_Ji(r) = ji$'$ **end**
```

Annotations:

- A net n is connected if
  - ⋆ for all two distinct connectors of the net
  - ⋆ where ji and ji$'$ are their junction identifications,
  - ⋆ there exists a route, r, of the net,
  - ⋆ whose first junction identification is ji and whose last junction identification is ji$'$.

∎

B.1.8 Net Decomposition

14. One can decompose a net into all its connected subnets. If a net exhaustively consists of m disconnected nets, then for any pair of nets in different disconnected nets it is the case that they share no junctions and no segments. The set of disconnected nets is the smallest such set that together makes up all the segments and all the junctions of the ("original") net.

```
**value**
    decompose: N $\to$ N-**set**
    decompose(n) **as** ns
        obs_Ss(n) = $\cup${obs_Ss(n$'$)|n$'$:N•n$'$ $\in$ ns} $\wedge$
        obs_Js(n) = $\cup${obs_Js(n$'$)|n$'$:N•n$'$ $\in$ ns} $\wedge$
        {} = $\cap${obs_Ss(n$'$)|n$'$:N•n$'$ $\in$ ns} $\wedge$
        {} = $\cap${obs_Js(n$'$)|n$'$:N•n$'$ $\in$ ns} $\wedge$
        $\forall$ n$'$:N•n$'$ $\in$ ns $\Rightarrow$ connected(n$'$) $\wedge$ ...
```

Annotations:

- A set ns of nets constitutes a decomposition of a net, n,
  - (a) if all the segments of n appear in some net of ns,
  - (b) if all the junctions of n appear in some net of ns,
  - (c) if no two or more distinct nets of ns share segments,
  - (d) if no two or more distinct nets of ns share junctions, and
  - (e) if all nets of ns are connected.
- **Comment:** It appears that items 3 and 4 are unnecessary, that is, are properties once items 1, 2 and 5 hold. ∎

That is, we have the following:

```
**Lemma:**
    $\forall$ n:N •
        **let** ns = decompose (n) **in**
        $\forall$ n$'$,n$''$:N • {n$'$,n$''$}$\subseteq$ns $\wedge$ n$'$$\neq$n$''$ $\Rightarrow$
        obs_Ss(n′) ∩ obs_Ss(n″) = {} ∧
        obs_Js(n′) ∩ obs_Js(n″) = {} end
```

The above 14 items define a lot of what there is to know about transportation nets if we only operate with the sorts that have been introduced (N, S, Si, J, Ji) and the observer functions that have likewise been introduced (obs_Ss, obs_Js, obs_Si, obs_Ji, obs_Jis and obs_Sis). The relationships between sorts, i.e., net, segment, segment identification, junction and junction identification values are expressed by the axioms. The above is a so-called property-oriented model of the topology of transportation nets. That model is abstract in that it does not hint at a mathematical model or at a data structure representation of nets, segments and junctions, let alone their topology. By topology we shall here mean how segments and junctions are "wired up". The axioms above guarantee that no segment of a net is left "dangling": It is always connected to two distinct junctions; and no junctions of a net is left isolated: It is always connected to some segments of the net.

We have tacitly assumed that all segments are two way segments, that is, transport can take place i either direction. Hence a segment gives rise to two paths.

B.2 Multi-Modal Nets

Interesting transportation nets are multi-modal. That is, consists of segments of different transport modalities: roads, rails, air-lanes, shipping lanes, and, within these of different categories. Thus roads can be either freeways, motorways, ordinary highways, and so on.

B.2.1 General Issues

15. We introduce a concept, M, of transport mode. M is a small set of distinct, but otherwise further undefined tokens. An m in M designates a transport modality.

```
type
   M
```

B.2.2 Segment and Junction Modes

16. With each segment, s, we can associate a single mode, m, and with each junction we can associate the set of modes of its connected segments.

```
value
   obs_M: S → M
   obs_Ms: J → M-set
```

```
axiom
  ∀ n:N, j:J • j ∈ obs_Js(n) ⇒
    let ss = xtr_Ss(n,obs_Ji(j)) in
    obs_Ms(j) = {obs_M(s)|s:S • s ∈ ss} end
  ∀ n:N, s:S • s ∈ obs_Ss(n) ⇒
    let {ji,ji′} = obs_Jis(s) in
    let {j,j′} = {xtr_J(n,ji),xtr_J(n,ji′)} in
    obs_M(s) ∈ obs_Ms(j) ∩ obs_Ms(j′) end end
```

Annotations:

- From a segment one can observe its mode.
- From a junction one can observe its set of modes.
- Let us read the first axiom:
  - ⋆ for all net, n, and all junctions, j, of that net
  - ⋆ let ss be the set of segments connected to j,
  - ⋆ now the set of modes of c is equal to the set of modes of the segments in ss.
- Let us read the second axiom:
  - ⋆ for all net, n, and all segments, s, of that net
  - ⋆ let ji and ji′ be the junction identifiers of the two junctions to which s is connected, and
  - ⋆ let j and j′ be the two corresponding junctions,
  - ⋆ then the segment mode is in both the set of modes of the two junctions.
- We can define a function, xtr_Ss, which from a net, n, and a junction identification, ji, extracts the set of segments, ss, connected to the junction identified by ji.
- xtr_Ss(n,ji) yields the set of segments, ss, in the net n for which ji is one of the observed junction identifications of s.
- And we can define a function, xtr_J, of signature N × Ji → J, which when applied to a net, n, and a junction identification, ji,
- extracts the junction in the net which has that junction identifier. ■

B.2.3 Single-Modal Nets and Net Projection

17. Given a multi-modal net one can project it onto a set of single modality nets, namely one for each modality registered in the multi-modal net.

```
type
  mmN = {|n:N • card xtr_Ms(n) > 1|}
  smN = {|n:N • card xtr_Ms(n) = 1|}
value
  xtr_Ms: N → M-set
  xtr_Ms(n) ≡ {obs_M(s) | s:S • s ∈ obs_Ss(n)}
```

```
projs: N → smN-set
projs(n) ≡ {proj(n,m) | m:M • m ∈ xtr_Ms(n)}

proj: N × M → smN
proj(n,m) as n′
   post
      let ss = obs_Ss(n), ss′ = obs_Ss(n′),
          js = obs_Js(n), js′ = obs_Js(n′) in
      ss′ = {s | s:S • s ∈ ss ∧ m=obs_M(s)} ∧
      js′ = {j | j:J • j ∈ js ∧ m ∈ obs_Ms(j)}
      end
```

Annotations:

- A multi-modal net is a net with more than one mode. mmN is thus the subtype of nets, n:N, which are multi-modal.
- A single-modal net is a net with exactly one mode. smN is thus the subtype of nets, n:N, which are multi-modal.
- The xtr_Ms function extracts the mode of every segment of a net.
- The projs function applies to any net, n:N, and yields the set of single-modal subnets of n, one for each mode of n. The projs function makes use of the proj function.
- The proj function applies to any n, n:N, and any mode of that net, and yields the single-modal subnet on n whose mode is the given mode.
  - ⋆ The proj function is expressed by a post condition, i.e., a predicate that characterises the necessary and sufficient relation between the argument net, n, and the result net n′.
  - ⋆ In a single-modal net, n′, projected from a multi-modal net, n, and of mode m, we keep exactly those segments, ss′, of n whose mode is m,
  - ⋆ and we keep exactly those junctions, js′, of n whose mode contains m.
  - ⋆ No more is needed in order to express the necessary and sufficient condition for a single-modal net to be a subnet of a proper net.
  - ⋆ That is, some single-modal nets are not proper nets since in proper nets every junction have the set of modes of all the segments connected to the junction.

■

B.3 Segment and Junction Attributes

B.3.1 Segment and Junction Attribute Observations

We now enrich our segments and junctions.

18. Segments have lengths.

19. Junctions have modality-determined lengths between pairs of (same such modality) segments connected to the junction.
20. Segments have standard transportation times, i.e., time durations that it takes to transport any number of units of freight from one end of the segment to the other.
21. Junctions have standard transfer time per modality of transport between pairs of segments connected to the junction.
22. Junctions have standard arrival time per modality of transport.
23. Junctions have standard departure times per modality of transport.
24. Segments have standard costs of transporting a unit of freight from one end of the segment to the other end.
25. Junctions have standard costs of transporting a unit of freight from the end of one connecting segment to the beginning of another connecting segment.

We can now assess

- (i) length of a route,
- (ii) shortest routes between two junctions,
- (iii) duration time of standard transport along a route, including transfer, stopover and possible reloading times at junctions, and
- (iv) shortest duration time route of standard transport between two junctions.

```
type
   L, TI
value
   ms:M-set,                        axiom ms≠{}
   obs_L: S → L
   obs_L: Si × J × M × Si → L
   obs_TI: S → TI
   obs_TI: Si × J × Si → TI
   obs_TI: J × M ~→ TI,         pre obs_TI(j,m): m ∈ obs_Ms(j)
   obs_TI: J × M × M ~→ TI,  pre obs_TI(j,m,m′): {m,m′}⊆obs_Ms(j)
   obs_arr_TI: J × M ~→ TI,    pre obs_arr_TI(j,m): m ∈ obs_Ms(j)
   obs_dep_TI: J × M ~→ TI,   pre obs_dep_TI(j,m): m ∈ obs_Ms(j)
   +: L × L → L
   +: TI × TI → TI
```

Annotations:

- L and Ti are sorts designating length and time values.
- ms denotes a non-empty set of modes.
- From a segment one can observe, obs_L, its length.
- From a segment one can observe, obs_TI, a time duration for a normal conveyor of the mode of the segment to travel the length of the segment.

- From a junction and a mode (of that junction) one can observe, obs_TI, a time duration for a normal conveyor of the mode to cross, i.e., to travel through the junction.
- From a junction and a pair of modes (m and m′ of that junction) one can observe, obs_TI, a time duration which represents the normal time it takes to transfer freight from a conveyor of mode m to a conveyor of mode m′. (The two modes may be the same.)
- From a junction and a mode (of that junction) one can observe, obs_arr_TI, a time duration for an item of freight destined for a normal conveyor of the mode to arrive and be "entry" processed (including loaded) at that junction.
- From a junction and a mode (of that junction) one can observe, obs_dep_TI, a time duration for an item of freight destined for a normal conveyor of the mode to arrive and be "exit" processed (including unloaded) at that junction.
- One can add lengths.
- One can add time durations. ∎

B.3.2 Route Lengths

26. One can compute the length of a route of a net and one can find the shortest such route between two identified junctions.

```
value
    length: R → N $\xrightarrow{\sim}$ L
    length(r)(n) ≡
        case r of
            ⟨⟩ → 0,
            ⟨(jf,si,jt)⟩ → obs_L(xtr_S(si,n)),
            ⟨(ji1,sii,ji2),(jj1,sij,jj2)⟩^r′ →
                let si=xtr_S(sii,n),sj=xtr_S(sij,n) in
                obs_L(si) +
                obs_L(sii,xtr_J(ji2,n),sij) +
                length(⟨(jj1,sij,jj2)⟩^r′)
        end end
        pre: r ∈ routes(n) ∧ ji2=jj1

value
    shortest_route: Ji × Ji → N $\xrightarrow{\sim}$ R
    shortest_route(jf,jt)(n) ≡
        let rs = routes(n) in
        let crs = {r|r:R•r ∈ rs∧first_Ji(r)=jf∧last_Ji(r)=jt} in
        let sr:R • sr ∈ crs∧∼∃ r:R•r ∈ crs∧length(r)(n)<length(sr)(n) in
        sr end end end
        pre: {jf,jt}⊆obs_Jis(n) ∧ jf≠jt
```

Annotations:

- The length of a single modality route of a net
  - ⋆ is 0 if the route is empty,
  - ⋆ otherwise it is the length of the first segment of the route plus the length of the rest of the route computed as follows:
    - ◇ If the route consists of just one segment, then 0,
    - ◇ else, the length of the junction from incident segment to emanating segment plus
    - ◇ the length of the rest of the route computed as otherwise specified above.
- The shortest route of a net between two of its identified junctions (the precondition) can be abstractly determined as follows:
  - ⋆ First we find all the routes, rs, of the net.
  - ⋆ Then we find those routes, crs, whose first and last connection identifications are the given ones, cf and ct.
  - ⋆ Amongst those we find a shortest one, that is, one, in crs, for which there are no shorter routes, r, in crs.

■

B.3.3 Route Traversal Times

27. One can find the total time it takes to traverse a route, including the times it takes to pass through a junction, and one can find the quickest route between two identified junctions.

```
all_time: R → N → TI
all_time(r)(n) ≡
    obs_arr_TI(xtr_J(first_J(r),n),obs_M(first_S{r}))
    + time(r)(n)
    + obs_dep_TI(xtr_J(last_J{r},n),obs_M(last_S(r)))

time: R → N → TI
time(r)(n) ≡
    case r of
        ⟨⟩ → 0,
        ⟨(jf,si,jt)⟩ → obs_TI(xtr_S(si,n)),
        ⟨(ji1,sii,ji2),(jj1,sij,jj2)⟩⌢r′ →
            let si=xtr_S(sii,n),sj=xtr_S(sij,n) in
            obs_TI(si) +
            obs_TI(sii,xtr_J(ji2,n),sij) +
            time(⟨(jj1,sij,jj2)⟩⌢r′)
    end end
    pre: r ∈ routes(n) ∧ ji2=jj1

quickest_route: Ji × Ji → N → R
```

```
quickest_route(jf,jt)(n) ≡
    let rs = routes(n) in
    let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=jf ∧ last_Ji(r)=jt} in
    let qr:R •
        qr ∈ crs∧∼∃ r:R•r ∈ crs∧all_time(r)(n)<all_time(qr)(n)
    in qr end end end
```

B.3.4 Function Lifting

28. Notice how the two functions shortest_route and quickest_route differ only by the length, respectively the time functions. Hence:

```
type
    Q
    FCT = R → N → Q
value
    less: Q × Q → Bool
    lowest: Ji × Ji → N → FCT → R
    lowest(jf,jt)(n)(fct) ≡
        let rs = routes(n) in
        let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=jf ∧ last_Ji(r)=jt} in
        let lr:R • lr ∈ crs ∧ ∼∃ r:R • r ∈ crs ∧ less(fct(r)(n),fct(qr)(n)) in
        lr end end end
```

29. Similarly one could also lift the 'less' predicate:

```
    Q
    PRE = Q × Q → Bool
    FCT = R → N → Q
value
    best: Ji × Ji → N → FCT → PRE → R
    best(cf,ct)(n)(fct)(pre) ≡
        let rs = routes(n) in
        let crs = {r|r:R•r ∈ rs ∧ first_Ji(r)=cf ∧ last_Ji(r)=ct} in
        let br:R • lr ∈ crs ∧ ∼∃ r:R • r ∈ crs ∧ pre(fct(r)(n),fct(qr)(n)) in
        br end end end
```

And so on.

B.3.5 Transportation Costs

30. We can further assess (i) transport costs, (ii) lowest (per unit) freight cost between two junctions, etc. We assume that if a freight item is transported

into a junction and out of that junction by the same modality conveyour, then it is not reloaded, i.e., along segments of the same modality.[2]

```
type
    K, F
value
    obs_K: (S|J) → K
    obs_F: (S|J) → F

    +: K × K → K

    cost: R → N → K
    cost(r)(n) ≡
        case r of
            ⟨⟩ → 0,
            ⟨(jf,si,jt)⟩ →
                obs_K(xtr_J(jf,n)) +
                obs_K(xtr_S(si,n)) +
                obs_K(xtr_J(jt,n))
            ⟨(jf,si,jt),(jf′,si′,jt′)⟩⌢r′ → assert: jt=jf′
                obs_K(xtr_J(jf,n)) +
                obs_K(xtr_S(si,n)) + ... +
                cost(r′)
        end

    cheapest: Ji×Ji → N → ((K×K)→K) → ((K×K)→Bool) → R
    cheapest(jf,jt)(n) ≡
        best(jf,jt)(n)(λ(k1,k2):(K×K)•k1+k2)(λ(k1,k2):(K×K)•k1<k2)
```

B.4 Road Nets

We wish to view road nets at different levels of abstraction. At a most detailed such level we make no distinction between the road kinds, whether community roads, provincial roads, motor roads or freeways. At another level of abstraction we wish to make exactly those distinctions. And at least detailed level of abstraction we consider certain road junctions to designate road nets of smaller or larger communities.

31. Figure [A] B.5 on the next page shows a road net. Instead of showing junctions J1, J2 and J3 as small black disks we show them as larger circles — for reasons that transpires from Fig. [B] B.5 on the following page.

[2]This grossly simplifying assumption will be removed later. For the time being it allows us to operate with the simple notion of routes that was introduced above. For the reloading case we need to decorate the route notion, effectively making it into a bill of ladings notion: one that prescribes possible reloading at junctions.

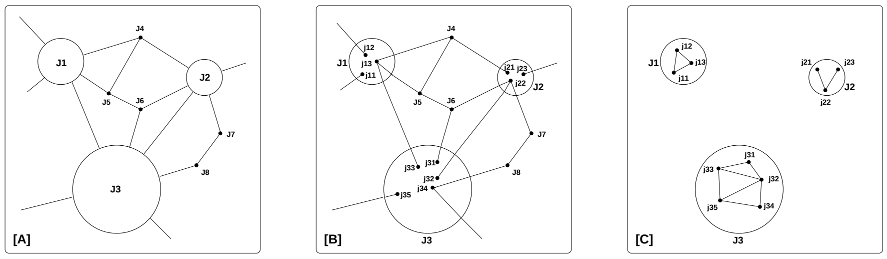


Fig. B.5. Gross [A] versus semi-detailed [B] road net — and community road nets [C]

32. Junctions J1, J2 and J3 are considered composite, that is, to represent communities.
33. We may consider the road net of Fig.[A] B.5 to be an abstraction of the road net hinted at in Fig.[B] B.5.
34. Junctions j11, j12, . . . , j35 are considered simple embedded junctions.
35. We decide to allow three kinds of junctions:
    (a) composite,
    (b) simple embedded and
    (c) simple.

    They are as follows:
    (a) Composite junctions stand for road nets themselves. The junctions of those road nets are all simple embedded junctions.
    (b) Simple embedded junctions are the junctions, hence, of composite junction road nets.
    (c) Simple junctions are those junctions which are not composite (that is: are not standing for road nets) and are not simple embedded junctions (that is: simple, hence un-embedded junctions are those remaining junctions of a net which include modality road).
36. In Fig. [B] B.5 we have left out the internal roads, that is, segments of junctions J1, J2 and J3, that is between the simple embedded junctions j11, j12 and j13, between j21, j22 and j23, and between j31, j32, j33, j34 and j35.
37. The internal segments of junctions J1, J2 and J3 are shown in Fig. [C] B.5. They are to be considered complete nets "in and by" themselves.
38. We may consider the implied junction identifications Ji1, Ji2 and Ji3 to be names of communities.
39. We may consider the implied junction identifications ji11, ji12 and ji13 to abstract to J1, ji21, ji22 and ji23 to abstract to J2, and ji31, ji32, ji33, ji34 and ji35 to abstract to J3.
40. We shall assume that from these junction identifications, say jikℓ, one can observe the more abstract junction identifications, i.e., Jik.

41. We shall, conversely, assume that from segment junction identifications one can observe whether they are identifications of composite, of simple embedded or of simple junctions, and, if of composite junctions, that one can further observe which simple embedded junction of the composite junction the segment is connected to.
42. In summary: When consider any multi-modality net and from it project, that is, consider only the net, n_r, of modality road, then we may find that some junctions are composite while are are simple. When then examining the road nets, r_n, contained in composite junctions then we will find that their junctions are simple embedded. The embedded road nets, r_n, otherwise satisfy all the properties (i.e., axioms) of nets in general. To link up the segments of n_r incident upon, that is, connected to composite junctions (in n_r) we provide their junction identifications with two levels of observability: the abstract one that made us see that they were connected to composite junctions (cf. Fig. [A] B.5 on the preceding page), and a concrete one that enables us to decide which ones of the simple embedded junctions they are "finally" linked to (cf. Fig. [B] B.5 on the facing page).

```
type
    M == road | ...
    Jc, Js, Jse
    Jic, Jis, Jise
    J = Jc | Js | Jse
    Cn
value
    is_composite_J: J → Bool
    is_simple_J: J → Bool
    is_simple_embedded_J: J → Bool
    obs_N: Jc → N
    obs_Jic: Jc → Jic, obs_Jis: Js → Jis, obs_Jise: Jse → Jise
    obs_Cn: Jic → Cn, obs_Cn: Jise → Cn
    obs_Jise: Jic → Jise
axiom
    ∀ j:Jc • is_composite_J(j) ∧ xtr_Ms(obs_N(j,road))={road},
    ∀ j:Js • is_simple_J(j),
    ∀ j:Jse • is_simple_embedded_J(j)

    ∀ n:N,j:J • j ∈ obs_Js(n) ∧ is_composite_J(j) ⇒
        let rn = obs_N(j) in

        end
```

B.5 Railway Nets

B.5.1 General

A transportation net of modality railway has segments be lines between stations and have junctions be stations.

43. We concretise the concept of modes. Mode m=railway will now designate railway nets:

    type
    M == road | railway | ...

44. From a multi-modal transportation net we can project the railway net, rn:RN:

    value
    proj: N $\times$ {railway} $\to$ RN

45. Junctions of a transportation net of modality railway have sub-junctions which are stations:

    value
    proj: J $\times$ {railway} $\to$ ST

46. Segments of a transportation net of modality railway become lines:

    value
    proj: S $\times$ {railway} $\to$ LI

B.5.2 Lines, Stations, Units and Connectors

Railway segments are thus called lines, and railway sub-junctions are thus called stations. A notion of connectors is introduced. It is not to be confused with the previous notion of junctions.

47. A railway net is a net of mode railway.
48. Its segments are lines of mode railway.
49. Its junctions are stations of mode railway.
50. A railway net consists of one or more lines and two or more stations.
51. A railway net consists of rail units.
52. A line is a linear sequence of one or more linear rail units.
53. The rail units of a line must be rail units of the railway net of the line.
54. A station is a set of one or more rail units.
55. The rail units of a station must be rail units of the railway net of the station.
56. No two distinct lines and/or stations of a railway net share rail units.

57. A station consists of one or more tracks.
58. A track is a linear sequence of one or more linear rail units.
59. No two distinct tracks share rail units.
60. The rail units of a track must be rail units of the station (of that track).
61. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.
62. A rail unit has one or more connectors.
63. A linear rail unit has two distinct connectors. A switch (a point) rail unit has three distinct connectors. Crossover rail units have four distinct connectors (whether simple or switchable), etc.
64. For every connector there are at most two rail units which have that connector in common.
65. Every line of a railway net is connected to exactly two distinct stations of that railway net.
66. A linear sequence of (linear) rail units is an acyclic sequence of linear units such that neighbouring units share connectors.

```
type
47.  RN  = {| n:smN • obs_M(n)=railway |}
48.  LI  = {| s:S • obs_M(s)=railway |}
49.  ST  = {| c:C • obs_M(c)=railway |}
   Tr, U, K

value
   50.  obs_LIs: RN → LI-set
   50.  obs_STs: RN → ST-set
   51.  obs_Us: RN → U-set
   52.  obs_Us: LI → U-set
   54.  obs_Us: ST → U-set
   57.  obs_Trs: ST → Tr-set
   61.  is_Linear: U → Bool
   61.  is_Switch: U → Bool
   61.  is_Simple_Crossover: U → Bool
   61.  is_Switchable_Crossover: U → Bool
   62.  obs_Ks: U → K-set

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

axiom
```

```
50. $\forall$ n:RN $\bullet$ **card** obs_LIs(n) $\geq 1 \wedge$ **card** obs_STs(n) $\geq 2$

52. $\forall$ n:RN, l:LI $\bullet$ l $\in$ obs_LIs(n) $\Rightarrow$ lin_seq(l)

53. $\forall$ n:RN, l:LI $\bullet$ l $\in$ obs_LIs(n) $\Rightarrow$ obs_Us(l) $\subseteq$ obs_Us(n)

54. $\forall$ n:RN, s:ST $\bullet$ s $\in$ obs_STs(n) $\Rightarrow$ **card** obs_Us(s) $\geq 1$

55. $\forall$ n:RN, s:ST $\bullet$ s $\in$ obs_LIs(n) $\Rightarrow$ obs_Us(s) $\subseteq$ obs_Us(n)

56. $\forall$ n:RN,l,l$'$:LI$\bullet$\{l,l$'$\}$\subseteq$obs_LIs(n)$\wedge$l$\neq$l$'$$\Rightarrow$obs_Us(l)$\cap$ obs_Us(l$'$)=\{\}

56. $\forall$ n:RN,l:LI,s:ST$\bullet$l $\in$ obs_LIs(n)$\wedge$s $\in$ obs_STs(n)$\Rightarrow$obs_Us(l)$\cap$ obs_Us(s)=\{\}

56. $\forall$ n:RN,s,s$'$:ST$\bullet$\{s,s$'$\}$\subseteq$obs_STs(n)$\wedge$s$\neq$s$'$$\Rightarrow$obs_Us(s)$\cap$ obs_Us(s$'$)=\{\}

57. $\forall$ s:ST$\bullet$**card** obs_Trs(s)$\geq$1

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

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

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

65. $\forall$ n:RN,l:LI $\bullet$ l $\in$ obs_LIs(n) $\Rightarrow$
     $\exists$ s,s$'$:ST $\bullet$ \{s,s$'$\} $\subseteq$ obs_STs(n) $\wedge$ s$\neq$s$'$ $\Rightarrow$
        **let** sus=obs_Us(s),sus$'$=obs_Us(s$'$),lus=obs_Us(l) **in**
        $\exists$ u,u$'$,u$''$,u$'''$:U $\bullet$ u $\in$ sus $\wedge$
           u$'$ $\in$ sus$'$ $\wedge$ \{u$''$,u$'''$\} $\subseteq$ lus $\Rightarrow$
           **let** sks = obs_Ks(u), sks$'$ = obs_Ks(u$'$),
               lks = obs_Ks(u$''$), lks$'$ = obs_Ks(u$'''$) **in**
           $\exists$!k,k$'$:K$\bullet$k$\neq$k$'$$\wedge$sks $\cap$ lks=\{k\}$\wedge$sks$'$ $\cap$ lks$'$=\{k$'$\}
        **end end**
```

B.6 Net Dynamics

By net dynamics we shall mean the changing possibilities of flow of conveyors (cars, trains, aircraft, ships, etc.) along segments and through junctions. We speak of direction of flow along segments in terms of *"from the junction at one end of the segment to the junction at the other end"*. And we speak of flow through a junction as *"proceeding from one segment incident upon the junction into a (usually different) segment emanating from that junction"*. Segments connected to a junction are both incident upon that junction and emanates from that junction.

B.6.1 Segment and Junction States

67. Segments may be open for traffic in either or both directions (between the segments' two junctions [identified by ji_x and ji_y]) or may be closed.
68. We model the state, $s\sigma : S\Sigma$, of a segment, $s : S$, as a set of pairs of junction identifications, namely of the two identifications of the junctions that the segment connects. This state, $s\sigma : S\Sigma$, is
    (a) either empty, i.e., the segment is closed ($\{\}$),
    (b) or has one pair, $\{(ji_x, ji_y)\}$, that is, the segment is open in direction from junction ji_x to junction ji_y,
    (c) or another pair $\{(ji_y, ji_x)\}$,
    (d) or both pairs $\{(ji_x, ji_y), (ji_y, ji_x)\}$, that is, is open in both directions.
69. Junctions may direct traffic from any subset of incident segments to any subset of emanating segments.
70. We model the state, $j\sigma : J\Sigma$, of a junction, $j : J$, as a set of pairs of segment identifications, namely of identifications of segments connected to the junction.
    (a) Let the set of identifications of segments connected to junction j be $\{si_1, si_2, ..., si_m)\}$.
    (b) If, in some state, $j\sigma$ of the junction, it is possible (allowed) to pass through the junction from the segment identified by si_j to the segment identified by si_k, then the pair (si_j, si_k) is in $j\sigma$.
    (c) The junction state may be empty, i.e., closed: no traffic is allowed through the junction.
    (d) Or the junction state may be "anarchic full", that is, it contains all combinations of the pairs of identifiers of segments incident upon the junction.

```
type
    SΣ = (Ji×Ji)-set
    JΣ = (Si×Si)-set
value
    obs_SΣ: S → SΣ
    obs_JΣ: J → JΣ

    xtr_Jis: SΣ → Ji-set
    xtr_Jis(sσ) ≡ {ji|ji:Ji • (ji,_) ∈ obs_sσ ∨ (_,ji) ∈ obs_sσ}
    xtr_Sis: JΣ → Si-set
    xtr_Sis(jσ) ≡ {si|si:Si • (si,_) ∈ obs_jσ ∨ (_,si) ∈ obs_jσ}
axiom
```

∀ s:S • xtr_Jis(obs_SΣ(s)) ⊆ xtr_Jip(s),
∀ j:J • xtr_Sis(obs_JΣ(j)) ⊆ xtr_Sis(j)

Observations:

- A junction, $j : J$, of just one segment, $s : S$, that is, s is a cul de sac, may either be closed, and vehicles trying to enter j will be queued up, or it is open, and vehicles entering j will be lead back to s.
- As a consequence segment s, in order for this latter routing to happen, must be open in both directions when j is "open".
- In general, if the state of a junction j (identified by ji) contains a pair (si_x, si_y) then the state of the designated segments, sx and sy, must respectively contain pairs (ji', ji), respectively (ji, ji''), where $\{ji, ji'\}$ and $(ji, ji''\}$ are the pairs of junction identifications associated with si_x and si_y respectively.
- And this must hold for all states of junctions and adjacent segments.
- This is captured in the axioms below.

axiom
...

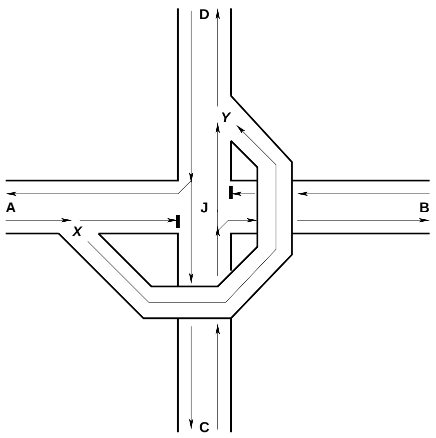


Fig. B.6. A Special "Carrefour" Junction

71. The junction of Fig. B.6 shows four segments, identified by A, B, C and D.
72. The figure also suggests a state in which traffic lights prohibit movements from A into J, from B into J,
73. from C via J into A, and from D via J into B.

74. The “bypass” from A/X into Y/D appears to be such that traffic can always pass from A into D.
75. The current state alluded to in Fig. B.6 on the preceding page appears to be:

$$j\sigma_J : \{(A,D),(C,B),(C,D),(D,A),(D,C)\}$$

76. (A,D) is potentially a member of every state that the junction can possibly be in — see next section.

B.6.2 Segment and Junction State Spaces

```
**type**
    S$\Omega$ = S$\Sigma$-**set**
    J$\Omega$ = J$\Sigma$-**set**
**value**
    obs_S$\Omega$: S → S$\Omega$
    obs_J$\Omega$: J → J$\Omega$
**axiom**
    ∀ s:S • obs_S$\Sigma$(s) ⊆ obs_S$\Omega$(s),
    ∀ j:J • obs_J$\Sigma$(j) ⊆ obs_J$\Omega$(j)
```

Etcetera!

B.7 Conclusion

We have shown but a fragment of a carefully worked-out transport net domain model of just some simple entities and some auxiliary functions over these. By saying that we mean to alert the reader to the fact that working out, i.e., experimentally resarching and developing any realistic domain model takes time and the result takes much space to present. But it is fun.

C

Towards a Domain Model of Manufacturing

Summary

We bring a fragment of a domain model of manufacturing. The model, in effect, is also, more generally, a model of data flow. And, hence, the model is really a model of applied Petri nets [148, 199, 210–212].

C.1 Introduction

C.1.1 Definitions

In this chapter we present a model of a number of aspects of manufacturing. By manufacturing Merriam-Webster's (MW) Collegiate Dictionary understands: *to make into a product suitable for use; to make from raw materials by hand or by machinery; to produce according to an organized plan and with division of labor.* Anther term is production. Again MW, amongst several alternatives, understands: *the act or process of producing; the creation of utility; especially: the making of goods available for use.* We equate the composite terms: manufacturing plant, production facility and factory. The latter, according to MW: *a building or set of buildings with facilities for manufacturing; the seat of some kind of production.* By plant, in our context, MW means: *the land, buildings, machinery, apparatus, and fixtures employed in carrying on a trade or an industrial business; a factory or workshop for the manufacture of a particular product; the total facilities available for production or service; the buildings and other physical equipment of an institution.*

Central to the concept of manufacturing are the concepts of machines and products. By a machine MW means: *an assemblage of parts that transmit forces, motion, and energy one to another in a predetermined manner; an instrument (as a lever) designed to transmit or modify the application of power, force, or motion; a mechanically, electrically, or electronically operated* *device for performing a task*. By a product MW means: *something produced (produce: to compose, create, or bring out by intellectual or physical effort).* Central to the concept of production is the concept of part: *one of the often indefinite or unequal subdivisions into which something is or is regarded as divided and which together constitute the whole; an essential portion or integral element; one of several or many equal units of which something is composed or into which it is divisible* (MW).

C.1.2 Examples of Machines

The concept of machine in this chapter is best understood by bringing some examples: lathe, band saw, belt sander, milling machine, drill press, grinder, shear, notscher, and press brake. If you are not familiar with these names perhaps a look at:

- http://www-me.mit.edu/Lectures/MachineTools/outline.html

might help!

C.1.3 Structure of Chapter

General

From models of "smallest", atomic, phenomena and concepts we build up models of increasingly more complex phenomena and concepts, ending with models of manufacturing plants.

These models are very general. The reader may think: far too general. That may very well be so. In Sect. C.7 we shall instantiate our models to models of manufacturing plants that are claimed to be typical of specific factories.

Reading Guide

The text consists of sequences of one, two, three or four sub-texts. Always a narrative explication of a phenomenon or a concept. Additionally a formal model of that phenomenon or a concept. Then, in most cases, at least in this chapter, an annotation which explains the formal notation. And, sometimes some observations. Readers with a background in formal specification languages a la B [1, 71], RAISE's RSL [31–33, 44, 101, 104, 106], VDM-SL [55, 56, 95, 96] or Z [132, 133, 229, 230, 242] can skip the annotations. The formal notation that is used is RSL [104]. We refer to [31–33] for a thorough introduction to abstract and modelling using RSL.

C.2 Parts

77. An atomic part is a smallest unit of man-made production.

```
type
   P
value
   is_atomic_P: P → Bool
```

Annotations:

- P is a type name, that is, stands for a set of values. We shall think of these values as parts.
- is_atomic is an observer function, i.e.., a postulated predicate which when applied to parts (i.e., to entities of type P) yield truth if they are atomic, false otherwise. ■

78. An atomic part has a part number, as has all parts, whether atomic or composite.

```
type
   Pn
value
   obs_Pn: P → Pn
```

Annotations:

- Pn is a type name, that is, a set of values. We shall think of these values as part numbers.
- obs_Pn is an observer function, that is a postulated function which when applied to values of type P yields their part numbers. ■

79. A composite part consists of two or more parts which have been manufactured (fitted, assembled, welded, etc.) together according to some mereology.

```
value
   is_composite_P: P → Bool
axiom
   ∀ p:P•
      is_atomic_P(p)∧∼is_composite_P(p) ∨
      is_composite_P(p)∧∼is_atomic_P(p)
```

80. A composite part may have two or more occurrences (components) of parts of the same part number. Mereologically they appear (occur) in distinct "locations" of the whole part. We abstract locations (etc.) by associating with each part a unique identification, $\pi : \Pi$.

```
type
    $\Pi$
    COMPS = P-set
value
    obs_$\Pi$: P $\rightarrow$ $\Pi$
    obs_COMPS: P $\rightarrow$ COMPS

    no_of_occurrences: P $\times$ P $\xrightarrow{\sim}$ Nat
    no_of_occurrences(cp,p) $\equiv$
        card {p'|p':P • p' $\in$ obs_COMPS(cp) $\wedge$ obs_Pn(p')=obs_Pn(p)}
        pre: is_composite_P(cp)
axiom
    $\forall$ p:P •
        is_atomic_P(p) $\Rightarrow$ obs_COMPS(p)={} $\wedge$
        is_composite_P(p) $\Rightarrow$
            obs_COMPS(p)$\neq${} $\wedge$
            $\forall$ p',p'':P • {p',p''}$\subseteq$dom obs_COMPS(p) $\wedge$ p$\neq$p' $\wedge$
                p'$\neq$p'' $\Rightarrow$ obs_$\Pi$(p') $\neq$ obs_$\Pi$(p'') $\wedge$
            obs_$\Pi$(p) $\neq$ obs_$\Pi$(p')
```

Annotations:

- *Π* is a type name. We shall think of these values as part identifiers.
- COMPS is a type name. Its values are sets of parts.
- obs_*Π* names an observer function which, when applied to values of type part yields their part identification.
- obs_COMPS names an observer function.
- no_of_occurrences names a function which, when applied to a pair of parts, (cp,p) yields the number of occurrences of p in cp — where cp is assumed to be a composite part. The function is not defined for cp being atomic. If p is not a sub-component of cp then the function value is zero.
- When obs_COMPS is applied to values of type part yields the set of their sub-components. An atomic part has no such. A composite part has one or more sub-components which are parts (i.e., are part values). Part identifiers of distinct subcomponents are distinct and different also from the "mother" component.
- In the following we shall say no more about part identifiers, and hence we consider them atomic. ■

81. From the above we observe that no two physically manifest parts are identical: They may be of the same kind, i.e., part number, but they will, as a "law of nature" have distinct identifications. We are not saying that these identifications are physically manifest things, i.e., "labels" the part. We are saying that these identifications are concepts.[1]

[1]It is like the coins in your pocket: There may be several instances of a shilling but they are all distinct in space (and otherwise).

C.3 Machines

82. A machine basically offers a function which takes a non-zero number of parts and produces a non-zero number of other, distinct parts.

type
  Parts′ = P-**set**
  Parts = {| parts:Parts′ • parts≠{} |}
  MOp
  MFct = Parts → Parts
value
  obs_MFct: MOp → MFct
axiom
  ∀ mop:MOp • {obs_Pn(p)|p:P•p ∈ $\mathcal{D}$} ∩ {obs_Pn(p)|p:P•p ∈ $\mathcal{D}$} = {}

$\mathcal{D}$, the definition set function, is a non-computable function. So is $\mathcal{R}$, the range set function.
Annotations:

- Parts is a type name which denotes a non-empty set of parts.
- MOp is a type name which denotes a set of machine operations.
- MFct is a type name which denotes a set of machine functions. Machine functions are total functions which when applied to a set of not necessarily part number distinct parts yields a set of not necessarily part number distinct parts. Thus a machine operation (i.e., machine function) may take more than one part of a given part number, and may yield not only one or more parts, but also such that two or more of these may have the same part number.
- obs_MFct names an observer function which when applied to a machine operation yields a machine function.
- None of the yielded parts of a machine function have the same part number as any of the input parts. ■

83. We can characterise the functionality of a machine by the signature of the machine function mop:MOp.

type
  Config′ = Pn $\overrightarrow{m}$ **Nat**
  Config = {| c:Config′ • 0∉ **rng** c |}
  Input,Output = Config
  MSig′ = Input × Output
  MSig = {| (i,o):MSig′ • **dom** i ∩ **dom** o = {} |}

Annotations:

- Config is a type name. It denotes a map from part numbers to non-zero natural numbers.

- Input and Output are type names of the same configuration type.
- MSig is a type name. It denotes the set of pairs of respectively input and output configurations where none of the input part numbers are the same as the output part numbers. The idea is that MSig designates a machine function signature. If in an input configuration, i, part number pn_j maps into quantity $\mathrm{n} = \mathrm{i}(\mathrm{pn}_j)$ then the machine function to which the machine signature will be associated (see below) shall require n occurrences of parts $\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_n$ — all having the same part number pn_j. ∎

84. A machine proper can pragmatically be thought of as an “optional formal machine and zero, one or more workers”. If the “optional formal” machine is not there, then the machine “embodies” at least one worker. In any case we consider the “optional formal machine and zero, one or more workers” as a unit. The machine, in addition to its machine operation[2], can be thought of as also being represented by the machine signature and a machine in-tray and a machine out-tray. The in- and out-trays, at one one moment, consists of zero, one or more parts. The parts may, or may not be relevant to the machine operation. Usually they are. We shall, in fact, expect that the in-tray [out-tray] parts are of the kind (i.e., having the input [output] part numbers) of the machine signature.

```
type
   MC
   MACH′ = Input × MSig × MOp × Output
   MACH = {| mach:MACH′ • wf_MACH(mach) |}
value
   obs_MACH: MC → MACH
   wf_MACH(i,(isig,osig),op,o) ≡
      dom i ⊆ dom isig ∧ dom o ⊆ dom osig ∧
      dom i = {pn | pn:Pn • ∃ p:P • p ∈𝒟(op)∧pn=obs_Pn(p)}∧
      dom o = {pn | pn:Pn • ∃ p:P • p ∈ℛ(op)∧pn=obs_Pn(p)}
```

Annotations:

- MC is a type name. It denotes a sort.
- MACH is a type name. It denotes the set of machine well-formed quadruples (i.e., Cartesians) of inputs, machine signatures, (associated) machine operations, and outputs. The idea is that inputs stand for machine in-trays, outputs for machine out-trays, and that the machine signature is associated the machine operation.
- obs_MACH names an observer function which, when applied to values of type MC yields values of type MACH.

[2]The machine operation is performed either by the “optional formal machine” or the “optional formal machine and one or more workers” or, when the “optional formal machine” is not there, the “one or more workers”.

- wf_MACH names a predicate. Its definition expresses the well-formedness of machine. A machine quadruple (i,(isig,osig),op,o) is well-formed if the in-tray [out-tray] does not contain parts of types that are not of interest to the machine (that is, which does not contain parts which (for i:) are not needed in order for the machine to perform its operation [(for o:) are not yielded by the machine operation]).
- Note: We have modelled the occurrence of actual parts in the trays, not by their real sets of parts, but by a recording of how many parts there is of given part numbers. ▪

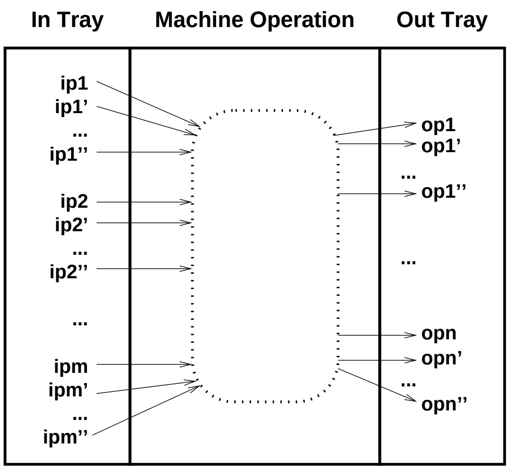


Fig. C.1. A schematic machine

85. Figure C.1 intends to illustrate that a machine can be considered to consist of an in tray, a facility for performing the machine operation and an out tray.

C.4 Machine Operation

86. Figure C.1 also intends to show that a machine operation consumes one or more occurrences (ip1, ip1′, ..., ip1″) of parts of one part number (ip_1), one or more occurrences (ip2, ip2′, ..., ip2″) of parts of another part number (ip_2), etc., and one or more occurrences (ipm, ipm′, ..., ipm″) of parts of yet another part number (ip_m),
87. For a machine operation to take place it must be enabled. A machine is enabled if the in-tray contains at least the number of parts for each of

the parts required in the machine operation, that is, as designated in the machine operation signature.

```
value
  is_enabled: MACH → Bool
  is_enabled(input,(isig,osig),op,output) ≡
     dom input = dom isig ∧
     ∀ pn:Pn • pn ∈ dom input ⇒ input(pn)≥isig(pn)
```

Annotations:

- is_enabled names a predicate. When applied to machines it checks (tests) whether
- the in-tray of the machine has exactly the kind (i.e., type) of parts needed for the machine operation: of the right part number
- and at least in the required quantity. ■

88. An enable machine can fire. Firing means that the machine performs its function: consumes an appropriate number of parts (removes them) from the in-tray and produces another appropriate number of parts (adds them) to the out-tray.

```
value
  fire: MACH → MACH
  fire(input,(isig,osig),op,output) as (input′,(isig′,osig′),op′,output′)
     pre: is_enabled(input,(isig,osig),op,output)
     post: isig′ = isig ∧ osig′ = osig ∧ op′ = op ∧
              input′ = input \ 𝒟op ∧ output′ = output ∪ ℛop
```

Annotations:

- fire is a generator function: from a machine it generates a new machine.
- The fire function is defined by a pre/post pair of predicates.
- In order to perform the machine operation the machine must be (in an) enabled (state).
- Once a machine operation has been performed (by an enabled machine) the parts required for the operation has been removed from the in-tray and the parts produced by the machine operation as been added to the out-tray. ■

C.5 Production Floors

89. A production floor of a manufacturing plant, MP, consists of a non-zero number of uniquely identified machines.

```
type
    MP, PFId, MId
    PFs′ = PFId  m→  PF, PFs = {|pfs:PFs′ • pfs≠[ ]|}
    PF′ = Mid  m→  MC, PF = {|pf:PF′ • pf≠[ ]|}
value
    obs_PFs: MP → PFs
    obs_PF: MP × PFid −~> PF, pre obs_PF(mp)(pfid): pfig ∈ dom mp
```

Annotations:

- MP is a type name. It designates the set of all manufacturing plants.
- PFId is a type name. It designates the set of all production floor identifiers.
- MId is a type name. It designates the set of all machine identifiers.
- PFs is a type name. It designates the set of all uniquely identified, non-empty production floors.
- PF is is a type name. It designates the set of all production floors — which are here modelled as non-empty sets of uniquely named machine.s
- From (or in) a manufacturing plant one can observe, obs_PFs, its set of uniquely identified production floors.
- Given a manufacturing plant and a valid production floor identifier of that plant one can observe, obs_PF, the identified plant. ▪

Observations:

- This we allow a manufacturing plant to consist of more than one production floor.
- A manufacturing plant may have two or more occurrences of what might otherwise be considered identical production floors — only they are distinguished by distinct production floor identifiers.
- A production floor may have two or more occurrences of what might otherwise be considered identical machines — only they are distinguished by distinct machine identifiers.

90. We can think of a production floor as shown in Fig. C.2.

91. Let us focus on the central — what we call — the Input/Machine/Output Machinery in that figure. Machine M potentially receives one or more parts of possibly different part numbers from machine M_{j_1}, one or more parts of possibly different part numbers from machine M_{j_2}, etc., and one or more parts of possibly different part numbers from machine M_{j_n}. And machine M potentially delivers one or more parts of possibly different part numbers to machine M_{k_1}, one or more parts of possibly different part numbers to machine M_{k_2}, etc., and one or more parts of possibly different part numbers to machine M_{k_m}. We say "potentially" since, as wee shall see later, machine M may receive or deliver parts from, respectively to other machines.

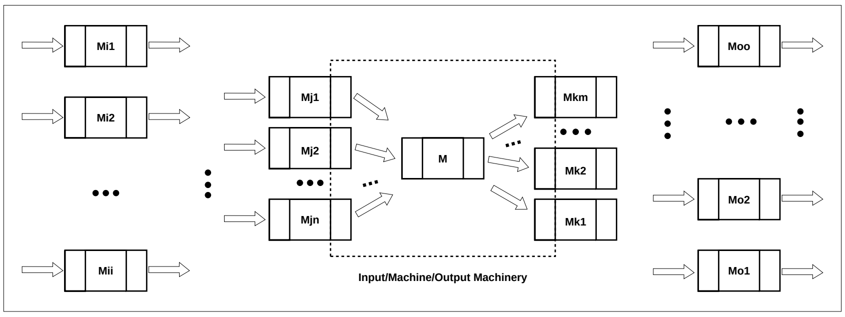


Fig. C.2. A schematic production floor

For a machine to "potentially receive" parts from another machine means two things: the parts received are necessary for machine M to perform its operation, i.e., are required inputs to M.
For a machine to "potentially deliver" parts to another machine means two things: the parts delivered are produced by machine M, i.e., are output from M.

92. Figure C.3 shows a more general situation for the input/machine/output machinery of Fig. C.2.

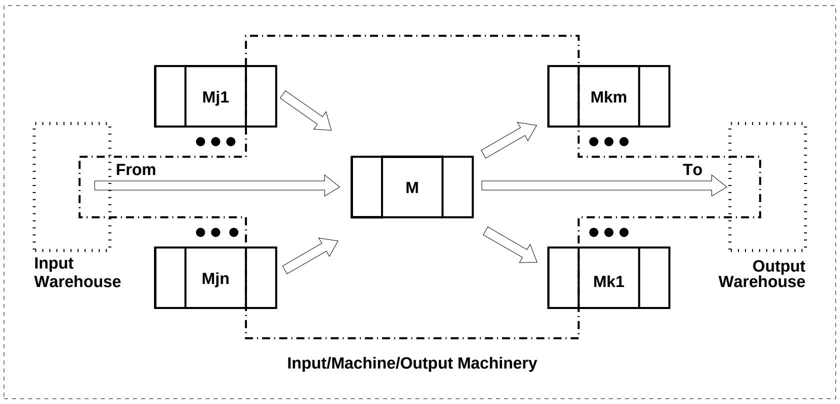


Fig. C.3. A general input/machine/output machinery

93. Whereas Fig. C.2 could be construed as expressing that all inputs to the central machine M came from other machines, Fig. C.3 hints at the possibility that some, or all, parts input to a machine operation may come from an input warehouse.

The same wrt. outputs. Instead of all central machine *M* outputs being delivered to other machines, some, or all, may go to an output warehouse.

94. It is thus that we arrive at a production unit consisting of a production floor and an input and an output warehouse.

```
type
    InWh,OutWh
value
    obs_Ps: (InWh|OutWh) → P-set
    obs_InWh: MP → InWh
    obs_OutWh: MP → OutWh
    xtr_Pns: (InWh|OutWh) → Pn-set
    xtr_Pns(wh) ≡ {pn|pn:Pn,p:P•p ∈ obs_Ps(wh)∧pn=obs_Pn(p)}
type
    BoM = Pn →m Nat
value
    xtr_BoM: (InWh|OutWh) → BoM
    xtr_BoM(wh) ≡
        [ pn↦n|pn:Pn,p:P•p ∈ obs_Ps(wh)∧pn=obs_Pn(p)∧
                n=card{p|p:P•p ∈ obs_Ps(wh)∧pn=obs_Pn(p)} ]
```

Annotations:

- InWh and OutWh are type names. They designate input, respectively output warehouses.
- From a input and output warehouses one can observe, obs_Ps, their parts.
- From a manufacturing plant one can observe, obs_InWh and obs_OutWh, their input and output warehouses.
- From a warehouse one can extract the part numbers of the parts housed in that warehouse.
- BoM is a type name. It designates the set of all Bills-of-Material. A Bills-of-Material is like a table which to distinct part numbers list a number of occurrences.
- From a warehouse one can extract a Bills-of-Material:
  - ⋆ The Bill-of-Material maps part numbers of parts in the warehouse
  - ⋆ into their number of occurrences in that warehouse.

■

Observations:

- Notice that we only observe one input and one output warehouse with a given manufacturing plant.
- That is, we consider the input and output warehouses shared between all the production floors of a manufacturing plant.
- A Bills-of-Material may list a part as having no, i.e., zero occurrences — but the Bills-of-Materials extracted from a warehouse will always have non-zero numbers of occurrence of its parts.

C.6 Production Plans

C.6.1 Production Layouts

95. By a production layout we mean any arbitrary composition (i.e., set) of machines on a production floor, pf:PF.
96. The definition of PF given in item 89 is a model of a production layout.
97. Assume, as a thought experiment, that there is such a pf:PF. It consists on n machines: m_1, m_2, ..., m_n.
98. A number of situations can now occur:
    (a) The operation of machine m_i requires parts that can all be provided by other machines m_{j_k} in the set $\{M1, m_2, \ldots, m_n\} \backslash \{m_i\}$.
    (b) Some parts necessary for the operation of machine m_i cannot be provided by any machine m_{j_k} the set $\{M1, m_2, \ldots, m_n\} \backslash \{m_i\}$. They must hence be available, i.e., provided by, the in ware house.
    (c) The operation of machine m_i produces parts that are not provided to any of the machines m_{j_k} in the set $\{M1, m_2, \ldots, m_n\} \backslash \{m_i\}$. They must hence be sent to the out ware house.
    (d) Any realistic production floor is a combination of the above (items 98(a)–98(c)).
99. We decide not to model, as part of the individual machines, from where they obtain their input production parts or to where they provide their output production parts.
100. Instead we decide to model this aspect of production in the form of a production plan.

C.6.2 Production Targets

101. By a production target we mean the number of products of a finite number of distinct part numbers
102. A production target could be expressed, as intimated in Fig. C.4, by a set of pairs of non-zero natural numbers and part numbers.
    (a) The $m : p$ pairs in out-tray boxes mean that each machine operation produces m "copies" of part p.
    (b) The $m : p$ pairs on arrows into a specific machine means that that machine, in order to perform a machine operation, consumes n copies of part p.
103. In order to fulfill a production target one or more of the machines which provide the target parts may need more than one machine operation each.
104. If such a machine, say M_i, requires n_{M_i} operations and each of these require $(m_{M_i} : p)$, $(m'_{M_i} : p')$, ..., $(m''_{M_i} : p'')$ input parts, then machine M_i requires an input production target of $(n_{M_i} \times m_{M_i} : p)$, $(n_{M_i} \times m'_{M_i} : p')$, ..., $(n_{M_i} \times m''_{M_i} : p'')$.
105. And so on.

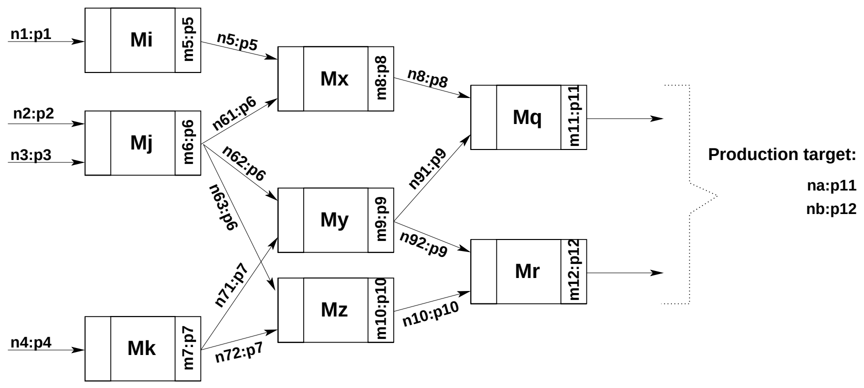


Fig. C.4. A production layout

type
  PP
  PT′ = Pn $\overrightarrow{m}$ **Nat**, PT = {|pt:PT′:∀ n:**Nat**•n ∈ **rng** pt ⇒ n≥0|}
value
  obs_PT: PP → PT

Annotations:

- PP names the sort of production plans.
- PT names the concrete type of production targets. A production target is a map from part numbers to non-zero natural numbers. ■

C.6.3 Part Dependencies

106. Figure C.4 instantiates a rather general layout. Same kind (i.e., part number) input parts may come from different machines, etc.
107. Given that p is a part of part number p_i, and given a layout as formalised by PF, one can raise the question: are there machines in that layout which produces every part required in the production of p?
108. To answer that question we first perform the following investigation. This investigation examines proper input/output part precedence relations between machines on the production floor (and the input warehouse of parts).
109. Let us examine Fig. C.3 on page 380 and Fig. C.4. Some machine input parts are (expected to be) provided by other machines on the factory floor while the remaining are (expected to be) provided by the input warehouse. Let us assume that parts p1–p4 (of Fig. C.4) are provided by the input warehouse. We must now assume that all other parts are provided by other machines from the same floor. We must further assume that there are no cycles of parts provision: That some machine m provides parts to

a subsequent machine m' which provides parts to a subsequent machine m'' which ... provides parts to a machine $m'^{\cdots'}$ which provides parts to machine m.

110. To express this formally, and thus precisely, we simplify some aspects of machines: The abstract from machine signatures pairs of sets of part numbers. (To express that there are indeed machine which can provide necessary parts, and to express non-circularity one does not need to know the quantities of parts consumed and produced.)

```
type
   MP, PFId, MId
   PFs′ = PFId $\overrightarrow{m}$ PF, PFs = {|pfs:PFs′ • pfs≠[ ]|}
   PF′ = Mid $\overrightarrow{m}$ MC, PF = {|pf:PF′ • pf≠[ ]|}
   MCSign′ = Pn-set × Pn-set
   MCSign = {|(ips,ops):MCSign′ • ips ∩ ops = {}|}
value
   obs_PFs: MP → PFs
   obs_PF: MP × PFid −∼> PF, pre obs_PF(mp)(pfid): pfig ∈ dom mp
   obs_InWhPns: MP → Pn-set
   obs_OutWhPns: MP → Pn-set
   obs_MCSign: MC → MCSign
   wf_MP: Plant → Bool
   wf_PM(mp) ≡
      let iws,ows = (obs_InWhPns(mp),obs_OutWhPns(mp)),
            pfs = obs_PFs(mp) in
      ∀ pf:PF • pf ∈ pfs ⇒
         let ... in
         ...
      end end
```

C.6.4 Production Plans

Etcetera.

C.7 Interpretations of the Model — So Far!

We present a few examples of “near”-realistic production facilities.

C.7.1 A Matchbox Factory

111. Figure C.5 intends to show the production floor of a simple minded match factory.
112. Descriptions of machines are only indicative of what “goes on”!

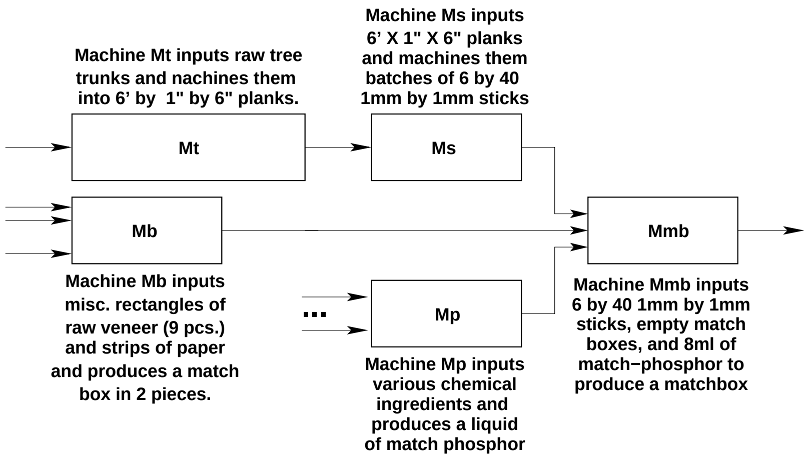

Machine Mt inputs raw tree trunks and nachines them into 6' by 1" by 6" planks.
Machine Ms inputs 6' X 1" X 6" planks and machines them batches of 6 by 40 1mm by 1mm sticks
Mt
Ms
Mb
Mmb
Mp
Machine Mb inputs misc. rectangles of raw veneer (9 pcs.) and strips of paper and produces a match box in 2 pieces.
Machine Mp inputs various chemical ingredients and produces a liquid of match phosphor
Machine Mmb inputs 6 by 40 1mm by 1mm sticks, empty match boxes, and 8ml of match−phosphor to produce a matchbox

Fig. C.5. A match factory

113. The reader may wish to make these descriptions more complete.

114. Leftmost and rightmost "dangling" arrows designate input from the in warehouse, respectively output to the out warehouse.
115. The reader may ponder about such questions as:
    (a) How is the output production capacity of machine Mt "tuned" to the input production capacity of machine Ms,
    (b) How is the output production capacity of machine Ms "tuned" to the input production capacity of machine Mmb,
    (c) How is the output production capacity of machine Mb "tuned" to the input production capacity of machine Mmb and
    (d) How is the output production capacity of machine Mp "tuned" to the input production capacity of machine Mmb.
116. So do we!
117. Please not that this is just one interpretation of the concept of machines, warehouses and production floors.
118. For more realistic pictures of match production, please see:
    (a) http://server18.joeswebhosting.net/~xx9185/english/variety/variety01.html.
    (b) http://server18.joeswebhosting.net/ xx9185/english/column/column03.html
    (c) http://phillumeny.onego.ru/collect/articles/phil36/1.html

C.7.2 A Hot Strip Mill

119. For an intuition of hot strip tubular and sheet production, please see:
    (a) http://www.ussteel.com/corp/sheet/hr/pmcpline.htm
    (b) http://www.wcisteel.com/operations/main.html

C.7.3 Cog Wheel Factory

120. For an intuition of cog wheel production, please see:
    - http://www.hero.dk/engelsk/
      Click successive images in left quadrangle

C.8 Conclusion

We can say this about the treatment given in this domain model of manufacturing: It is a rather conventional abstraction; it leaves much to be desired: many aspects are just barely sketched, some aspects not treated at all, modern machine cells are there, but are abstracted to almost not being recognisable, etcetera. We have, we think, however, achieved a main purpose of this appendix, namely: that one can and, we therefore conclude, must provide clear domain models of manufacturing, such as they are today, “out there”, with all their uncertainties, non-determinism, faults, so that we can perform more appropriate business process reengineering and design of next generation IT systems for manufacturing. Much R&D work is needed.

D

Towards a Model of **CyberRail**[1]

Summary

CyberRail is a vision of how road/rail transport may be semi-automated. It is a vision of the Japan Railway Technical Research Institute (2-8-38 Hikari-cho, Kokubunji-shi, Tokyo 185-8540, Japan) together with its railway partners in Japan. The present appendix "speculates" on the form of a possible formal model of CyberRail.

Our domain model is behaviour-oriented and uses CSP. It also reflects a "high degree" of internal non-determinism.

D.1 Background

The background for the work reported in this extended abstract is threefold: (i) Many years of actual formal specification as well as research into how to engineer such formal specifications, by the first author, of domains, including the railway domain [47] [52] [60] [61] [18] [53] [21] [20] [50] — using abstraction and modelling principles and techniques extensively covered in three forthcoming software engineering textbooks [31–33]. (ii) A term project with four MSc and one PhD students[2]. And (iii) Some fascination as whether one could formalise an essence of the novel ideas of CyberRail. We strongly believe that we can capture one crucial essence of CyberRail[3] — such as this paper will show.

The formalisation of CyberRail is expressed in the RAISE [106] Specification Language, RSL [104]. RAISE stands for Rigorous Approach to Industrial Software Engineering. In the current abstract model we especially make use of

[1]This is an edited version of [24].

[2]Peter Chiang, Morten S. T. Jacobsen, Jens Kielsgaard Hansen, Michael P. Madsen and Martin Pěnička

[3]See Sect. D.3 (Page 393) for references to literature on CyberRail.

RSL's parallel process modeling capability. It builds on, ie., borrows from Tony Hoare's algebraic process concept of Communicating Sequential Processes, CSP [137].

D.2 A Rough Sketch Formal Model

D.2.1 An Overall CyberRail System

CyberRail consists of an index set of traveller behaviours and one cyber behaviour "running" in parallel. Each traveller behaviour is uniquely identified, p:Tx. Traveller behaviours communicate with the cyber behaviour. We abstract the communication medium as an indexed set of channels, ct[p], from the cyber behaviour to each individual traveller behaviour, and tc[p], from traveller behaviours to the cyber behaviour. Messages over channels are of respective types, CT and TC. The cyber behaviour starts in an initial state ω_i, and each traveller behaviour, p, starts in some initial state $m\sigma_i(p)$.

```
type
    Tx, $\Sigma$, $\Omega$, CT, TC
    M$\Sigma$ = Tx $\overrightarrow{m}$ $\Sigma$
channel
    {ct[p]:CT,tc[p]:TC|p:Tx}, cr:CR, rc:RC
value
    m$\sigma_i$:M$\Sigma$, $\omega_i$:$\Omega$

    cyberrail_system: Unit $\rightarrow$ Unit
    cyberrail_system() $\equiv$
        $\|$ { traveller(p)(m$\sigma_i$(p)) | p:Tx } $\|$ cyber($\omega$)

    cyber: $\Omega$ $\rightarrow$ in {tc[p]|p:Tx},cr out {ct[p]|p:Tx},rc Unit
    cyber($\omega$) $\equiv$
        cyber_as_server($\omega$) $\sqcap$
        cyber_as_proactive($\omega$) $\sqcap$
        cyber_as_co_director($\omega$)

    traveller: p:Tx $\rightarrow$ $\Sigma$ $\rightarrow$ in ct[p] out tc[p]  Unit
    traveller(p)($\sigma$) $\equiv$
        active_traveller(p)($\sigma$) $\sqcap$
        passive_traveller(p)($\sigma$)
```

The cyber behaviour either acts as a server: Ready to engage in communication input from any traveller behaviour; or the cyber behaviour acts pro–actively: Ready to engage in performing output to one, or some traveller behaviours; or the cyber behaviour acts in consort with the "rest" of the transportation market (including rail infrastructure owners, train operators, etc.), in improving and changing services, and in otherwise responding to unforeseen circumstances of that market.

Similarly any traveller behaviour acts as a client: Ready to engage in performing output to the cyber behaviour; or its acts passively: Ready to accept input from the cyber behaviour.

D.2.2 Travellers

Active Travellers

Active traveller behaviours alternate internally non–deterministically, ie., at their own choice, between *start (travel) planning st_pl, select (among suggested) travel plan(s) se_pl, change (travel) planning ch_pl, begin travel be_tr, board train bo_tr, leave train lv_tr, ignore train ig_tr, cancel travel ca_tr, seeking guidance se_gu, notifying cyber no_cy, entertainment ent, deposit resource de_re* (park car, . . .), *claim resource cl_re* (retreive car, . . .), *get resource ge_re* (rent a car, . . .), *return resource re_re* (return rent-car, . . .), *going to restaurant rest* (or other), *change travel ch_tr, interrupt travel in_tr, resume travel re_tr, leave train le_tr, end travel en_tr,* and many other choices. Each of these normally entail an output communication to the cyber behaviour, and for those we can assume immediate response from the cyber behaviour, where applicable.

```
value
  active_traveller: p:Tx → Σ → out tc[p] in ct[p] Unit
  active_traveller(p)(σ) ≡
    let choice = st_pl ⌈⌉ ac_pl ⌈⌉ ch_pl ⌈⌉ en_tr ⌈⌉ ... ⌈⌉ le_tr ⌈⌉ te_tr in
    let σ′ = case choice of
                st_pl → start_planning(p)(σ),
                se_pl → select_travel_plan(p)(σ),
                ch_pl → change_trael_plan(p)(σ),
                be_tr → begin_travel(p)(σ),
                bo_tr → board_train(p )(σ),
                ... → ..,
                le_tr → leave_train(p)(σ),
                en_tr → end_travel(p)(σ),
                ... → ..
             end in
    traveller(p)(σ′) end end

  start_planning: p:Tx → Σ → out tc[p] in ct[p] Σ
  start_planning(p)(σ) ≡
    let (σ′,plan) = magic_plan(σ) in
    tc[p]!plan;
    let sps = ct[p]? in updateΣ((plan,sps))(σ′) end end
    ...
    update$\Sigma$: Update → $\Sigma$ → $\Sigma$
type
    Update == mkInPlRes(ip:InitialPlan,ps:Plan-set) | ...
```

Passive Travellers

When not engaging actively with the cyber behaviour, traveller behaviours are ready to accept any cyber initated action. The traveller behaviour basically "assimilates" messages received from cyber — and may make use of these in future.

```
value
    passive_traveller: p:Tx → $\Sigma$ → in ct[p] out tc[p] Unit
    passive_traveller(p)($\sigma$) ≡ let res = ct[p]? in update$\Sigma$(res)($\sigma$) end
```

Active Traveller Actions

The *active_traveller* behaviour performs either of the internally non–deterministically chosen actions: *start_planning*, *select_travel_plan*, *change_travel_plan*, *begin_travel*, *board_train*, ..., *leave_train*, or *end_travel*. They make use only of the "sum total state" (σ) that that traveller behaviour "is in". Each such action basically communicates either of a number of plans (or parts thereof, here simplified into plans). Let us summarise:

```
type
    Plan
    Request = Initial_Plan | Selected_Plan | Change_Plan | Begin_Travel
              | Board_Train | ... | Leave_Train | End_Travel | ...
    Initial_Plan == mkIniPl(pl:Plan)
    Selected_Plan == mkSelPl(pl:Plan)
    Change_Plan == mkChgPl(pl:Plan)
    Begin_Travel == mkBTrav(pl:Plan)
    Board_Train == mkBTrai(pl:Plan)
    ...
    Leave_Train == mkLeTr(pl:Plan)
    End_Travel == mkEnTr(pl:Plan)
value
    ∀ f: p:Tx → $\Sigma$ → out tc[p] $\Sigma$
    magic_f: $\Sigma$ → $\Sigma$ × Request

    f(p)($\sigma$) ≡ let ($\sigma'$,req) = magic_f($\sigma$) in tc[p]!req;$\sigma'$ end
```

The magic_functions access and changes the state while otherwise yielding some request. They engage in no events with other than the traveller state. There are the possibility of literally “zillions” such functions, all fitted into the above sketched traveller behaviour.

D.2.3 cyber

cyber as Server

cyber is at any moment ready to engage in actions with any traveller behaviour. cyber is assumed here to respond immediately to “any and such”.

```
value
    cyber_rail_as_server: $\Omega \rightarrow$ in {tc[p]|p:Tx} out {ct[p]|p:Tx} Unit
    cyber_rail_as_server($\omega$) $\equiv$
        $\lceil\rceil$ {let req = tc[p]? in cyber(serve_traveller(p,req)($\omega$)) end | p:Tx}

    serve_traveller: p:Tx $\times$ Req $\rightarrow \Omega \rightarrow$ in {tc[p]|p:Tx} out {ct[p]|p:Tx} $\Omega$
    serve_traveller(p,req)($\omega$) $\equiv$
        case req of
            mkIniPl(pl) $\rightarrow$
                let ($\omega'$,pls) = sugg_pls(p,pl)($\omega$) in ct[p]!pls;cyberrail($\omega'$) end
            mkSelPl(pl) $\rightarrow$
                let ($\omega'$,res) = res_pl(p,pl)($\omega$) in ct[p]!book;cyberrail($\omega'$) end
            mkChgPl(pl) $\rightarrow$
                let ($\omega'$,pl$'$) = chg_pl(p,pl)($\omega$) in ct[p]!pl$'$;cyberrail($\omega'$) end
            mkBTrav(pl) $\rightarrow$ ...
            mkBTrai(pl) $\rightarrow$ ...
            ...
            mkLeTr(pl) $\rightarrow$ ...
            mkEnTr(pl) $\rightarrow$ ...
        end
```

cyber as Pro–Active

cyber, on its own volition, may, typically based on its accumulated knowledge of traveller behaviours, engage in sending messages of one kind or another to selected groups of travellers. Section D.2.3 rough sketch–formalises one of these.

```
type
    CR_act == gu_tr | no_tr | co_tr | wa_tr | ...
value
    cyber_as_proactive: $\Omega \rightarrow$ out {ct[p]|p:Tx} Unit
    cyber_as_proactive($\omega$) $\equiv$
    let cho = gu_tr ⌈⌉ no_tr ⌈⌉ co_tr ⌈⌉ wa_tr ⌈⌉ ... in
    let ω′ = case cho of gu_tr → guide_traveller(ω),
                         no_tr → notify_traveller(ω),
                         co_tr → commercial_to_travellers(ω),
                         wa_tr → warn_travellers(ω),
                         ... → ... end in
    cyber(ω′) end end
```

cyber as Co–Director

We do not specify this behaviour. It concerns the actions that cyber takes together with the "rest" of the transportation market. One could mention input from cyber_as_co_director to the train operators as to new traveller preferences, profiles, etc., and output from the rail (ie., net) infrastructure owners or train operators to cyber_as_co_director as to net repairs or train shortages, etc. The decomposition of CyberRail into cyber and the "rest", may — to some — be articificial, namely in countries where there is no effective privatisation and split–up into infrastructyre owners and train operators. But it is a decomposition which is relevant, structurally, in any case.

cyber Server Actions

We sketch:

```
value
  sugg_plans: p:Tx × Plan → Ω → Ω × Plan-set
  res_pl: p:Tx × Plan → Ω → Ω × Plan
  chg_pl: p:Tx × Plan → Ω → Ω × Plan
  ...
```

There are many other such traveller instigated cyber actions.

Pro–Active cyber Actions

We rough sketch just a single of the possible "dozens" of cyber inititated actions versus the travellers.

```
value
  guide_traveller: Ω → out {ct[p]|p:Tx} Ω
  guide_traveller(ω) ≡
    let (ω′,(ps,guide)) = any_guide(ω) in broadcast(ps,guide) ; ω′ end

  any_guide: Ω → Ω × (Tx-set × Guide)
```

```
    notify_traveller: $\Omega$ → **out** {ct[p]|p:Tx} $\Omega$
    commercial_to_travellers: $\Omega$ → **out** {ct[p]|p:Tx} $\Omega$
    warn_traveller: $\Omega$ → **out** {ct[p]|p:Tx} $\Omega$
    ...

    broadcast: Tx-**set** × CT → **Unit**
    broadcast(ps,msg) ≡
       **case** ps **of** {}→**skip**,{p}∪ ps′→ct[p]!msg;broadcast(ps′,msg) **end**

**type**
    CT = Guide | Notification | Commercial | Warning | ...
    Guide == mkGui(...)
    Notification == mkNot(...)
    Commercial == mkCom(...)
    Warning == mkWar(...)
    ...
```

D.3 A CyberRail Bibliography

1. Takahiko Ogino: "Advanced Railway Transport Systems and ITS", RTRI Report Vol 13, No. 1, January 1999 (in Japanese)
2. Takahiko Ogino, Ryuji Tsuchiya: "CyberRail: A Probable Form of ITS in Japan", RTRI Report Vol 14, No. 7, July 2000 (in Japanese)
3. Takahiko Ogino: "CyberRail: An Enhanced Railway System for Intermodal Transportation", Quarterly Report of RTRI, Vol 42, No. 4, November 2001
4. Takahiko Ogino: "When Train Stations become cyber Stations", Japanese Railway Technology Today, pp 209-219, December 2001
5. Takahiko Ogino: "CyberRail Study Group Activities and Achievements", RTRI Report Vol 16, No.11, November 2002 (in Japanese)
6. Takahiko Ogino, Ryuji Tsuchiya, Akihiko Matsuoka, Koichi Goto: "A Realization of Information and Guidance function of cyber", RTRI Report Vol 17, No. 12, December 2003 (in Japanese)
7. Takahiko Ogino:"CyberRail - In search of IT infrastracture in intermodal transport", JREA, Vol.45., No.1 (2002) (in Japanese)
8. Ryuji Tsuchiya, Koichi Goto, Akihiko Matsuoka, Takahiko Ogino, "CyberRail and its significance in the coming ubiquitous society", Proc. of the World Congress on Railway Research 2003 (2003-9) (in Japanese)
9. Takashi Watanabe, :"Experiment ofCyberRail Passenger guidance using Bluetooth", Preprint of RTRI Annual Lecture Meeting in 2000 (in Japanese)
10. Takashi Watanabe, et. al.: "Personal Navigation System Using Bluetooth", Technical Report of ITS-SIG,IPSJ(2001-ITS-4), p.55 (in Japanese)
11. Ryuji Tsuchiya, Koichi Goto, Akihiko Matsuoka, Takahiko Ogino, "Deriving interoperable traveler support system specification through requirements engineering process", Proc. of the 7th World Multiconference on Systemics, Cybernetics and Informatics (July, 2003)

12. Ryuji Tsuchiya, Takahiko Ogino, Koichi Goto, Akihiko Matsuoka, "Personalized Passenger Information Services and cyber", Technical Report of SIG-IAC, IPSJ (2002-IAC-4), p15 (in Japanese)
13. Akihiko Matsuoka, Ryuji Tsuchiya: "Current Status of cyber SIG",Technical Report of SIG-ITS, IPSJ, p.45 (2002-ITS-11) (in Japanese)
14. Akihiko Matsuoka, Koichi Goto, Ryuji Tsuchiya, Takahiko Ogino: "CyberRail and new passengers information services", IEE Japan, TER-03-22 (2003-6) (in Japanese)
15. Yuji Shinoe, Ryuji Tsuchiya, "Personalized Route Choice Support System for Railway Passengers", Technical Report of SIG-ITS, IPSJ (2001-ITS-6), p.23 (in Japanese)
16. Hiroshi Matsubara, Noriko Fukasawa, Koichi Goto, "Development of Interactive Guidance System for Visually Disabled", Technical Report of SIG-ITS, IPSJ (2001-ITS-6), p.75 (in Japanese)
17. Ryuji Tsuchiya, Takahiko Ogino, Koichi Goto, Akihiko Matsuoka, "Location-sensitive Itinerary-based Passenger Information System", Technical Report of ITS-SIG, IPSJ, p.85 (2003-ITS-6) (in Japanese)
18. Ryuji Tsuchiya, Kiyotaka Seki, Takahiko Ogino, Yasuo Sato: "User services ofCyber-Rail - toward system architecture of future railway-", Proc. of the World Congress on Railway Research 2001 (2001-11)
19. Takahiko Ogino, Ryuji Tsuchiya, Kiyotaka Seki, Yasuo Sato: "CyberRail - information infrastructure for intermodal passengers-", Proc. of the World Congress on Railway Research 2001 (2001-11)
20. Kiyotaka Seki, Ryuji Tsuchiya, Takahiko Ogino, Yasuo Sato: "Construction of future railway system utilizing information and telecommunication technologies", Proc. of the World Congress on Railway Research 2001 (2001-11)
21. Ryuji Tsuchiya, Akihiko Matsuoka, Takahiko Ogino, Kouich Goto, Toshiro Nakao, Hajime Takebayashi: "Experimental system for CyberRail passenger information providing and guidance", 40th Railway-Cybernetics Symposium (2003-11) (in Japanese)
22. Akihiko Matsuoka, Ryuji Tsuchiya, Takahiko Ogino, Toshio Hirota: "CyberRail System Architecture", 40th Railway-Cybernetics Symposium (2003-11) (in Japanese)
23. Ryuji Tsuchiya, Akihiko Matsuoka, Takahiko Ogino, Kouich Goto, Toshiro Nakao, Hajime Takebayashi: "Location-sensitive Itinerary-based Passenger Information System", Applying to IEE Journal (in Japanese)

D.4 Conclusion

A formalisation of a crucial aspect of CyberRail has been sketched. Namely the interplay between the rôles of travellers and the central CyberRail system.

Next we need analyse carfully all the action functions with respect to the way in which they use and update the respective states ($\sigma : \Sigma$) of traveller behaviours and the cyber behaviour ($\omega : \Omega$). At the end of such an analysis one can then come up with precise, formal descriptions, including axioms, of a CyberRail *information infrastructure*. We look forward to report on that in a near future.

The objective of this work is to provide a foundation, a domain theory, for CyberRail. A set of models from which to "derive", in a systematic way, proposals for computing systems, including software architectures.

We have sketched a model of CyberRail such as we have understood in from several of the English language sources referenced in the previous section. Our aim, when we first made the model, was to show our Japanese colleagues that abstraction can be useful in understanding new ideas, in discovering further new transportation concepts, etc. Our objectives are unchanged and derived from the aims: to actually see abstractions and formalisations being applied in industries such as the railway industry.

E

Towards a Domain Model of ‘The Market’[1]

By a domain we understand an area of human (or other) activity. Examples are: “the railway domain”, “the health–care domain”, the domain of the “financial service industry”, etc. Elsewhere the composite term ‘application domain’, where ‘application’ signals that the person who utters the composite term intends to apply computers & communication to problems of the domain.

We present our understanding of a domain through documents. Software development is focused on the development of (semantically meaningful) documents.

We present a fair selection of parts of descriptive documents.

E.1 A Rough Sketch and its Analysis

We first bring an example rough sketch, then its analysis. After that we bring both rough sketches and analyses.

E.1.1 Buyers and Sellers

First a rough sketch of what is meant by buyers and sellers, then its analysis.

Rough Sketch

Consumers, retailers, wholesalers and producers form the major “players” in the market.

A consumer may inquire with a supposedly appropriate retailer as to the availability of certain products (cum merchandise): Their price, delivery times, other delivery conditions (incl. quantity rebates), and financing (ie., payment). A retailer may respond to a consumer inquiry with either of the following

[1]This is an extensive pre-version of [19].

responses: A quote of the requested information, or a (courteous) declination, or a message that the inquiry was misdirected (refusals), or the retailer may decide to not, or fail to, respond ! A consumer may decide to order products with a supposedly appropriate retailer, whether such an order has been or has not been preceded by a related inquiry. The retailer may respond to a consumer order with either of the following responses: Confirming, declining or "no–response", with a confirmation being following either by a delivery, or no delivery — or the retailer may just provide a delivery, or inform the consumer that a back–order has been recorded: The desired products may not be in store, but has been (or will be) ordered from a wholesaler — for subsequent delivery. A delivery may deliver the ordered or some other, not ordered, products ! The consumer may decide to not accept, or to accept a delivery. The retailer may invoice the consumer before, at the same time as, or after delivery. The consumer may pay, or not pay an invoice, including performing a payment based on no invoice, for example at the same time as placing the order. The retailer may acknowledge payments. The consumer may find faults with a previously accepted delivery and return that (or, by mistake, another) delivery. The retailer may refund, or not refund such a return.

Analysis

Based on an analysis of the above rough sketch we suggest to treat market interactions between retailers and wholesalers, and between wholesalers and producers in exactly the same way as interactions between consumers and retailers. That is: we observe that retailers acts as (a kind of) "consumers" vis-a–vis wholesalers (who, similarly acts as retailers).

We thus summarise the interactions into the following enumeration: inquiries, quotes, declinations, refusals, orders, confirmations, deliveries, acceptances, invoicings, payments, acknowledgments, returns, and refunds.

Figure E.1 on the next page attempts to illustrate possible transaction transitions between buyers and sellers.

E.1.2 Traders

As a consequence of the analysis we shall "lift" the labels 'consumer', 'retailer', 'wholesaler' and 'producer' into the labels 'buyer' and 'seller'. And we shall use the term 'trader' to cover both a buyer and a seller. Since the consumers and producers mentioned in the rough sketch above may also act as any of the other kinds of traders, all will be labeled traders.

Figure E.2 on the facing page attempts to show that a trader can be both a buyer and a seller. Thus traders "alternate" between buying and selling, that is: Between performing 'buy' and performing 'sell' transactions.

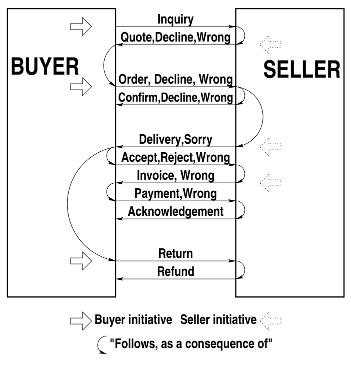


Fig. E.1. Buyer / Seller Protocol

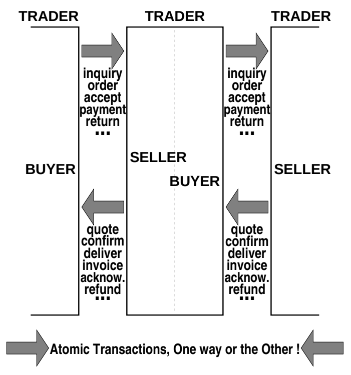


Fig. E.2. Trader=Buyer+Seller

E.1.3 Supply Chains

Figure E.3 on the next page attempts to show "an arbitrary" constellation of buyer and seller traders. It highlights three supply chains. Each chain, in this example, consists, in this example, of a "consumer", a retailer, a wholesaler, and a producer.
A collection, a set, of traders may thus give rise to any set of supply chains, with each supply chain consisting of a sequence of two or more traders. Supply chains are not static: They form, act and dissolve. They are a result of positive inquiries, orders, deliveries, etc.

'Likeness', 'Kinds', 'Adjacency', and 'Supply Chain Instances'

As a result of analysis we identify a need for some abstract concepts: 'likeness', 'kinds', and 'supply [chain] instances' (where [...] expresses that we can omit the ...).

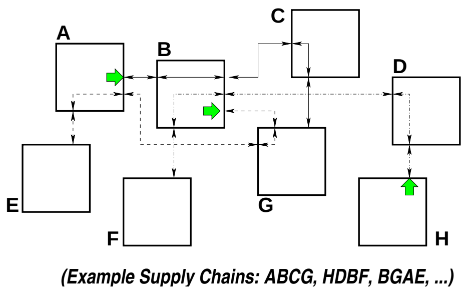


Fig. E.3. A Network of Traders and Supply Chains

Like traders are of the same 'kind', where the 'kind' of a trader is either consumer, retailer, wholesaler, or producer.

We can also speak of the 'kind' of a transaction.

The 'kind' of a transaction is either than of inquiry, quote, declination, refusal, order, confirmation, delivery, acceptance, invoice, payment, acknowledgment, return, or refund.

There may be chains of one or more wholesalers: Global, regional, national, or, within a state, area wholesalers. We therefore allow for the following kinds of adjacent traders: (consumer,retailer), (retailer,wholesaler), (wholesaler,wholesaler), and (wholesaler,producer).

A supply [chain] instance is a specific and related occurrence of two or more transactions. The following is an elaborate supply chain instances — where we omit reference to the specifics by only mentioning the transaction kinds: (i) *inquiry* (consumer to retailer), → *inquiry* (retailer to wholesaler), → *quote* (wholesaler to retailer), → *quote* (retailer to consumer), → *order* (consumer to retailer), → *order* (retailer to wholesaler), → *order* (wholesaler to producer), → *confirm* (producer to wholesaler), → *confirm* (wholesaler to retailer), → *confirm* (retailer to consumer), → *delivery* (producer to wholesaler), → *acceptance* (wholesaler to producer), → *delivery* (wholesaler to retailer), → *acceptance* (retailer to wholesaler), → *delivery* (retailer to consumer), → *acceptance* (consumer to retailer), → *invoice* (retailer to consumer), → *payment* (etc., the reader fills in possible details), → *acknowledge*, → *invoice*, → *invoice*, → *payment*, → *payment*, → *acknowledge*, → *acknowledge*, → *return*, and → *refund*.

E.1.4 Agents and Brokers

Although not formalised explicitly in the present paper we discuss the concepts of brokers and traders. We then, later on, "reduce" agents and brokers to become like traders are.

Agents

An agent, α, in the domain, is any human or any enterprise, including media advertisement, who, or which, acts on behalf of one trader, t_1, in order to mediate possible purchase (or sale) of goods from another trader, t_2. So t_1 may be a consumer, or a retailer, or a wholesaler who, through α acquires goods from t_2 who, respectively, is a retailer, a wholesaler and a producer. Or t_1 may be a retailer, or a wholesaler, or a producer who, through α sells to t_2 who, respectively, is a consumer, a retailer, and a wholesaler. One can generalise the notion of agents to such who (or which) acts on behalf of a group of like traders to "reach" a corresponding group of like and adjacent traders.

Figure E.4 attempts to show a buyer–agent (left hand figure), respectively a seller–agent (right hand figure). The buyer–agent "searches" the market for suitable sellers of a specific product. The seller–agent searches the market for suitable buyers of a specific product.

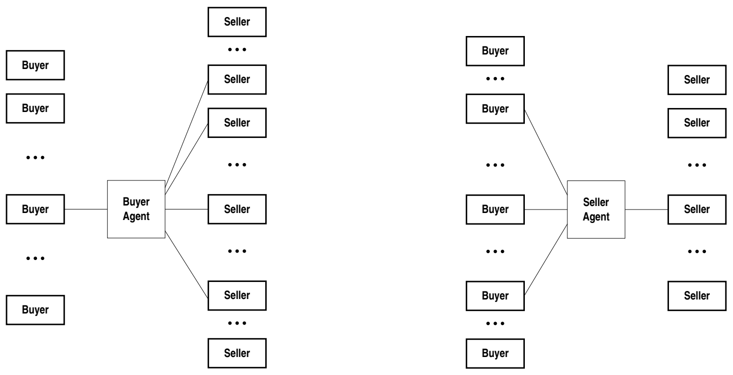


Fig. E.4. Buyer and Seller Agents

The idea is that the two kinds of agents behave like buyers, respectively like sellers: The buyer–agent "learns" from the buyer about what is to be inquired, is instructed when to order, etc. (This is designated by the single line (between the Buyer and the Buyer Agent rectangles) of the left–hand side of Figure E.4.) The buyer–agent then iterates over a set of sellers known to meet inquired expectation. (This is designated by the mostly slanted lines (between the Buyer Agent and the Seller Agent rectangles) of the left–hand side of Figure E.4.)

Similarly for seller–agents (the right–hand side of Figure E.4).

Brokers

A broker, β, in the domain, is any human or any enterprise, including media advertisement, who, or which, acts on behalf of two (or more, respectively) adjacent groups of like traders, bringing them together in order to effect instances of supplies.

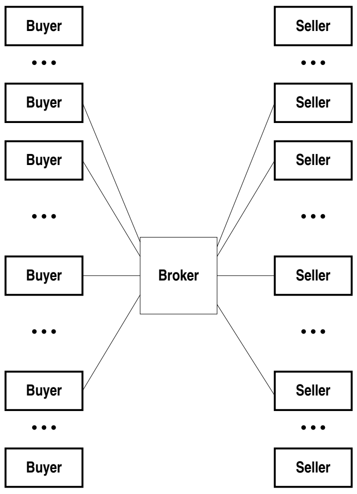


Fig. E.5. A Simple ("One Stage") Broker

Figure E.5 attempts to diagram a broker mediating between m buyers and n (adjacent kind) sellers.

The idea is that a combination of buyer and seller searches, and hence a combination of the buyer– and seller–agent behaviours are needed.

Brokers can span more than one stage.

Figure E.6 on the facing page attempts to diagram a broker mediating between m_1 consumers, m_2 retailers, m_3 wholesalers and m_4 producers — subsets of all the known such.

The three sets of dashed lines in the three vertical "stems" of the broker shall designate "local" brokerage between adjacent pairs of buyers and sellers. The set of dashed lines in the horisontal branch of the broker shall designate overall, "global" brokerage between all parties.

The aim of the mediation is to create a consortium of subsets of consumers, retailers, wholesalers and producers. The objective of the consortium is, like a *"Book of the Month Club"*, to create a stable set of complete supply chains for a given set of products.

As for simple brokers we shall (ever so briefly) argue that the same iterated searching of resolution protocols and mechanisms as for agents are to be

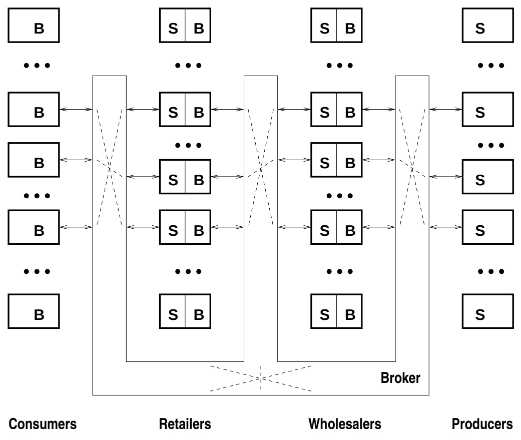


Fig. E.6. A Multiple (here: Three) Stage Broker

deployed, and that these are based on the all the transaction kinds as first sketched.

E.1.5 Catalogues

An important concept of the market is that of a catalogue. It may be implicit, or it may exist explicitly. A catalogue, in a widest sense of that term, is any form of recording that lists what merchandise is for sale, its price, conditions of delivery, payment, refund, etc. An ordinary retailer — your small neighborhood *"Mom & Pop"* store — may not be able to display a catalogue in the form of, for example, a ring binder each of whose pages lists, in some order, the merchandise by name, order number, producer, etc., and which records the above mentioned forms of information. But, from the shelves of that store one can "gather" that information. For wholesalers and producers we can probably assume such more formal catalogues. But, as a concept, we can in any case speak of catalogues. And hence we can speak of such concepts as *searching* in a catalogue, marking entries as being *out of stock, how many sold, when, to whom* etc.

E.1.6 The Transactions

We have, above, just hinted at the kind of transactions, to wit: inquiry, quote, declination, refusal, order, confirm, delivery, acceptance, invoice, payment, acknowledge, return, and refund. Instead of treating them in more detail — as part of a narrative — we relegate, for the sake of brevity, such a treatment to the terminology section, next, and to the formalisation, following.

E.1.7 Contractual Relations

Issuance of orders, order confirmations, acceptance of deliveries, issuance of invoices and attemots of payments, etc., imply a number of contractual relations. Again notions of ‘parties to the contract’, ‘subject matter’, and ‘considerations’ arise. For the first two is seems reasonably as to what is meant. With respect to considerations we briefly mention such things as *conditions of delivery, conditions of acceptance (testing),* and *whether credit worthyness, specific forms of payments,* and *credit period* have been *established,* are being *fullfilled,* and the *extension* or *termination* of *credit lines.*

We shall not go into whether new kinds a transactions are needed to deal with contractual considerations — other than suggesting that the ones already implied (inquiry, quotation, reject, order, conform, delivery, acceptance, invoicing, payments, acknowledment, return and refund) — used, in a sense, at a meta–level — already suffice ! But to justify this, perhaps cryotic remark, requires a proper demonstration — which will not be given in the current paper.

• • •

This completes our, lengthy, rough sketch of “The Market” domain. It was made deliberately long in order to make the point: That rough sketching is an important process, and that rough sketches serve a purpose — as we shall subsequently see.

E.2 Narrative and Formal Model

We combine, into one document, the informal description and the formal description of the domain of traders. We describe only the basic protocols for inquiry, quote, order, confirmation, delivery, acceptance, invoice, payment, etc. transactions. We thus do not describe agents and brokers. We leave that to a requirements modeling phase.

Please observe the extensive need for expressing selection of and responses to transactions non–deterministically. In the real world, ie., in the domain, all is possible: Diligent staff will indeed follow–up on inquiries, orders, payments, etc. Loyal consumers will indeed respond likewise. But sloppy such people may not. And outright criminals may wish to cheat, say on payments or rejects. And we shall model them all. Hence non–determinism.

E.2.1 Formalisation of Syntax

```
type
  Trans == Inq|Ord|Acc|Pay|Ret
         | Qou|Con|Del|Acc|Inv|Ref
         | NoR|Dec|Mis
```

The first two lines above list the 'buyer', respectively the 'seller' initiated transaction types. The third line lists common transaction types.

In the domain we can speak of the uniqueness of a transaction: *"it was issued at such–and–such time, by such–and–such person, and at such–and–such location,"* etcetera.

U below stand for (supposedly, or possibly) unique identifications, including time, location, person, etc., stamps (T, P, L), Sui (where i=1,2) stands for surrogate information, and MQP alludes to Merchandise identification, Quantity, and Price.

```
type
    U, M, Q, P, T, Su1, Su2, Inf
    Inq :: MQP × U
    MQP == mk(m:M,q:Q,p:P,...)
    Quo :: ((Inq|Su1) × Inf) × U
    Ord :: Qou|Su2 × U
    Con :: Ord × U
    Del :: Ord × U
    Acc :: Del × U
    Inv :: Ord × U
    Pay :: Inv × U
    Ret :: Del × U
    Ref :: Pay × U
    NoR :: Trans × U
    Dec :: Trans × U
    Mis :: Trans × U
value
    obs_T: U → T
```

The above defines the syntax of classes of disjoint transation commands, of the abstract form mk_Name(kind,u) where Name is either of Inq, Quo, Ord, Con, Del, ... or Mis.

An inquiry:Inq consists of a pair, some (desired) merchandise, (desired) quantity and (desired) price information, and a supposedly unique identification (of time, location, person, etc.) of issue – this "mimics" a consumer inquiry of the form *"I am in the market for such–and–such merchandise, in such–and–such a quantity, and at such–and–such prices. What can you offer ?"*..

An quote:Quo either refers to the inquiry in which the quote is a response or presents surrogate information — typically (where the seller takes the initiative to advertise some merchandise and then) of a form similar to an inquiry: *"If you are in the market for such–and–such merchandise, in such–and–such a quantity, and at such–and–such prices, then here is what we offer".*

information:Inf is then what is offered.

In general we model, in the *domain*, a "subsequent" transaction by referring to a complete trace of (supposedly) unique time, location, person, etc., stamped transactions. Thus, in general, a transaction "embodies" the transaction it is a manifest response to, and time, location, person, etc. of response.

Do not mistake this for a requirement. A requirement may or may not impose unique identification wrt. time and location and person etc. Therefore we do not detail U. Nor do we actually say that no two transactions can be issued with the same uniqueness.

E.2.2 Formalisation of Semantics of Market Interactions

"The Market" consist of n traders, whether buyers, or sellers, or both; whether additionally agents or brokers. Each trader τ_i is able, potentially to communicate with any other trader:

$$\{\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n\}.$$

We omit formal treatment of how traders come to know of one another. An arbiter for such information is just like a trader. Other traders sell information about their existence to such an arbiter. Thus no special formal treatment is necessary.

We focus on the internal and external non–determinism which is always there, in the *domain*, when transactions are selected, sent and received.

Our model is expressed in a variant of `CSP`, as "embedded" in `RSL` [104].

```
type
[0]  $\Theta$, MSG
[1]  Idx = {| 1..n |}

value
[2]  sys: (Idx $\overrightarrow{m}$ $\Theta$) → Unit
[3]  sys(m$\theta$) ≡ ‖ { tra(i)(m$\theta$(i)) | i:Idx }

channels {tc[i,j]:MSG | i,j:Idx • i< j}

value
[4]  tra: i:Idx → $\Theta$ → in {tc[j,i]|j:Idx•i≠j} out {tc[i,j]|j:Idx•i≠j} Unit
[5]  tra(i)($\theta$) ≡ tra(i)(nxt(i)($\theta$))

[6]  nxt: i:Idx → $\Theta$ → in {tc[j,i]|j:Idx•i≠j} out {tc[i,j]|j:Idx•i≠j} $\Theta$
[7]  nxt(i)($\theta$) ≡
[8]    let choice = rcv ⌈⌉ snd in
[9]    cases choice of rcv→receive(i)($\theta$), snd→send(i)($\theta$) end end
```

(0) Θ is the state space that any trader may span. MSG is type space of all messages that can be exchanged between traders (ie., over channels). We detail neither Θ nor MSG: In the "real world", ie., in the domain, all is possible. Determination of Θ and MSG is usually done when "deriving" the functional requirements from the domain model. (1) Idx is the set of n indexes, where each trader has a unique index. We do not detail Idx. That usually is done as late as possible, say during code implementation. (2) The system initialises each trader with a possibly unique local state (from its only argument). (3) The system is the parallel combination of n traders. (4) A trader has a unique, constant index, i, and is, at any moment, in some state θ. (4) Traders communicate (both **in**put and **out**put) over channels: tc[i,j] — from trader i to trader j. (5)

Each trader is modeled as a process which “goes on forever”, (5) but in steps of next state transitions. (8) The next state transition non—deterministically (internal choice, ⌈⌉) “alternates” between (9) expressing willingness to receive, respectively desire to send.

In “real life”, ie. in the domain, the choice as to which transactions are pursued is non–deterministic. And it is an internal choice. That is: The choice is not influenced by the environment.

We model receiving as something “passive”: No immediate response is made, but a receive state component of the trader state is updated. A trader that has decided to send (something), may non–deterministically decide to inspect the receive component of its state so as to ascertain whether there are received transactions pending that ought or may be responded to.

The update_rcv_state invokes further functions.

```
receive: i:Idx → $\Theta$ → in {tc[j,i]|j:Idx•i≠j} $\Theta$
receive(i)($\theta$) ≡
    ⌈⌉ {let msg=tc[j,i]? in update_rcv_state(msg,j)($\theta$) end | j:Idx}
```

Once the internal non–deterministic choice (⌈⌉) has been made ((8) above): Whether to receive or send, the choice as to whom to ‘receive from’ is also non–deterministic, but now external ([]). That is: receive expresses willingness to receive from any other trader. But just one. As long as no other trader j does not send anything to trader i that trader i just “sits” there, “waiting” — potentially forever. This is indeed a model of the real world, the domain. A subsequent requirement may therefore, naturally, be to provide some form of time out. A re–specification of receive with time out is a correct implementation of the above.

```
[2]  update_rcv_state: MSG × i:Idx → $\Theta$ → $\Theta$
[3]  update_rcv_state(msg,j)($\theta$) ≡
[4]    cases obs_Trans(msg) of
[5]      mk_Del(_,_)
[6]        → upd_rcv(msg,j)(upd_del(msg,j)($\theta$)),
[7]      mk_Ret(_,_)
[8]        → upd_rcv(msg,j)(upd_ret(msg,j)($\theta$)),
[9]      _ → upd_rcv(msg,j)($\theta$)
[10]   end
```

(2) any message received leads to an update of a ’receive’ component of the local trader state (upd_rec). (5–6) If the received “message” constitutes a (physical package) delivery, then a ‘Merchandise’ component of the local trader state is first updated (deposit_delivery). (7–8) If the received “message” constitutes the return (of a physical package), then the ‘merchandise’ component of the local trader state is first updated (remove_return).

```
[0]  upd_rec(msg,j)($\theta$) ≡ deposit_trans((sU(msg),j),msg)(cond_rec(msg,j)($\theta$))
```

```
[1] upd_del(msg,j)($\theta$) ≡ deposit_delivery((sU(msg),j),msg)($\theta$)
[2] upd_ret(msg,j)($\theta$) ≡ remove_return((sU(msg),j),msg)($\theta$)

[3] cond_rcv(msg,j)($\theta$) ≡
[4]   if intial_trans(msg)($\theta$)
[5]     then $\theta$
[6]     else remove_prior_trans(sU(msg),j)($\theta$) end

      sU: Trans → U, sU(_,u) ≡ u
```

(0) The upd_rec operation invokes the cond_rec operation and then extends the possibly new state by depositing the argument message under the unique identification and message–sending trader identification. (3–6) The cond_rec operation examines ((4) initial_trans) whether the received message is a first such, ie., “contains” no prior transactions, or whether it contains such prior transactions. In this latter case (6) the prior transaction may be conditionally removed (remove_prior_trans) — this is not shown here, but commented upon below.

```
[0] send: i:Idx → Θ → in {tc[i,j]|j:Idx•i≠j} Θ
[1] send(i)($\theta$) ≡
[2]   let choice = ini ⌈⌉ res ⌈⌉ nor in
[3]   cases choice of
[4]     ini → send_initial(i)($\theta$),
[5]     res → send_response(i)($\theta$),
[6]     nor → remove_received_msg($\theta$) end end
```

Either a trader, when communicating a transaction chooses (2,4) an initial (ini) one, or chooses (2,5) one which is in response (res) to a message received earlier, or chooses (2,6) to not respond (nor) to such an earlier message The choice is again non–deterministic internal (2). In the last case (6) the state is thus non–deterministically internal choice updated by removing the, or an earlier received message.

Note that the above functions describe the internal as well as the external non–determinism of protocols. We omit the detailed description of those functions which can be claimed to not be proper protocol description functions — but are functions which describe updates to local trader states. We shall, below, explain more about these state–changing functions.

```
send_initial: i:Idx → Θ → out {tc[i,j]|j:Idx•i≠j} Θ
send_initial(i)($\theta$) ≡
  let choice = buy ⌈⌉ sell in
  cases choice of
    buy → send_init_buy(i)($\theta$),
    sell → send_init_sell(i)($\theta$) end end
```

```
send_response: i:Idx → $\Theta$ → **out** {tc[i,j]|j:Idx•i≠j} $\Theta$
send_response(i)($\theta$) ≡
    **let** choice = buy ⌈⌉ sell **in**
    **cases** choice **of**
        buy → send_res_buy(i)($\theta$),
        sell → send_res_sell(i)($\theta$) **end end**
```

In the above functions we have, perhaps arbitrarily chosen, to distinguish between buy and sell transactions. Both send_initial and send_response functions — as well as the four auxiliary functions they invoke — describe aspects of the protocol.

```
send_init_buy: i:Idx → $\Theta$ → **out** {tc[i,j]|j:Idx•i≠j} $\Theta$
send_init_buy(i)($\theta$) ≡
    **let** choice = inq ⌈⌉ ord ⌈⌉ pay ⌈⌉ ret ⌈⌉ ... **in**
    **let** (j,msg,$\theta'$) = prepare_init_buy(choice)(i)($\theta$) **in**
    tc[i,j]!msg ; $\theta'$ **end end**

send_init_sell: i:Idx → $\Theta$ → **out** {tc[i,j]|j:Idx•i≠j} $\Theta$
send_init_sell(i)($\theta$) ≡
    **let** choice =  quo ⌈⌉ con ⌈⌉ del ⌈⌉ inv ⌈⌉ ... **in**
    **let** (j,msg,$\theta'$) = prepare_init_sell(choice)(i)($\theta$) **in**
    tc[i,j]!msg ; $\theta'$ **end end**
```

prepare_init_buy is not a protocol function, nor is prepare_init_sell. They both assemble an initial buy, respectively sell message, msg, a target trader, j, and update a send repository state component.

```
send_res_buy: i:Idx → $\Theta$ → **out** {tc[i,j]|j:Idx•i≠j} $\Theta$
send_res_buy(i)($\theta$) ≡
    **let** ($\theta'$,msg)=sel_update_buy_state($\theta$), j=obs_trader(msg) **in**
    **let** ($\theta''$,msg′) = response_buy_msg(msg)($\theta'$) **in**
    tc[i,j]!msg′; $\theta''$ **end end**

send_res_sell: i:Idx → $\Theta$ → **out** {tc[i,j]|j:Idx•i≠j} $\Theta$
send_res_sell(i)($\theta$) ≡
    **let** ($\theta'$,msg)=sel_update_sell_state($\theta$), j=obs_trader(msg) **in**
    **let** ($\theta''$,msg′) = response_sell_msg(msg)($\theta'$) **in**
    tc[i,j]!msg′; $\theta''$ **end end**
```

sel_update_buy_state is not a protocol function, neither is sel_update_sell_-state. They both describe the selection of a previously deposited, buy, respectively a sell message, msg, (from it) the index, j, of the trader originating that message, and describes the update of a received messages repository state component. response_buy_msg and response_sell_msg both effect the assembly, from msg, of suitable response messages, msg′. As such they are partly protocol functions. Thus, if msg was an inquiry then msg′ may be either a quote, decline, or a misdirected transaction message. Etcetera.

E.2.3 On Operations on Trader States

We have left a number of trader state operations undefined. In fact we have not said anything about ‘the state’ — other than it may have a ‘received messages’ component. It likewise is expected to have a ‘sent messages’ component, a ‘catalogue’, and a ‘merchandise (wharehouse)’ component. Etcetera. To be too specific would unnecesaruly bind requirements development and bias possible software implementations.

Below we give their signature and otherwise comment informally. The reason for not formally defining them is simple: Since we are modeling the domain, and since, in the domain, these updates are typically performed by humans, and since these humans are either diligent, or sloppy, or delinquent, or outright criminal in the dispatch of their duties we really cannot define the operations as we would really like to see them dispatched — namely diligently.

value
  deposit_trans: $(\mathrm{U} \times \mathrm{Idx}) \times \mathrm{MSG} \to \Theta \to \Theta$
  deposit_delivery: $(\mathrm{U} \times \mathrm{Idx}) \times \mathrm{MSG} \to \Theta \to \Theta$
  remove_return: $(\mathrm{U} \times \mathrm{Idx}) \times \mathrm{MSG} \to \Theta \to \Theta$
  initial_trans: $\mathrm{MSG} \times \mathrm{Idx} \to \Theta \to$ **Bool**
  remove_prior_trans: $\mathrm{U} \times \mathrm{Idx} \to \Theta \to \Theta$
  remove_received_msg: $\Theta \to \Theta$

The above operations have all basically been motivated earlier. The deposit_trans unconditionally deposits a received message, for example in a part of the local trader state that could be characterised as a repository for received transactions. That repository may have messages identified by the sender and the unique identification. To specify so is not a matter of binding future requirements and therefore also not future implementations. It just models that one can, in the domain “talk” about these things.

An initial_transaction is one which does not contain prior transactions, that is: Is one which is either an inquiry transaction or contains surrogates (Sur1, Sur2).

To remove a prior transaction models that people may no longer keep a record of such a transaction — since it is embedded in the message in response to which this removal is invoked. We do not show the details of removal, but expect a model to capture that such prior transactions need not be removed. In other words: The removal may be internal non–deterministically “controlled”.

remove_received_msg unconditionally removes a message: This models that people and institutions (internal non–deterministically) may choose to ignore inquiries, quotations, orders, confirmations, deliveries, etc.

prepare_init_buy: Choice → Idx → Θ → Idx × MSG × Θ
prepare_init_sell: Choice → Idx → Θ → Idx × MSG × Θ

The above operations internal deterministically chooses which prior transactions to respond to.

obs_trader: MSG → Idx

No matter which transaction (ie., message) one can always identify, say from the unique identification, which trader originated that message. We do not specify how since that might bias an implementation.

For the sake of completeness we also state the signatures of remaining and previously described operations:

value
upd_rec: MSG × Idx → Θ → Θ
upd_del: MSG × Idx → Θ → Θ
upd_ret: MSG × Idx → Θ → Θ
cond_rcv: MSG × Idx → Θ → Θ

sel_update_buy_state: Θ → Θ × MSG
sel_update_sell_state: Θ → Θ × MSG

response_buy_msg: MSG → Θ → Θ × MSG
response_sell_msg: MSG → Θ → Θ × MSG

In summary: All operations on local trader states are, in the domain, basically under–specified. It will be a task for requirements to, as we shall call it, determine precise functionalities for each of these operations.

E.3 Discussion

As for local trader state operations, so it is for the possible sequences of transactions between "market players" (ie., the traders): They are all, in the above model, left "grossly" non–deterministic.

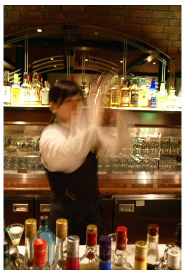
Those trader who initiate transactions toward other traders can be viewed as "clients", while those others are seen as "servers". Thus it is that we see that "clients" are characterisable by internal non–determinism, while "servers" are characterisable by external non–determinism.

It is now a task for requirements to determine the extent of non–determinisms and the more precise rôles of 'clients' and 'servers'.

Part V

Support Example Appendices

Appendices F–H (Pages 415–441) bring material in support of Chap. 7, 10 and 10 respectively. This material was not present in any of the JAIST Technical Memoranda.



F

Time and Time/Space — Two Axiom Systems

Borrowed Material: Johan van Benthem and Wayne Blizzard

Why This Appendix ?

This appendix contains some basic material on the concepts of time and of time/space. This material is, in a strict sense, required in Sects. 7.3.10 and 7.3.12 as well as in Appendix G: Time is first used in a syntactic sense, in Sect. G.2 and then in a semantics sense in Sect. G.4.

Time is ever pervasive — used in well-nigh all domain descriptions. In those that we show we model time. More axiomatic (logic) approaches to formalising time properties make use of temporal logics (`DC` [Duration Calculus] [247, 248] and `TLA+` [155, 156, 175, 176]). They are not covered in this monograph.

F.1 van Benthem's Theory of Time

The following is taken from Johan van Benthem [239]: Let P be a point structure (for example, a set). Think of time as a continuum; the following axioms characterise ordering ($<$, $=$, $>$) relations between (i.e., aspects of) time points. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the application at hand, while also fitting our intuition.

Time is transitive: If $p{<}p'$ and $p'{<}p''$ then $p{<}p''$. Time may not loop, that is, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, i.e., linear "to the left" of a given time. One could designate a time axis as beginning at some time, that is, having no predecessor times. And one can designate a time axis as ending at some time, that is, having no successor times. General, past and future successors (predecessors, respectively successors in daily talk) can be defined. Time can be dense: Given any two times one can always find a time between them. Discrete time can be defined.

axiom

[TRANS: Transitivity] $\forall$ p,p′,p″:P • p < p′ < p″ $\Rightarrow$ p < p″

[IRREF: Irreflexitivity] $\forall$ p:P • p $\not<$ p

[LIN: Linearity] $\forall$ p,p′:P • (p=p′ $\vee$ p<p′ $\vee$ p>p′)

[L−LIN: Left Linearity]
$\forall$ p,p′,p″:P • (p′<p $\wedge$ p″<p) $\Rightarrow$ (p′<p″ $\vee$ p′=p″ $\vee$ p″<p′)

[BEG: Beginning] $\exists$ p:P • ∼$\exists$ p′:P • p′<p

[END: Ending] $\exists$ p:P • ∼$\exists$ p′:P • p<p′

[SUCC: Successor]
[PAST: Predecessors] $\forall$ p:P,$\exists$ p′:P • p′<p
[FUTURE: Successor] $\forall$ p:P,$\exists$ p′:P • p<p′

[DENS: Dense] $\forall$ p,p′:P (p<p′ $\Rightarrow$ $\exists$ p″:P • p<p″<p′)

[DENS: Converse Dense] $\equiv$ [TRANS: Transitivity]
$\forall$ p,p′:P ($\exists$ p″:P • p<p″<p′ $\Rightarrow$ p<p′)

[DISC: Discrete]
$\forall$ p,p′:P • (p<p′ $\Rightarrow$ $\exists$ p″:P • (p<p″ $\wedge$ ∼$\exists$ p‴:P • (p<p‴<p″))) $\wedge$
$\forall$ p,p′:P • (p<p′ $\Rightarrow$ $\exists$ p″:P • (p″<p′ $\wedge$ ∼$\exists$ p‴:P • (p″<p‴<p′)))

A strict partial order, SPO, is a point structure satisfying TRANS and IRREF. TRANS, IRREF and SUCC imply infinite models. TRANS and SUCC may have finite, "looping time" models.

F.2 Blizard's Theory of Time-Space

We shall present an axiom system (Wayne D. Blizard, 1980, [63]) which relates abstracted entities to spatial points and time. Let $A, B, \ldots$ stand for entities, $p, q, \ldots$ for spatial points; and t, τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p, q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator applicable, pairwise to entities, spatial locations and times, respectively. A_p^t expresses that entity A is at location p at time t. We omit (obvious) typings of

A, B, P, Q, and T. The suffix prime, $'$, designates the time successor function. Thus t' designates the next time after t.

$$\begin{array}{lrcll}
(I) & \forall A \forall t \exists p & : & A^t_p & \\
(II) & & & (A^t_p \wedge A^t_q) \supset p = q & \\
(III) & & & (A^t_p \wedge B^t_p) \supset A = B & \\
(IV) & & & (A^t_p \wedge A^{t'}_p) \supset t = t' & \\
(V\ i) & \forall p,q & : & N(p,q) \supset p \neq q & \text{Irreflexivity} \\
(V\ ii) & \forall p,q & : & N(p,q) = N(q,p) & \text{Symmetry} \\
(V\ iii) & \forall p \exists q,r & : & N(p,q) \wedge N(p,r) \wedge q \neq r & \text{No isolated pts.} \\
(VI\ i) & \forall t & : & t \neq t' & \\
(VI\ ii) & \forall t & : & t' \neq 0 & \\
(VI\ iii) & \forall t & : & t \neq 0 \supset \exists \tau : t = \tau' & \\
(VI\ iv) & \forall t,\tau & : & \tau' = t' \supset \tau = t & \\
(VII) & & & A^t_p \wedge A^{t'}_q \supset N(p,q) & \\
(VIII) & & & A^t_p \wedge B^t_q \wedge N(p,q) \supset \sim (A^{t'}_q \wedge B^{t'}_p) &
\end{array}$$

- (II–IV,VII, VIII): The axioms are universally 'closed', that is, we have omitted the usual $\forall A, B, p, q, t$s.
- (I): For every entity, A, and every time, t, there is a location, p, at which A is located at time t.
- (II): An entity cannot be in two locations at the same time.
- (III): Two distinct entities cannot be at the same location at the same time.
- (IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture, and could be questioned.*
- (V): These three axioms define N.
- (V i): Same as $\forall p :\sim N(p,p)$. "Being a neighbour of", is the same as "being distinct from".
- (V ii): If p is a neighbour of q, then q is a neighbour of p.
- (V iii): Every location has at least two distinct neighbours.
- (VI): The next four axioms determine the time successor function $'$.
- (VI i): A time is always distinct from its successor: Time cannot rest. There are no time fix points.
- (VI ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.
- (VI iii): Every nonbegin time has an immediate predecessor.
- (VI iv): The time successor function $'$ is a one-to-one (i.e., a bijection) function.
- (VII): The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time t', then p and q are neighbours.
- (VIII): *No "switching":* If entities A and B occupy neighbouring locations at time t the it is not possible for A and B to have switched locations at the next time t'.

Discussion of the *Blizard* Model of Space/Time

Except for axiom (IV) the system applies to systems of entities that "sometimes" rest, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities "occupy more" space volume than others, then we may suitably "repair" the notion of the point space P (etc.), however, this is not shown here.

G

Timetable Scripts

Summary

In this appendix we cover two notions: timetables and scripts (the latter with focus on timetables as scripts).

We shall view timetables as scripts.

In this appendix we shall first narrate and formalise the syntax, including the well-formedness of timetable scripts, then we consider the pragmatics of timetable scripts, including the bus routes prescribed by these journey descriptions and timetables marked with the status of its currently active routes, and finally we consider the semantics of timetable, that is, the traffic they denote.

In the next section, Sect. 10.6, on licenses for bus traffic, we shall assume the timetable scripts of this section.

What we shall capture is, of course, an abstraction of "such timetables". We claim that the enumerated narrative which now follows and its accompanying formalisation represents an adequate description. Adequate in the sense that the reader "gets the idea", that is, is shown how to narrate and formalise when faced with an actual task of describing a concept of timetables.

In the following we distinguish between bus lines and bus rides. A bus line description is basically a sequence of two or more bus stop descriptions. A bus ride is basically a sequence of two or more time designators.[1] A bus line description may cover several bus rides. The former have unique identifications and so has the latter. The times of the latter are the approximate times at which the bus of that bus line and bus identification is supposed to be at respective stops. You may think of the bus line identification to express something like "The Flying Scotsman", and the bus ride identification something like "The 4.50 From Paddington".

[1]We do not distinguish between a time and a time description. That is, when we say February 16, 2009, 16: 58 we mean it either as a description of the time at which this text that you are now reading was LaTeX compiled, and as "that time !".

G.1 The Syntax of Bus Lines

121. A line sector is a triple: the "from" hub, to "on" link an the "to" hub.
122. A line sector description is a triple: the "from" hub identifier, to "on" link identifier and the "to" hub identifier.
123. Line sectors must be well-defined.
124. So must line sector descriptions.
125. A line is a sequence of one or more sectors
126. and its representation must be well-defined.

```
type
 121. Sect′ = H × L × H,
 122. SectDescr′ = HI × LI × HI
 123. Sect = {|(h,l,h′):Sect′ • obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
 124. SectDescr =  {|(hi,li,hi′):SectDescr′ •
                          ∃ (h,l,j′):Sect•obs_HIs(l)={obs_HI(h),obs_HI(h′)}|}
 125. Line′ = Sect*,
 126. Line = {|line:Line′•wf_Line(line)|}
value
   wf_Line: Line′ → Bool
   wf_Line(line) ≡
      ∀ i:Nat • {i,i+1}⊆inds(line) ⇒
         let (_,l,h)=line(i),(h′,l′,_)=line(i+1) in h=h′ end
```

G.2 The Syntax of Timetable Scripts

127. Time is a concept covered earlier. Bus lines and bus rides have unique names (across any set of time tables). Hub and link identifiers, HI, LI, were treated from the very beginning.
128. A TimeTable associates to Bus Line Identifiers a set of Journies.
129. Journies are designated by a pair of a BusRoute and a set of BusRides.
130. A BusRoute is a triple of the Bus Stop of origin, a list of zero, one or more intermediate Bus Stops and a destination Bus Stop.
131. A set of BusRides associates, to each of a number of Bus Identifiers a Bus Schedule.
132. A Bus Schedule a triple of the initial departure Time, a list of zero, one or more intermediate bus stop Times and a destination arrival Time.
133. A Bus Stop (i.e., its position) is a Fraction of the distance along a link (identified by a Link Identifier) from an identified hub to an identified hub.
134. A Fraction is a **Real** properly between 0 and 1.
135. The Journies must be well_formed in the context of some net.

type
127. T, BLId, BId
128. TT = BLId $\overrightarrow{m}$ Journies
129. Journies′ = BusRoute × BusRides
130. BusRoute = BusStop × BusStop* × BusStop
131. BusRides = BId $\overrightarrow{m}$ BusSched
132. BusSched = T × T* × T
133. BusStop == mkBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
134. Frac = {|r:**Real**•0<r<1|}
135. Journies = {|j:Journies′•∃ n:N • wf_Journies(j)(n)|}

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

G.3 Well-formedness of Journies

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

value
136. wf_Journies: Journies → N → **Bool**
136. wf_Journies((bs1,bsl,bsn),js)(hs,ls) ≡
137. diff_bus_stops(bs1,bsl,bsn) ∧
138. is_net_embedded_bus_line(⟨bs1⟩^bsl^⟨bsn⟩)(hs,ls) ∧
139. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)

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

value
140. diff_bus_stops: BusStop × BusStop* × BusStop → **Bool**
140. diff_bus_stops(bs1,bsl,bsn) ≡
141. **card elems** ⟨bs1⟩^bsl^⟨bsn⟩ = **len** ⟨bs1⟩^bsl^⟨bsn⟩

We shall refer to the (concatenated) list (⟨bs1⟩^bsl^⟨bsn⟩ = **len** ⟨bs1⟩^bsl^⟨bsn⟩) of all bus stops as the bus line.

142. To explain that a bus line is embedded in a line of the net
143. let us introduce the notion of all lines of the net, lns,
144. and the notion of projecting the bus line on link sector descriptors.

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

145. For a bus line to be embedded in a net then means that there exists a line, ln, in the net, such that a compressed version of the projected bus line is amongst the set of projections of that line on link sector descriptors.

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

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

```
value
146. proj_on_links: BusStop* → SectDescr*
146. proj_on_links(bsl) ≡
147.   case bsl of
148.     ⟨⟩ → ⟨⟩,
149.     ⟨mkBS(hi,li,f,hi′)⟩^bsl′ → ⟨(hi,li,hi′)⟩^proj_on_links(bsl′)
149.   end
```

150. By compression of an argument sector descriptor list we mean a result sector descriptor list with no duplicates.
151. The compress function, as a technicality, is expressed over a diminishing argument list and a diminishing argument set of sector descriptors.
152. We express the function recursively.
153. If the argument sector descriptor list an empty result sector descriptor list is yielded;
154. else
155. if the front argument sector descriptor has not yet been inserted in the result sector descriptor list it is inserted else an empty list is “inserted”
156. in front of the compression of the rest of the argument sector descriptor list.

```
150. compress: SectDescr* → SectDescr-set → SectDescr*
151. compress(sdl)(sds) ≡
152.   case sdl of
153.     ⟨⟩ → ⟨⟩,
154.     ⟨sd⟩^sdl′ →
```

```
155.        (if sd ∈ sds then ⟨sd⟩ else ⟨⟩ end)
156.        ⌢compress(sdl′)(sds\{sd}) end
```

In the last recursion iteration (line 156.) the continuation argument sds\{sd} can be shown to be empty: {}.

157. We recapitulate the definition of lines as sequences of sector descriptions.
158. Projections of a line generate a set of lists of sector descriptors.
159. Each list in such a set is some arbitrary, but ordered selection of sector descriptions. The arbitrariness is expressed by the "ranged" selection of arbitrary subsets isx of indices, isx⊆**inds** ln, into the line ln. The "orderedness" is expressed by making that arbitrary subset isx into an ordered list isl, isl=sort(isx).

```
type
157. Line′ = (HI×LI×HI)* axiom ... type Line = ... Page 420
value
158. projs_on_links: Line → Line′-set
158. projs_on_links(ln) ≡
159.   {⟨isl(i)|i:⟨1..len isl⟩⟩|isx:Nat-set•isx⊆inds ln∧isl=sort(isx)}
```

160. sorting a set of natural numbers into an ordered list, isl, of these is expressed by a post-condition relation between the argument, isx, and the result, isl.
161. The result list of (arbitrary) indices must contain all the members of the argument set;
162. and "earlier"elements of the list must precede, in value, those of "later" elements of the list.

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

163. The bus trips of a bus schedule are commensurable with the list of bus stop descriptions if the following holds:
164. All the intermediate bus stop times must equal in number that of the bus stop list.
165. We then express, by case distinction, the reality (i.e., existence) and timeliness of the bus stop descriptors and their corresponding time descriptors – and as follows.
166. If the list of intermediate bus stops is empty, then there is only the bus stops of origin and destination, and they must be exist and must fit timewise.

167. If the list of intermediate bus stops is just a singleton list, then the bus stop of origin and the singleton intermediate bus stop must exist and must fit time-wise. And likewise for the bus stop of destination and the the singleton intermediate bus stop.
168. If the list is more than a singleton list, then the first bus stop of this list must exist and must fit time-wise with the bus stop of origin.
169. As for Item 168 but now with respect to last, resp. destination bus stop.
170. And, finally, for each pair of adjacent bus stops in the list of intermediate bus stops
171. they must exist and fit time-wise.

```
value
163. commensurable_bus_trips: Journies → N → Bool
163. commensurable_bus_trips((bs1,bsl,bsn),js)(hs,ls)
164.   ∀ (t1,til,tn):BusSched•(t1,til,tn)∈ rng js∧len til=len bsl∧
165.     case len til of
166.       0 → real_and_fit((t1,t2),(bs1,bs2))(hs,ls),
167.       1 → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
167.               fit((til(1),t2),(bsl(1),bsn))(hs,ls),
168.       _ → real_and_fit((t1,til(1)),(bs1,bsl(1)))(hs,ls)∧
169.               real_and_fit((til(len til),t2),(bsl(len bsl),bsn))(hs,ls)∧
170.               ∀ i:Nat•{i,i+1}⊆inds til ⇒
171.                 real_and_fit((til(i),til(i+1)),(bsl(i),bsl(i+1)))(hs,ls) end
```

172. A pair of (adjacent) bus stops exists and a pair of times, that is the time interval between them, fit with the bus stops if the following conditions hold:
173. All the hub identifiers of bus stops must be those of net hubs (i.e., exists, are real).
174. There exists links, l, l′, for the identified bus stop links, li, li′,
175. such that these links connect the identified bus stop hubs.
176. Finally the time interval between the adjacent bus stops must approximate fit the distance between the bus stops
177. The distance between two bus stops is a loose concept as there may be many routes, short or long, between them.
178. So we leave it as an exercise to the reader to change/augment the description, in order to be able to ascertain a plausible measure of distance.
179. The approximate fit between a time interval and a distance must build on some notion of average bus velocity, etc., etc.
180. So we leave also this as an exercise to the reader to complete.

```
172. real_and_fit: (T×T)×(BusStop×BusStop) → N → Bool
172. real_and_fit((t,t′),(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴)))(hs,ls) ≡
173.   {hi,hi′,hi″,hi‴}⊆his(hs)∧
174.   ∃ l,l′:L•{l,l′}⊆ls∧(obs_LI(l)=li∧obs(l′)=li′)∧
```

175. $\quad$ obs_HIs(l)={hi,hi′}∧obs_HIs(l′)={hi″,hi‴}∧
176. $\quad$ afit(t′−t)(distance(mkBS(hi,li,f,hi′),mkBS(hi″,li′,f′,hi‴))(hs,ls))

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

179. afit: TI → Distance → **Bool**
180. $\quad$ [time interval fits distance between bus stops]

G.4 The Semantics of Timetable Scripts

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

181. We postulate a type of Buses.
182. From a bus one can observe the value of a number of attributes: current number of passengers, identity of driver, number of passengers who alighted and boarded at the most recent bus stop, etc. (We let X stand for any one of these attributes.)
183. Bus traffic maps discrete times into the pair of a bus net and the positions of buses.
184. A bus positions is either at a hub, on a link or at a bus stop.
185. When a bus is at a hub we can also observe from which link it came and to which link it proceeds.
186. When a bus is on a link we can observe how far it has progressed down the link from one of the two hubs it connects.
187. When a bus is at a bus stop — which is like "on a link" — we can observe that bus stop accordingly.
188. Fractions have also be described earlier.

type
181. Bus
value
182. obs_X: Bus → X
type
183. BusTraffic = T $\overrightarrow{m}$ (N × (BusNo $\overrightarrow{m}$ (Bus × BPos)))
184. BPos = atHub | onLnk | atBS
185. atHub == mkAtHub(s_fl:LIs_hi:HI,s_tl:LI)
186. onLnk == mkOnLnk(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
187. atBSt == mkAtBS(s_fhi:HI,s_ol:LI,s_f:Frac,s_thi:HI)
188. Frac = {|r:**Real**•0<r<1|}

We omit detailing necessary well-formedness constraints – such as (i) all bus positions being on the designated net, (ii) traffic moving monotonically, (iii)

no two buses of the same pair of bus line and bus identification at the same time (or otherwise conflicting), (iv) no "ghost" busses, etcetera. '
From a bus timetable we can generate the set of all bus traffics that satisfy the bus timetable. (We have covered this notion earlier.)

```
value
    gen_BusTraffic: TT → BusTraffic-infset
    gen_BusTraffic(tt) as btrfs
        post ∀ btrf:BusTraffic • btrf ∈ btrfs ⇒ on_time(btrf)(tt)
```

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

G.5 Discussion

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

H

The Gunter et al. Model & its Reformulation

Why This Appendix ?

This appendix, based on [113], is brought as a background text for Chap. 10. [113] appears to represent the first 'formalisation' of a license language.

Summary

This appendix presents the essence of a paper by Carl Gunter et al. [113]. That proceedings paper was studied by Drs. Arimoto Yasuhito, Chen Xiaoyi, Xiang Jianwen and myself. I presented a reformulation of the paper, as presented here, expressed in RSL. Mr. Arimoto Yasuhito presented more-or-less the same reformulation expressed in CafeOBJ, a very noce abstraction by the way!.

There were some seemingly open issues in [113]. I therefore felt a need to 'republish' an essence of the Gunter et al. paper — with our "corrections". The open issues were finally partly resolved in an e-mail from Carl Gunter, 13 Feb. 2009 (after three attempts over three years). The open issues are explicitly mentioned in itemized form, in framed paragraphs.

This written presentation is basically in colloquium form, that is, in the form of bulleted (etc.) items. Sects. H.1–H.4 "repeats" the Gunter et al. paper albeit with our "Open Issues" concerns and now with their partial resolution. Section H.5 presents my reformulation in RSL.

I thank Carl Gunter for his kind reply.

H.1 Digital Rights Licensing

H.1.1 What is Digital Rights?

- Digital rights deal with the rights of owners of artistic expressions

  - ⋆ music, movies, readings, photographs, paintings, ...
- in the context of the downloading of these
- via the Internet to users
- who are then supposed to pay for the right to render,
- i.e., to listen, see, hear, and see–see, ..., these.

H.1.2 Realities by Users and Licenses Issued by Owners

- Users perform **pay**ment and **render**ing events.
- Sequences of events as performed by users make up realities.
- Owners issue licenses which describe which realities are permissible.

H.1.3 Digital Rights Management (DRM)

- DRM is then about
  - ⋆ the design of appropriate license languages
  - ⋆ and the enforcement of user realities
  - ⋆ in order for these to not breach, but to fulfil the licenses.

H.1.4 Structure of Presentation

- First a loyal, but problematic transcription of a published paper.
- Then a "similar", but believed correct reformulation (in `RSL`).

H.2 Transcript of the Gunter/Weeks/Wright Paper

H.2.1 Actions, Events, Realities and Licenses

- The model centers around
  - ⋆ a domain of *realities* and
  - ⋆ a domain of *licenses*,
- where
  - ⋆ A *reality* is a sequence of events.
  - ⋆ A *license* is a set of realities.

- The semantics of a rights management languages can be expressed as a function
- that maps terms of the language to elements of their domain of licenses.
- Their abstract model
  - ⋆ represents an *event*, $e \in Event$, as a pair of a *time*, $t \in Time$,
  - ⋆ and an *action*, $a \in Action$:

$$e ::= t : a$$

- $Time$ is totally ordered by $<$.

- The function $+$:
  - ⋆ $Time \times Period \rightarrow Time$
  - ⋆ adds a period, $p \in Period$, to a time.
- There are two kinds of actions:

$$a ::= \mathsf{render}[w, d] \mid \mathsf{pay}[x]$$

- $w \in Work$ denotes a rights-managed work.
- $d \in Device$ represents a DRM-enabled device.
- x is a decimal.
- Action $\mathsf{render}[w, d]$ represents rendering of work w by rights-enabled device d.
- Action $\mathsf{pay}[x]$ represents a payment of amount x of some currency from a licensee to a license issuer.
- The event $t : \mathsf{render}[w, d]$ means that at time t, work w was rendered on device d.
- The event $t : \mathsf{pay}[x]$ means that at time t, a payment of amount x was made.

- A *reality*, $r \in Reality$,
  - ⋆ is a finite set of events,
  - ⋆ such that all events occur at distinct times:

$$\text{Reality} = \{\ \text{E} \mid \text{E:Event-}\mathbf{set} \bullet \text{t:a} \in \text{E} \wedge \text{t:a}' \in \text{E} \Rightarrow \text{a=a}'\}$$

  where
  - ⋆ $P(x)$ represents the powerset of E (the set of all subsets of E).
  - ⋆ Notation:
    - ◇ $r_{\leq t}$ represents the prefix of r that occurs at or before time t;
    - ◇ that is: $r_{\leq t} = \{t' : a \in r \mid t' \leq t\}$.
  - ⋆ Notation:
    - ◇ $r \sqsubseteq r'$ indicates that r is a prefix of r';
    - ◇ that is, there exists a t such that $r = r'_{\leq t}$.

- In the model, a *license*, $l \in License$, is a set of realities:

$$l \in License = P(Reality)$$

- Let us take an example:

$$l_A = \left\{ \begin{array}{l} \{8:00 : \mathsf{pay}[\$1],\ 8:01 : \mathsf{render}[w_1, d_1]\}, \\ \{8:00 : \mathsf{pay}[\$1],\ 8:02 : \mathsf{render}[w_1, d_1]\}, \\ \{8:00 : \mathsf{pay}[\$1],\ 8:03 : \mathsf{render}[w_1, d_1], \\ \qquad\qquad\qquad\quad 8:04 : \mathsf{render}[w_1, d_1]\}, \\ \{8:00 : \mathsf{pay}[\$1]\} \end{array} \right\}$$

- To give a more formal meaning to a license,
- suppose $\mathbf{r}$ is the (unique) complete reality

- that actually occurs over the entire lifetime of the DRM system.
- Let $\mathbf{r}[l]$ be events of $\mathbf{r}$ attributed to license l.
- **Definitions**
  - ⋆ Reality $r \in l$ of license l is *viable* for $\mathbf{r}[l]$ at t iff $\mathbf{r}[l]_{\leq t} \sqsubseteq r$.
  - ⋆ License l is *fulfilled* by $\mathbf{r}[l]$ at t iff $\mathbf{r}[l]_{\leq t} \in l$.
  - ⋆ License l is *breached* by $\mathbf{r}[l]$ at t iff there does not exist $r \in l$ that is viable for $\mathbf{r}[l]$ at t.

- **Example 1:**
  $\mathbf{r}[l_A] = \{$ 8:00:pay[\$1], 8:01:$\mathsf{render}[w_1, d_1]$, 8:05:$\mathsf{render}[w_1, d_1]$ $\}$.

  For $\mathbf{r}[l_A]$,
  - ⋆ at $t <$ 8:01, all four realities of license l_A are viable
  - ⋆ at t, 8:01 $\leq t <$ 8:05, only the first reality is viable
  - ⋆ at $t \geq$ 8:05, no reality is viable
- **Example 2:**
  $\mathbf{r}[l_A] = \{$ 8:00:pay[\$1], 8:01:$\mathsf{render}[w_1, d_1]$, 8:05:$\mathsf{render}[w_1, d_1]$ $\}$.

  License l_A is
  - ⋆ unfulfilled by $\mathbf{r}[l_A]$ for $t <$ 8:00
  - ⋆ fulfilled for 8:00 $\leq t <$ 8:05
  - ⋆ breached for $t \geq$ 8:05
- **Example 3:**
  $\mathbf{r}$'$[l_A] = \{$ 8:00:pay[\$1], 8:03:$\mathsf{render}[w_1, d_1]$ $\}$.

  license l_A is
  - ⋆ unfulfilled for $t <$ 8:00
  - ⋆ fulfilled for 8:03 $\leq t <$ 8:03
  - ⋆ unfulfilled for 8:03 $\leq t <$ 8:04
  - ⋆ breached for $t \geq$ 8:04

H.3 Standard Licenses

H.3.1 Up Front Licenses

- The "UP Front" license provides access
- to any work in set $W \in P(Work)$
- on any device in set $D \in P(Device)$
- beginning at time t_0 for period p,
- for an up-front payment of x:

$$\begin{aligned}
&\text{UpFront}(t_0, x, p, W, D) = \\
&\qquad \{t_0 : \mathsf{pay}[x], \\
&\qquad\quad t_1 : \mathsf{render}[w_1, d_1], ..., t_n : \mathsf{render}[w_n, d_n] \\
&\qquad | \; n \geq 0,
\end{aligned}$$

$$\begin{array}{l} t_0 < t_1 < ... < t_n < t_0 + p, \\ w_1, ..., w_n \in W, d_1, ..., d_n \in D \ \} \end{array}$$

Open Issues wrt. [113]

- The use of n is confusing.
- On one hand it is used to express up to n renderings.
- On the other hand subscript $_n$ is used for ranging both works and devices.
- The three uses of n should be separated into, say, i, j and k.[1]
- The same comments apply to the Flat Rate and Per Use formulations.
- Why not allow $t_n \leq t_0 + p$?

H.3.2 Flat Rate Licenses

- The "Flat Rate" license provides access
- to any work in set W
- on any device in set D
- beginning at time t_0 for period p,
- for a payment of x at the end of the period:

$$\begin{array}{l} \text{FlatRate}(t_0, x, p, W, D) = \\ \qquad \{t_1 : \mathsf{render}[w_1, d_1], ..., t_n : \mathsf{render}[w_n, d_n] \\ \qquad\quad t_{n+1} : \mathsf{pay}[x], \\ \qquad \mid n \geq 0, \\ \qquad\quad t_0 < t_1 < ... < t_n < t_{n+1} < t_0 + p, \\ \qquad\quad w_1, ..., w_n \in W, d_1, ..., d_n \in D \ \} \end{array}$$

Open Issues wrt. [113]

- Why not allow $t_n \leq t_0 + p$?

H.3.3 Per Use Licenses

- The "Per Use" license is provides access
- to any work in set W

[1] Gunter's 13 Feb. 2009 e-mail says that their formulation works. I think it does not: In my reformulation, in RSL, see the M(mkUF(t0,x,p,ws,ds)) function definition on Page 437. the reader will find that in order to make sure that all relevant combinations are considered that a trip set of embedded set comprehensions are necessary.

- on any device in set D
- beginning at time t_0
- for a period of length p,
- for a payment of x per use at the end of the period:

$$\begin{aligned}
&\mathrm{PerUse}(t_0, x, p, W, D) = \\
&\qquad \{t_1 : \mathsf{render}[w_1, d_1], ..., t_n : \mathsf{render}[w_n, d_n] \\
&\qquad\quad t_{n+1} : \mathsf{pay}[nx], \\
&\qquad \mid n \geq 0, \\
&\qquad\quad t_0 < t_1 < ... < t_n < t_{n+1} < t_0 + p, \\
&\qquad\quad w_1, ..., w_n \in W, d_1, ..., d_n \in D\ \}
\end{aligned}$$

Open Issues wrt. [113]

- Why not allow $t_{n+1} \leq t_0 + p$?

H.3.4 Up to Expiry Date Licenses

- A license that a consumer can accept any time before some future
  - ⋆ can be constructed by quantifying over t_0
  - ⋆ for any of the three families defined on preceding slides.
- For example,

$$\bigcup_{t_0 < t_{\text{expires}}} \mathrm{UpFront}(t_0, x, p, W, D)$$

- This license allows the period of the use to begin anytime prior to t_{expires}.

Open Issues wrt. [113]

- Why not allow $t_0 \leq t_{\text{expires}}$?

H.3.5 Non-cancellable Multi-use Licenses

- To construct multi-period non-cancellable licenses
- by composing single-period licenses,
- an operator $\triangle$ is defined.

$$l_1 \triangle l_2 = \{r_1 \cup r_2 \mid r_1 \in l_1, r_2 \in l_2\}$$

- here all of the events of l_1 occur at different times from the events of l_2.

Open Issues wrt. [113]

- The above is not the same as: $l_1 \triangle l_2 = l_1 \cup l_2$
- How does $\triangle$ associate? Gunter's 13 Feb. 2009 e-mail says: right to left.
- We guess: $l\triangle\ l'\triangle\ l'' = l\triangle\ (l'\triangle\ l'')$

. Gunter's 13 Feb. 2009 e-mail says: we were right!

- $\mathrm{UpFront}^{\triangle}(t_0, x, p, W, D, m)$
- provides access to any work in set W
- on any device in set D
- beginning at time t_0
- for m periods of length p,
- for payments of x at the beginning of each period:

$$\begin{aligned} &\mathrm{UpFront}^{\triangle}(t_0, x, p, W, D, m) = \\ &\qquad \mathrm{UpFront}(t_0, x, p, W, D)\triangle ...\triangle \\ &\qquad \mathrm{UpFront}(t_{m-1}, x, p, W, D) \\ &\qquad\quad \text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0 \text{ to } m-1 \end{aligned}$$

H.3.6 Cancellable Multi-use Licenses

- With the $\triangleright$ operator cancellable multi-period licenses can be defined.

$$l_1 \triangleright l_2 = \{r_1 \cup r_2 \mid r_1 \in l_1,\ r_1 \neq \emptyset,\ r_2 \in l_2\} \cup \{\emptyset\}$$

- here all of the events of l_1 occur at different times from the events of l_2.

Open Issues wrt. [113]

- Same question concerning associativity[2].
- What, exactly is the rôle of the empty set $\{\}$, or, rather, $\{\{\}\}$ (in Gunter et al. paper: $\{\emptyset\}$)[3].
- We guess: To have only actions from l_1 and then "get out"!
- Also: There is a problem with t_i: it is defined but never used![4]

[2]From Gunter's 13 Feb. 2009 e-mail: "The association is to the right in the right triangle in the UpFront definition. We should have said this since this operation is NOT associative."

[3]From Gunter's 13 Feb. 2009 e-mail: "Concerning the role of $\{\emptyset\}$, I don't recall why we wanted it this way but, judging by the definition, we always wanted to include the empty reality even if no other reality was in the license."

[4]From Gunter's 13 Feb. 2009 e-mail: "The notation with the t_i index is a bit awkward. The idea is that the sequence starts with $i = 0$ and ends with $i = m - 1$.

- $\text{UpFront}^{\triangleright}(t_0, x, p, W, D, m)$
- provides access to any work in set W
- on any device in set D
- beginning at time t_0
- for m periods of length p,
- for payments of x at the beginning of each period,
- cancellable after any period:

$$\begin{aligned}\text{UpFront}^{\triangleright}(t_0, x, p, W, D, m) = &\\ &\text{UpFront}(t_0, x, p, W, D) \triangleright \ldots \triangleright\\ &\text{UpFront}(t_{m-1}, x, p, W, D)\\ &\quad \text{where } t_i = t_0 + ip \text{ for } i \text{ from 0 to } m-1\end{aligned}$$

Open Issues wrt. [113]

- Same question concerning associativity.
  (See footnote 2 on the previous page.)
- There is still a problem with t_i: it is defined but never used!
  (See footnote 4 on the preceding page.)

H.4 A License Language

H.4.1 Syntax

The language *DigitalRights* is defined by the following grammar:

$$\begin{aligned}e ::= &\ (\mathsf{at}\ t \mid \mathsf{until}\ t)\\ &\ (\mathsf{for}\ p \mid \mathsf{for}\ [\mathsf{up\ \ to}]\ m\ p)\\ &\ \mathsf{pay}\ x\ (\mathsf{upfront} \mid \mathsf{flatrate} \mid \mathsf{peruse})\\ &\ \mathsf{for}\ W\ \mathsf{on}\ D\end{aligned}$$

H.4.2 Examples

```
until 01/01/03
for up to 12 months
pay $ 10.00 upfront
for "Jazz Classics"
on "devices registered to license holder"
```

To make this cleaner one could use some kind of "big triangle" notation with the index i on it. The same holds for the UpFront definition at the bottom of the page."

H.4.3 Semantics

$$\mathcal{M}[\text{at } t\ z] = \mathcal{M}_1[z](t)$$
$$\mathcal{M}[\text{until } t\ z] = \textstyle\bigcup_{t'<t} \mathcal{M}_1[z](t')$$

$$\mathcal{M}_1[\text{for } p\ z](t_0, p) = \mathcal{M}_2[z](t_0, p)$$
$$\mathcal{M}_1[\text{for up to } m\ p\ z](t_0) = \mathcal{M}_2[z](t_0, p) \rhd \ldots \rhd \mathcal{M}_2[z](t_{m-1}, p)$$
$$\text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0\ \mathit{to}\ m-1$$
$$\mathcal{M}_1[\text{for } m\ p\ z](t_0) = \mathcal{M}_2[z](t_0, p) \triangle \ldots \triangle \mathcal{M}_2[z](t_{m-1}, p)$$
$$\text{where } t_i = t_0 + ip \text{ for } i \text{ from } 0\ \mathit{to}\ m-1$$

$$\mathcal{M}_2[\text{pay } x \text{ upfront for } W \text{ on } D](t, p) = \text{UpFront}(t, x, p, W, D)$$
$$\mathcal{M}_2[\text{pay } x \text{ flatrate for } W \text{ on } D](t, p) = \text{FlatRate}(t, x, p, W, D)$$
$$\mathcal{M}_2[\text{pay } x \text{ peruse for } W \text{ on } D](t, p) = \text{PerUse}(t, x, p, W, D)$$

Open Issues wrt. [113]

- Parameters (t_0, p) in first left-hand side of $\mathcal{M}_1$ is wrong,
- should be just (t_0). (Gunter's 13 Feb. 2009 e-mail: Yes, the definition is wrong.

H.5 An RSL Model

H.5.1 Actions, Events, Realities and Licences

type
    T, W, D, P
    E = T × A
    A == mkR(w:W,d:D) | mkP(x:P) or: **rndr**(w,d) | **pay**(x)
    R = {| evs:E-**set** • wfEvs(evs) |}

Annotations:

- T, W, D, and P stands for time, work, device and payment.
- Events, e:E, are Cartesian pairs of times (i.e., time stamps) and actions.
- Actions are discriminated, either renderings (which are records mkR(w,d)) of works and devices. or payments (mkP(x)) of currency amounts.
- The trailing "or" shows our schematic way of representing actions.
- Realities, r:R, are well-formed sets of events. ■

value
    wfEvs: E-**set** → **Bool**
    wfEvs(evs) ≡ ∀ (t,a),(t′,a′):E•{(t,a),(t′,a′)}⊆evs ∧ t=t′ ⇒ a=a′

$r_{\leq t}$: prefix: R → T → R
$r_{\leq t}$ ≡ prefix(r)(t) ≡ {(t′,a)|(t′,a):E • (t′,a) ∈ r ∧ t′≤t}

$r' \sqsubseteq r$: is_prefix: R × R → **Bool**
$r' \sqsubseteq r$ ≡ is_prefix(r′,r) ≡ ∃ t:T • r′ = prefix(r)(t)

Annotations:

- A set of events form a reality if no two events have the same time stamp.
- The prefix of a reality up to and including time t is the set of all those events of the reality whose time stamp is equal to or less than t.
- A reality is a prefix of another event if there is a time t such that the former reality is a prefix of the latter reality. ▪

type
L = R-**set**

value
/∗ viable: capable of working ∗/
is_viable: R → R → L → T → **Bool**
is_viable(r′)(r)(l)(t) ≡ r ∈ l ∧ is_prefix(prefix(r′)(t),r)

/∗ fulfilled: a consumer reality r′ satisfies a license reality ∗/
is_fulfilled: R → L → T → **Bool**
is_fulfilled(r′)(l)(t) ≡ prefix(r′)(t) ∈ l

is_breached: R → L → T → **Bool**
is_breached(r′)(l)(t) ≡ ∼∃ r:R • r ∈ l ∧ is_viable(r′)(r)(l)(t)

Annotations:

- A reality r' is viable at a time t wrt. a reality r of a license l if r' is a prefix, at that time, of r.
- A reality r' is, at time t fulfilled wrt. a license l if the prefix of r' at time t is a reality of the license.
- License l is breached by consumer reality r' if there is no reality r in l that is viable for r' at t. ▪

H.5.2 Standard Licences

Syntax

type
I
Std_L = UpFront|FlatRate|PerUse|Until|NonCanMuUpFro|CanMuUpFro
Basics = T×P×I×W-**set**×D-**set**

```
    UpFront == mkUF(sb:Basics)
    FlatRate == mkFR(sb:Basics)
    PerUse == mkPU(sb:Basics)
    Until == mkU(sb:Basics,st:T)
    NonCanMuUpFro == mkNCMF(sb:Basics,sm:Nat)
    CanMuUpFro == mkCMF(sb:Basics,sm:Nat)
```

Annotations:

- I stands for an interval, i,e,, a time period.
- Std_L stands for standard licenses.
- There are six forms of standard licences: UpFront, FlatRate, PerUse, Until, NonCanMuUpFro and CanMuUpFro.
- They all share some basic information Basics, a Time limit, Payment amount, Interval, identification of the set of Works covered by the license, and identification of the set of Devises covered by the license.
- The NonCanMuUpFro and CanMuUpFro commands specify a natural, usually non-zero number (of times of rendering).
- The Std_ definition defines the set of commands as a union using the | type constructor.
- Each individual type is then defined by distinctly named record type constructors: mkUF, mkFR, mkPU, mkU, mkNCMF and mkCMF.
- We have for ease of recalling these mnemonics chosen to name the constructors with an initial mk (for 'make') and then an abbreviation of the type name being defined.
- The s...: parts of the body of the record type expressions designate selector functions.
- Meta-linguistically: ■

```
type
    A, B, ..., C
    R == mkR(sa:A,sb:B,...,sc:C)
axiom
    ∀ r:R, a:A, b:B, ..., c:C •
        r = mkR(sa(r),sb(r),...,sc(r)) ∧
        a = sa(mkR(a,b,...,c)) ∧
        b = sb(mkR(a,b,...,c)) ∧
        ... ∧
        c = sc(mkR(a,b,...,c))
```

Semantics

```
value
    M: Std_L → L
    M(mkUF(t0,x,p,ws,ds)) ≡
    {{(t0,pay(x))} ∪
       let ls = [ti↦rndr(w,d)|ti:T,w:W,d:D•ti ∈ ts∧w ∈ ws′∧d ∈ ds′] in
       {(ti,ls(ti))|ti:T•ti ∈ dom ls} end
    |n:Nat,ts:T-set,ws′:W-set,ds′:D-set•
    card ts=n∧ws′⊆ws∧ds′⊆ds∧t0<min(ts)∧max(ts)≤t0+p}
```

Annotations:

- The above formulation follows that of Gunter et al.,
- but where their model is plain wrong: it does not designate all combinations of works and devices, ours is right:
  - ⋆ At t_0 payment is issued;
  - ⋆ the above expression[5] has two set comprehensions:
  - ⋆ {{A} ∪ {B|C•D} | E•F }
  - ⋆ The inner comprehension {B|C•D} expresses all possible sequences of n renderings involving any combination of works w and devices d from subsets ws and ds of works and devices.
  - ⋆ The outer comprehension "selects" an arbitrary, finite n, a set ts of n time points all of which lies between t_0 and $t_0 + p$, and arbitrary subsets ws and ds or works and devices of those given,
  - ⋆ The inner comprehension ensures that all we express all runs of renderings of length n over all combinations of works and devices.
  - ⋆ The outer comprehension ensures that we express all possible and indefinite length runs.
- A better model would be one which, instead of constructively designating all possible runs, expresses the property that a run is an up front run and all such, for the given arguments, are present. ∎

```
type
   PayEvent = T × ({|pay|}×Nat)
   RndEvent = T × ({|render|}×W×D)
   UpFrontLicense = {| ℓs:L • wf_UPL(ℓs) |}
value
   wf_UPL: L → Bool
   wf_UPL(ℓ) ≡ ∃ t:T,x:Nat,p:P,ws:W-set,ds:D-set • is_ufl(t0,x,p)(ws,ds,ts)(ℓ)

   is_ufl: (T × Nat × P) → (W-set × D-set × T-set) → L → Bool
   is_ufl(t0,x,p)(ws,ds,ts)(ℓ) ≡
      ∃ t′:T,x′:X • (t′,(pay,x′)) ∈ ℓ⇒ t′=t0 ∧ x′=x ∧
      ∀ ti:T,w:W,d:D • ti ∈ ts ∧ w ∈ ws ∧ d ∈ ds ⇒
           (ti,(render,w,d)) ∈ ℓ∧
      ∼∃ (t,(render,w′,d′)):RndEvent •
           (t,(render,w′,d′)) ∈ ℓ∧ t∉ ts ∧ w′∉ ws ∧ d′∉ ds
```

[5] { {(t0,**pay**(x))} ∪ {(ti,**rndr**(w,d)) | ti:T,w:W,d:D • ti∈ts ∧ w∈ws′ ∧ d∈ds′ } | n:**Nat**,ts:T-**set**,ws′:W-**set**,ds′:D-**set** • **card**ts=n ∧ ws′⊆ws ∧ ds′⊆ds ∧t 0<**min**(ts) ∧ **max**(ts)≤t0+p }

- A set ℓs of licenses
- is the set of all up front licenses
- designated by M(mkUF(t0,x,p,ws,ds))
- if it satisfies are_all_ufl(t0,x,p)(ws,ds,ts)(ℓs).

```
value
  are_all_ufl: T × Nat × P × W-set × D-set → L-set → Bool
  are_all_ufl(t0,x,p)(ws,ds,ts)(ℓs) ≡
      ∀ ℓ:L • ℓ∈ ℓs ⇒
          ∃ n:Nat,ws′:W-set,ds′:D-set,ts:T-set •
              ws′⊆ws ∧ ds′⊆ds ∧ card ts=n ⇒ is_ufl(t0,x,p)(ws′,ds′,ts)(ℓ)
```

Annotations:

- The "do not exists" clauses shall ensure both largest sets of up front licenses over appropriate time spans, works, and devices and that there is no "junk".
- Otherwise we leave it to the reader to decipher the formulas. ∎

```
value
  M(mkFR(t,x,p,ws,ds)) ≡
      {let ls = [ti↦rndr(w,d)|ti:T,w:W,d:D•ti ∈ ts∧w ∈ ws′∧d ∈ ds′] in
       {(ti,ls(ti))|ti:T•ti ∈ dom ls} end
       ∪{(tn′,pay(x))}
      |n:Nat,tn′:T,ts:T-set,ws′:W-set,ds′:D-set•
      card ts=n∧ws′⊆ws∧ds′⊆ds∧t0≤min(ts)∧ max(ts)<tn′≤t0+p}
```

```
value
  M(mkPU(t,x,p,ws,ds)) ≡
      {let ls = [ti↦rndr(w,d)|ti:T,w:W,d:D•ti ∈ ts∧w ∈ ws′∧d ∈ ds′] in
       {(ti,ls(ti))|ti:T•ti ∈ dom ls} end
       ∪ {(tn′,pay(n∗x))}
      |n:Nat,tn′:T,ts:T-set,ws′:W-set,ds′:D-set•
      card ts=n∧ws′⊆ws∧ds′⊆ds∧t0≤min(ts)∧ max(ts)<tn′≤t0+p}
```

```
value
  M(mkU(te,x,p,ws,ds)) ≡ ∪{M(mkUF(t0,x,p,ws,ds))|t0:T•t0≤te}

  M(mkNCMF((t,x,p,ws,ds),m)) ≡ UpFront^Δ((t,x,p,ws,ds),m)

  M(mkCMF((t,x,p,ws,ds),m)) ≡ UpFront^▷((t,x,p,ws,ds),m)
```

```
value
  Δ: L* → L
  Δ(ll) ≡
```

```
    case ll of
        ⟨l⟩ → l,
        ⟨l⟩⌢ll′ → {r ∪ r′|r,r′:R • r ∈ l ∧ r′ ∈ Δ(ll′)}
    end

 ▷: L* → L
 ▷(ll) ≡
    case ll of
        ⟨l⟩ → l,
        ⟨l⟩⌢ll′ → {r,r ∪ r′|r,r′:R • r ∈ l ∧ r≠{} ∧ r′ ∈ ▷(ll′)}
    end
```

```
value
  UpFront^Δ: Basics × Nat → L
  UpFront^Δ((t,x,p,ws,ds),m) ≡
      Δ(⟨M(mkUF(ti,x,p,ws,ds))|i:Nat,ti:T•i:{0..m−1}∧ti=t0+i×p⟩)

  UpFront^▷: Basics × Nat → L
  UpFront^▷((t,x,p,ws,ds),m) ≡
      ▷(⟨M(mkUF(ti,x,p,ws,ds))|i:Nat,ti:T•i:{0..m−1}∧ti=t0+i×p⟩)
```

H.5.3 A License Language

```
type
  DRExpr = Time × Repetition × Payment × WorksDevices
  Time == mkAt(t:T) | mkUntil(t:T)
  Repetition == mkFor(p:I) | mkRepeat(m:Nat,p:I) | mkUpTo(m:Nat,p:I)
  Payment = P × Kind
  Kind == upfront | flatrate | peruse
  WorksDevices = W-set × D-set
```

```
value
  M0(mkAt(t),r,(x,k),wds) ≡ M1(r,(x,k),wds)(t)
  M0(mkUntil(t),r,(x,k),wds) ≡ ∪{M1(r,(x,k),wds)(t′)|t′:T•t′<t}

  M1(mkFor(p),(x,k),wds)(t) ≡ M2(x,k)(t,p)
  M1(mkRepeat(m,p),(x,k),wds)(t) ≡
      ▷(⟨M2((x,k),wds)(ti,p)|i:Nat,ti:T•i:{0..m−1}∧ti=t0+i×p⟩)
  M1(mkUpTo(m,p),(x,k),wds)(t) ≡
      Δ(⟨M2((x,k),wds)(ti,p)|i:Nat,ti:T•i:{0..m−1}∧ti=t0+i×p⟩)

  M2((x,upfront),(ws,ds))(t,p) ≡ M(mkUF(t,x,p,ws,ds))
  M2((x,upfront),(ws,ds))(t,p) ≡ M(mkFR(t,x,p,ws,ds))
  M2((x,upfront),(ws,ds))(t,p) ≡ M(mkPU(t,x,p,ws,ds))
```

H.6 End of “Gunter” Paper

What have we presented:

- On one hand:
  - ⋆ An introduction to the standard view of digital rights licenses.
- On the other hand:
  - ⋆ An illustration of
    - ◇ a pseudo-formal erroneous model versus
    - ◇ a correct formal presentation.
- Now we are ready
  - ⋆ to study digital rights license languages in general.

For that, pls. see Chap. 10.

Part VI

Administrative Appendices

I

RSL: The Raise Specification Language A Primer

This is an ultra-short introduction to the `RAISE` Specification Language, `RSL`.

I.1 Types and Values

Simplifying we consider a type to be a class (a possibly infinite set) of values, i.e., a set characterised by some unifying properties. Values are then instances of "things" that satisfy such properties. Examples of values are the numbers denoted by the numerals: 0, 1, 2, ..., etc.; the numbers denoted by the numerals: ..., -2, -1, 0, 1, 2, ..., etc.; and the numbers denoted by the numerals: ..., -5.43, -1.0, 0.0, 1.23···, 2,71828183···, 3,14159265···, 4.56 ..., etc. We shall refer to the types of these three example sets by the type names **Nat**, **Int**, **Real**. As we shall soon see, there are an infinity of types.

I.1.1 Some Distinctions

We distinguish between discrete and continuous types and hence between discrete and continuous values. By a discrete value we mean a value which is either atomic discrete or composite discrete: an atomic discrete value is a value which we are not interested in decomposing into components as it makes no sense for us to do so; a composite discrete value is a value which can be decomposed into (one or more) components which are all discrete values.

By a continuous value we mean a value which is either atomic continuous or composite continuous: an atomic continuous value is a value which we are not interested in decomposing into components as it makes no sense for us to do so and where the values of the type lie in some dense point space; a composite continuous value is a value which can be decomposed into meaningful (one or more) components some of which (one or more) are continuous values.

An Example: Atomic and Composite Types and Values

An oil pipe is a composite continuous value which consists of a pipe structure and some oil; the former being of discrete atomic value while the latter is a continuous atomic value: *"from no oil, via a tiny drop of oil, and more, say a gallon and a third of oil, to several million barrels of oil"*. Similarly for pumps, valves, depots, etc.: all are composite continuous values consisting of corresponding discrete atomic structures and continuous atomic valued (amounts of) oil.

I.1.2 An Aside

You might mistakenly think that continuous atomic values are composed from "subsegments" or "subspaces" or "subvolumes", etc., of continuous atomic values. Consider the following examples: (i) *Crude oil:* one can decompose crude oil into a very large number of molecules (of different hydrocarbons); the most commonly found molecules are alkanes (linear or branched), cycloalkanes, aromatic hydrocarbons, or more complicated chemicals like asphaltenes; but it is not a decomposition of a liter of crude oil to divide it up into ten deciliters. But if our choice of abstraction ignores the molecular structure of oil, then oil has an atomic, continuous value. (ii) *Time:* one can decompose time into years, months, weeks, days, hours, minutes and seconds; but if our choice of abstraction ignores these units, and just considers the time axis to be a linearly ordered dense set of points, then time has an atomic, continuous value. Thus we must consider the operations to be (i.e., that we wish) performed on what may appear as continuous values in order to determine our abstraction: either as atomic continuous values or as composite continuous values. Operations on oil divide a volume of oil into a set of two or more volumes of oil, displaces a volume of oil from one location into a (usually neighbouring) locations, etc. Operations on time calculates the time interval between two time points, adds an interval of time to time to obtain another time, etc. But in all these examples oil, respectively time remain atomic, continuous values.

I.2 Types

The reader is kindly asked to first study the decomposition of this section into its sub-parts and sub-sub-parts.

I.2.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of "that" type).

Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully “taken apart”.

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

Basic Types

```
type
   [1] Bool    true, false
   [2] Int     ... , −2, −2, 0, 1, 2, ...
   [3] Nat      0, 1, 2, ...
   [4] Real    ..., −5.43, −1.0, 0.0, 1.23···, 2,7182···, 3,1415···, 4.56, ...
   [5] Char    ”a”, ”b”, ..., ”0”, ...
   [6] Text    ”abracadabra”
```

An Example: Atomic Discrete Types

```
type
   DrainPumpStruct, PipeStruct, ValveStruct, FlowPumpStruct,
   FillPumpStruct, DepotStruct, SwitchStruct, ...
```

An Example: Atomic Continuous Types

```
type
   Time
   Oil, Gasoline, Gas, Ethanol, ...
```

Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully “taken apart”. There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

Concrete Composite Types

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

Composite Type Expressions

[7] A-**set**
[8] A-**infset**
[9] $A \times B \times ... \times C$
[10] A^*
[11] A^ω
[12] $A \xrightarrow[m]{} B$
[13] $A \rightarrow B$
[14] $A \overset{\sim}{\rightarrow} B$
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian $B \times C \times ... \times D$, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., $(A \xrightarrow[m]{} B)$, or (A^*)-**set**, or (A-**set**)list, or $(A|B) \xrightarrow[m]{} (C|D|(E \xrightarrow[m]{} F))$, etc.
16. The postulated disjoint union of types A, B, . . . , and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, . . . , bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

18. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Sorts and Observer Functions

```
type
    A, B, C, ..., D
value
    obs_B: A → B, obs_C: A → C, ..., obs_D: A → D
```

The above expresses that values of type A are composed from at least three values — and these are of type B, C, ..., and D. A concrete type definition corresponding to the above presupposing material of the next section

```
type
    B, C, ..., D
    A = B × C × ... × D
```

I.2.2 Type Definitions

Concrete Types

Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition

```
type
    A = Type_expr
```

Some schematic type definitions are:

Variety of Type Definitions

```
[1]  Type_name = Type_expr /* without |s or subtypes */
[2]  Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3]  Type_name ==
                mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
                ... |
                mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4]  Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[5]  Type_name = {| v:Type_name′ • 𝒫(v) |}
```

where a form of [2–3] is provided by combining the types:

Record Types

```
    Type_name = A | B | ... | Z
    A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
    B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
    ...
    Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

```
axiom
    ∀ a1:A_1, a2:A_2, ..., ai:Ai •
        s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
        ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
    ∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
        a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

Subtypes

```
type
    A = {| b:B • 𝒫(b) |}
```

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

Sorts

```
type
    A, B, ..., C
```

I.3 The RSL Predicate Calculus

I.3.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators). They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

I.3.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, . . ., k designate number values, then:

Simple Predicate Expressions

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

are simple predicate expressions.

I.3.3 Quantified Expressions

Let X, Y, . . ., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

Quantified Expressions

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are “read” as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

I.4 Concrete RSL Types: Values and Operations

I.4.1 Arithmetic

Arithmetic

type
  Nat, **Int**, **Real**
value
  $+,-,*$: **Nat**$\times$**Nat**$\rightarrow$**Nat** | **Int**$\times$**Int**$\rightarrow$**Int** | **Real**$\times$**Real**$\rightarrow$**Real**
  $/$: **Nat**$\times$**Nat**$\overset{\sim}{\rightarrow}$**Nat** | **Int**$\times$**Int**$\overset{\sim}{\rightarrow}$**Int** | **Real**$\times$**Real**$\overset{\sim}{\rightarrow}$**Real**
  $<,\leq,=,\neq,\geq,>$ (**Nat**|**Int**|**Real**) $\rightarrow$ (**Nat**|**Int**|**Real**)

I.4.2 Set Expressions

Set Enumerations

Let the below a’s denote values of type A, then the below designate simple set enumerations:

Set Enumerations

  $\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \in$ A-**set**
  $\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \in$ A-**infset**

Set Comprehension

The expression, last line below, to the right of the ≡, expresses set comprehension. The expression “builds” the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension

type
  A, B
  P = A $\rightarrow$ **Bool**

```
    Q = A $\overset{\sim}{\rightarrow}$ B
value
    comprehend: A-infset × P × Q → B-infset
    comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

I.4.3 Cartesian Expressions

Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, $\ldots$, C, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations

```
type
    A, B, ..., C
    A × B × ... × C
value
    (e1,e2,...,en)
```

I.4.4 List Expressions

List Enumerations

Let a range over values of type A, then the below expressions are simple list enumerations:

List Enumerations

```
    {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
    {⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

    ⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

List Comprehension

```
type
```

```
    A, B, P = A → **Bool**, Q = A ~→ B
value
    comprehend: A^ω × P × Q ~→ B^ω
    comprehend(l,P,Q) ≡
        ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```

I.4.5 Map Expressions

Map Enumerations

Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

Map Enumerations

```
type
    T1, T2
    M = T1 →m T2
value
    u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
    [ ],  [ u↦v ],  ...,  [ u1↦v1,u2↦v2,...,un↦vn ] ∀ ∈ M
```

Map Comprehension

The last line below expresses map comprehension:

Map Comprehension

```
type
    U, V, X, Y
    M = U →m V
    F = U ~→ X
    G = V ~→ Y
    P = U → Bool
value
    comprehend: M×F×G×P → (X →m Y)
    comprehend(m,F,G,P) ≡
        [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

I.4.6 Set Operations

Set Operator Signatures

Set Operations

```
value
  19 ∈: A × A-infset → Bool
  20 ∉: A × A-infset → Bool
  21 ∪: A-infset × A-infset → A-infset
  22 ∪: (A-infset)-infset → A-infset
  23 ∩: A-infset × A-infset → A-infset
  24 ∩: (A-infset)-infset → A-infset
  25 \: A-infset × A-infset → A-infset
  26 ⊂: A-infset × A-infset → Bool
  27 ⊆: A-infset × A-infset → Bool
  28 =: A-infset × A-infset → Bool
  29 ≠: A-infset × A-infset → Bool
  30 card: A-infset →̃ Nat
```

Set Examples

Set Examples

```
examples
  a ∈ {a,b,c}
  a ∉ {}, a ∉ {b,c}
  {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
  ∪{{a},{a,b},{a,d}} = {a,b,d}
  {a,b,c} ∩ {c,d,e} = {c}
  ∩{{a},{a,b},{a,d}} = {a}
  {a,b,c} \ {c,d} = {a,b}
  {a,b} ⊂ {a,b,c}
  {a,b,c} ⊆ {a,b,c}
  {a,b,c} = {a,b,c}
  {a,b,c} ≠ {a,b}
  card {} = 0, card {a,b,c} = 3
```

Informal Explication

19. ∈: The membership operator expresses that an element is a member of a set.

20. $\notin$: The nonmembership operator expresses that an element is not a member of a set.
21. $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22. $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23. $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24. $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
25. $\setminus$: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27. $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28. $=$: The equal operator expresses that the two operand sets are identical.
29. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions

The operations can be defined as follows ($\equiv$ is the definition symbol):

Set Operation Definitions

```
value
    s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
    s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
    s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
    s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
    s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
    s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
    s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
    card s ≡
        if s = {} then 0 else
        let a:A • a ∈ s in 1 + card (s \ {a}) end end
        pre s /∗ is a finite set ∗/
    card s ≡ chaos /∗ tests for infinity of s ∗/
```

I.4.7 Cartesian Operations

Cartesian Operations

```
type
    A, B, C
    g0: G0 = A × B × C
    g1: G1 = ( A × B × C )
    g2: G2 = ( A × B ) × C
    g3: G3 = A × ( B × C )

value
    va:A, vb:B, vc:C, vd:D
    (va,vb,vc):G0,
    (va,vb,vc):G1
    ((va,vb),vc):G2
    (va3,(vb3,vc3)):G3

decomposition expressions
    let (a1,b1,c1) = g0,
        (a1′,b1′,c1′) = g1 in .. end
    let ((a2,b2),c2) = g2 in .. end
    let (a3,(b3,c3)) = g3 in .. end
```

I.4.8 List Operations

List Operator Signatures

List Operations

value
- **hd**: $A^\omega \overset{\sim}{\to} A$
- **tl**: $A^\omega \overset{\sim}{\to} A^\omega$
- **len**: $A^\omega \overset{\sim}{\to}$ **Nat**
- **inds**: $A^\omega \to$ **Nat-infset**
- **elems**: $A^\omega \to$ A-**infset**
- .(.): $A^\omega \times$ **Nat** $\overset{\sim}{\to} A$
- $\widehat{\ }$: $A^* \times A^\omega \to A^\omega$
- =: $A^\omega \times A^\omega \to$ **Bool**
- $\neq$: $A^\omega \times A^\omega \to$ **Bool**

List Operation Examples

List Examples

examples
- **hd**$\langle$a1,a2,...,am$\rangle$=a1
- **tl**$\langle$a1,a2,...,am$\rangle$=$\langle$a2,...,am$\rangle$
- **len**$\langle$a1,a2,...,am$\rangle$=m

```
inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩^⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩
```

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, *i* larger than 0, into a list ℓ having a number of elements larger than or equal to *i*, gives the *i*th element of the list.
- $\widehat{}$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

```
value
   is_finite_list: A^ω → Bool

   len q ≡
      case is_finite_list(q) of
         true → if q = ⟨⟩ then 0 else 1 + len tl q end,
         false → chaos end

   inds q ≡
      case is_finite_list(q) of
         true → { i | i:Nat • 1 ≤ i ≤ len q },
         false → { i | i:Nat • i≠0 } end

   elems q ≡ { q(i) | i:Nat • i ∈ inds q }
```

```
q(i) ≡
   if i=1
      then
         if q≠⟨⟩
            then let a:A,q′:Q • q=⟨a⟩^q′ in a end
            else chaos end
      else q(i−1) end

fq ^ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
       | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
   pre is_finite_list(fq)

iq′ = iq″ ≡
   inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

I.4.9 Map Operations

Map Operator Signatures and Map Operation Examples

```
value
   m(a): M → A ~→ B, m(a) = b

   dom: M → A-infset [domain of map]
      dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

   rng: M → B-infset [range of map]
      rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

   †: M × M → M [override extension]
      [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

   ∪: M × M → M [merge ∪]
      [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]

   \: M × A-infset → M [restriction by]
      [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]

   /: M × A-infset → M [restriction to]
      [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]
```

$=,\neq$: M $\times$ M $\rightarrow$ **Bool**

$\circ$: (A $\overrightarrow{m}$ B) $\times$ (B $\overrightarrow{m}$ C) $\rightarrow$ (A $\overrightarrow{m}$ C) [composition]
  [a$\mapsto$b,a$'\mapsto$b$'$] $\circ$ [b$\mapsto$c,b$'\mapsto$c$'$,b$''\mapsto$c$''$] = [a$\mapsto$c,a$'\mapsto$c$'$]

Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- $\cup$: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- $\neq$: The nonequal operator expresses that the two operand maps are *not* identical.
- $\circ$: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

Map Operation Redefinitions

The map operations can also be defined as follows:

Map Operation Redefinitions

value
  rng m ≡ { m(a) | a:A • a ∈ **dom** m }

  m1 † m2 ≡
    [a$\mapsto$b | a:A,b:B •
      a ∈ **dom** m1 \ **dom** m2 ∧ b=m1(a) ∨ a ∈ **dom** m2 ∧ b=m2(a)]

```
m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

m1 = m2 ≡
   dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
m1 ≠ m2 ≡ ∼(m1 = m2)

m°n ≡
   [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
   pre rng m ⊆ dom n
```

I.5 λ-Calculus + Functions

I.5.1 The λ-Calculus Syntax

λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
   ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
   ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
   ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
   ⟨A⟩ ::= ( ⟨L⟩⟨L⟩ )
value /∗ Examples ∗/
   ⟨L⟩: e, f, a, ...
   ⟨V⟩: x, ...
   ⟨F⟩: λ x • e, ...
   ⟨A⟩: f a, (f a), f(a), (f)(a), ...
```

I.5.2 Free and Bound Variables

Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

I.5.3 Substitution

In RSL, the following rules for substitution apply:

Substitution

- **subst**([N/x]x) $\equiv$ N;
- **subst**([N/x]a) $\equiv$ a,
  for all variables a$\neq$ x;
- **subst**([N/x](P Q)) $\equiv$ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) $\equiv$ λ y•P;
- **subst**([N/x](λ y•P)) $\equiv$ λy• **subst**([N/x]P),
  if x$\neq$y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) $\equiv$ λz•**subst**([N/z]**subst**([z/y]P)),
  if y$\neq$x and y is free in N and x is free in P
  (where z is not free in (N P)).

I.5.4 α-Renaming and β-Reduction

α and β Conversions

- α-renaming: λx•M
  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)
  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) $\equiv$ **subst**([N/x]M)

I.5.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
    A, B, C
value
    obs_B: A → B,
    obs_C: A → C,
    gen_A: B×C → A
```

I.5.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions

```
value
    f: Arguments → Result
    f(args) ≡ DValueExpr

    g: Arguments $\overset{\sim}{\rightarrow}$ Result
    g(args) ≡ ValueAndStateChangeClause
    pre P(args)
```

Or functions can be defined implicitly:

Implicit Function Definitions

```
value
    f: Arguments → Result
    f(args) as result
    post P1(args,result)

    g: Arguments $\overset{\sim}{\rightarrow}$ Result
    g(args) as result
    pre P2(args)
    post P3(args,result)
```

The symbol $\overset{\sim}{\rightarrow}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

I.6 Other Applicative Expressions

I.6.1 Simple **let** Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

I.6.2 Recursive **let** Expressions

Recursive **let** expressions are written as:

Recursive **let** Expressions

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F $\equiv$ λg•λa•(E(g)) and YF = F(YF)

I.6.3 Predicative **let** Expressions

Predicative **let** expressions:

Predicative **let** Expressions

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

I.6.4 Pattern and "Wild Card" **let** Expressions

Patterns and *wild cards* can be used:

Patterns

let {a} $\cup$ s = set **in** ... **end**
let {a,_} $\cup$ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let $\langle$a$\rangle$$^\frown$$\ell$ = list **in** ... **end**
let $\langle$a,_,b$\rangle$$^\frown$$\ell$ = list **in** ... **end**

```
let [a↦b] ∪ m = map in ... end
let [a↦b,_] ∪ m = map in ... end
```

I.6.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
    if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
    choice_pattern_1 → expr_1,
    choice_pattern_2 → expr_2,
    ...
    choice_pattern_n_or_wild_card → expr_n
end
```

I.6.6 Operator/Operand Expressions

Operator/Operand Expressions

```
⟨Expr⟩ ::=
          ⟨Prefix_Op⟩ ⟨Expr⟩
         | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
         | ⟨Expr⟩ ⟨Suffix_Op⟩
         | ...
⟨Prefix_Op⟩ ::=
         − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
          = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
```

$| \in | \notin | \cup | \cap | \backslash | \subset | \subseteq | \supseteq | \supset | \hat{} | \dagger | \circ$

⟨Suffix_Op⟩ ::= !

I.7 Imperative Constructs

I.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Statements and State Change

```
    Unit
value
    stmt: Unit → Unit
    stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

I.7.2 Variables and Assignment

Variables and Assignment

```
0. variable v:Type := expression
1. v := expr
```

I.7.3 Statement Sequences and **skip**

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and **skip**

2. **skip**
3. stm_1;stm_2;...;stm_n

I.7.4 Imperative Conditionals

Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

I.7.5 Iterative Conditionals

Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

I.7.6 Iterative Sequencing

Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

I.8 Process Constructs

I.8.1 Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

Process Channels

```
channel c:A
channel { k[i]:B • i:KIdx }
```

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

I.8.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

Process Composition

```
P ∥ Q   Parallel composition
P ⌷ Q   Nondeterministic external choice (either/or)
P ⌈⌉ Q   Nondeterministic internal choice (either/or)
P ╫ Q   Interlock parallel composition
```

express the parallel (∥) of two processes, or the nondeterministic choice between two processes: either external (⌷) or internal (⌈⌉). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

I.8.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events

```
c ?, k[i] ?      Input
c ! e, k[i] ! e  Output
```

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

I.8.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Process Definitions

```
value
    P: Unit → in c out k[i]
    Unit
    Q: i:KIdx →  out c in k[i] Unit

    P() ≡ ... c ? ... k[i] ! e ...
    Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

I.9 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

Simple RSL Specifications

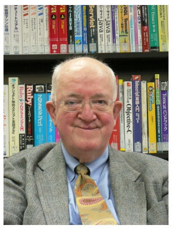
```
    type
        ...
    variable
        ...
    channel
        ...
    value
        ...
    axiom
        ...
```

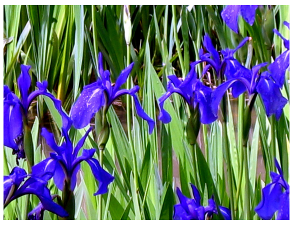
In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[1]

[1]For schemes, classes and objects we refer to [32, Chap. 10]

J

Biography





- **Family &c.:** Dines Bjørner (DB) was born in Odense, Denmark, 4 October 1937. His father had an MSc degree in Mathematics (from Copenhagen University, 1931) and his mother a BA degree in Nordic and Modern English/America Literature (also from Copenhagen University, 1929). Since 1965 DB has been married to Kari Skallerud Bjørner (Oslo, Norway). They have two children, Charlotte and Nikolaj, and five grandchildren.
- **Educational Background:** DB graduated, in 1956, with a senior high school degree in Mathematics and Natural Sciences from the Århus Cathedral School (founded in 1142). DB graduated in January 1962 with an MSc in Electronics Engineering and with a Ph.D. in Computer Science in January 1969 from the Technical University of Denmark (founded by Hans Christian Ørsted in 1828).
- **IBM Career:** DB joined IBM in March 1962 at their Nordic Laboratories (founded by Cai Kinberg) in Stockholm, Sweden (where DB also first met Jean Paul Jacob and Gunnar Wedell). DB was transferred to the IBM Systems Development Division (IBM SDD) at San Jose, California, USA, in December 1963. While doing his Ph.D. (September 1965–January 1969) DB was a lecturing consultant to IBM's European Systems Research Institute (ESRI) at Geneva, Switzerland (where DB received

valuable guidance from Carlo Santacroce and where DB's friendship with Gerald Weinberg started) (1967–1968). In 1969 DB worked at IBM's Advanced Computing Systems (IBM ACS) Laboratory, Menlo Park, California, and, later that year until early 1973 at IBM Research, San Jose (again Jean Paul Jacob became a colleague). Transferred to the IBM Vienna Laboratory (directed then by Heinz Zemanek), Austria, DB resigned from IBM in August 1975 to return to Denmark after basically 13 years abroad.

- **Career Outside and After IBM:** During his stay at IBM Research DB was a visiting lecturer, for several quarters, at University of California at Berkeley (1971–1972), instigated by Lotfi Zadeh whom DB considers his main mentor and for whom DB has the fondest regards. DB was a visiting guest professor at Copenhagen University in the academic year 1975–1976, before taking up his chair in September 1976 at the Technical University of Denmark (DTU). During the summer semester of 1980 DB was the Danish Chair Professor at the Christian-Albrechts University of Kiel, Germany — hosted by Prof. Dr. Hans Langmaack. Together with a colleague, Prof. Christian Gram, DB instigated the Dansk Datamatik Center (DDC) in the summer of 1979. During the 1980s DB was chief scientist of DDC. In 1982–1984 DB was chairman of a Danish Government (Ministry of Education) Commission on Informatics. DB was the founding and first UN Director of UNU-IIST, the United Nations University's International Institute for Software Technology, located in Macau. DB was a visiting professor at NUS: National University of Singapore in the academic year 2004–2005, and a research guest professor at JAIST, Japan Advanced Institute of Science and Technology, Ishikawa Prefecture, Japan for basically the calendar year 2006 — where the work reported in this monograph was begun. DB was a visiting professor at Université Henri Poincaré and at INRIA/LORIA, Nancy, France, for two months: Oct.–Dec., 2007. During the fall and spring of 2008–2009 DB was lecturing at the Techn. Univ. of Graz, Austria and at University of Saarland, Saarbrücken, Germany (March 2009).
- **Lectures and Graduates:** DB has lectured and regularly lectures on six continents in almost 50 countries and territories and has advised more than 130 MSc's and almost two dozen PhDs.
- **Research &c. Work:** At IBM DB first worked in the hardware (logic and systems) design of such equipment as the IBM 1070 (Sweden), the IBM 1800 and IBM 1130 computers (San Jose), and, finally, with Gene Amdahl and Ed Sussenguth, on the IBM ACS/1 supercomputer (Menlo Park). At Research DB worked with the late John W. Backus and the late Ted Codd on Functional Languages, resp. Relational Data Base Systems. At Vienna, DB, together with such colleagues as Peter Lucas, the late Hans Bekič, Kurt Walk, and Cliff B. Jones, worked on a Denotational (–like) Semantics Description of PL/I while, with his colleagues conceiving, researching, developing and using VDM (the Vienna software Development Method). At DTU and at DDC, supported by the European Community, DB initiated several advanced research & development projects: (1) Formal Semantics Description of and (2) full language compiler for CHILL (the Intl. Telecommunications Unions Communications [C.C.I.T.T.] High Level Language) — both significantly developed by Peter L. Haff (and the late Søren Prehn); (3) Formal Semantics Description of and (4) the first European US DoD officially validated compiler for the US DoD Ada embedded systems programming language — with significant and indispensable contributions by DB's colleague Dr. Hans Bruun and, again, the late Søren Prehn; (5) RAISE (Rigorous Approach to Industrial Software Engineering, headed by the late Søren Prehn

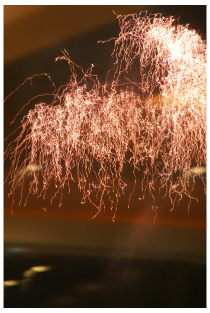
and Chris George); (6) Formal Semantics Definition of VDM–SL (the VDM Specification Language, Bo Stig Hansen and Peter Gorm Larsen); (7) ProCoS (Provably Correct Systems) with, amongst others, Profs. Sir Tony Hoare (then Oxford, now Microsoft Research, Cambridge, UK), Hans Langmaack (Kiel) and Ernst-Rüdiger Olderog (Oldenburg) and others.

- **UNU-IIST:** At UNU-IIST DB had a rather free hand, and was able, with a small team of excellent colleagues (Prof. Zhou Chaochen (Academician, the Chinese Academy of Science), the late Søren Prehn, Chris W. George, Richard Moore, Tomasz Janowski, Dang Van Hung, Xu Qi Wen and Kees Middelburg), to further explore the research issues still occupying DB's interest, and to apply them (i.e., test them out) in a number of joint R&D projects with institutions in developing and newly industrialised countries [including newly independent states] (Argentina, Belarus, Brasil, Cameroun, China, Gabon, India, Indonesia, Mongolia, North Korea, Pakistan, Philippines, Poland, Romania, Russia, South Africa, South Korea, Thailand, Vietnam, Ukraine, Uruguay, etc.).
- **Societal Work:** DB was a co-founder of VDM-Europe in 1987 and moved VDM-Europe onto FME: Formal Methods Europe in 1991. DB co-chaired two of the VDM Symposia (1987, 1990), and the International Conference on Software Engineering (ICSE) in 1989 in Pittsburgh, Pennsylvania, USA. DB was chairman of the IFIP World Congress in Dublin, Ireland in 1986, and was the instigator and General Chairman of the first World Congress on Formal Methods, FM'99, in Toulouse, France, September 20–24, 1999. DB has otherwise been involved in about 60 other scientific conferences.
- **Awards &c.:** DB is a Knight of The Danish Flag; is a member of Academia Europaea (MAE) and was chairman of its Informatics Section (2004-2009); is a member of The Russian Academy of Natural Sciences (MRANS [AB]), and of IFIP Working Groups 2.2 (1980-2004) and 2.3 (1980-2008). DB has received the John von Neumann Medal of the JvN Society of Hungary and the Ths. Masaryk Gold Medal from the Masaryk University, Brno, The Czech Republic. DB received the Danish Engineering Society's (IDA) Informatics Division's (IDA–IT) first BIT prize, March 1999. DB was given the degree of honorary doctor from the Masaryk University, Brno, The Czech Republic, in 2004. DB is an ACM Fellow and an IEEE Fellow.
- **Publications:** DB has authored more than 120 published papers and co-authored and co-edited some 15 books and written three books [31–33, 45].
- **Research Interests:** DB's research interests, since his Vienna days, have centered on programming methodology: *Methods as sets of principles for selecting and applying mathematics-based analysis and construction techniques and tools in order efficiently to construct efficient artefacts* — notably software. DB sees his main contributions to be in the research, development and propagation of formal specification principles and techniques. Currently DB focuses on the triptych of domain engineering, requirements engineering and software architecture and program organisation methods — emphasising such that relate these in mathematical as well as technical ways: (1) Intrinsic, support technology, management & organisation, rules & regulation, and human behaviour facets of domains; (2) projection, instantiation, extension and initialisation of domain requirements, etc.; (3) software architectures as refinements of domain requirements, and program organisation as refinements of machine requirements — with interface requirements (currently) being refinements of either and both!

• **Acknowledgements:** Among the very many people for whom DB has a special, professional fondness, people who have helped DB in his professional career, he wishes to bear tribute, in approximate chronological order, to (the late) Cai Kinberg, Gunnar Wedell, Jean Paul Jacob, Gerald Weinberg, Gene Amdahl, Ed Sussenguth, Tien Chi (T.C.) Chen, Lotfi Zadeh, (the late) Ted Codd, (the late) John W. Backus, Peter Lucas, Cliff Jones, (the late) Hans Bekič, Kurt Walk, Christian Gram, Ole N. Oest, Erich Neuhold, (the late) Søren Prehn, Sir Tony Hoare, Hans Langmaack, Zhou Chao Chen, Chris George and Kokichi Futatsugi.

[illegible]

Fredsvej 11, DK-2840 Holte, Denmark – February 16, 2009: 16:58



K

Consolidated Bibliography

Specification languages, techniques and tools, that cover the spectrum of domain and requirements specification, refinement and verification, are dealt with in Alloy: [146], ASM: [213,214], B/event B: [1,71], CSP [137,138,218, 222], DC [247,248] (Duration Calculus), Live Sequence Charts [80,128,153], Message Sequence Charts [142–144], RAISE [31–33,44,101,104,106] (RSL), Petri nets [148,199,210–212], Statecharts [123,124,126,127,129], Temporal Logic of Reactive Systems [168,169,187,203], TLA+ [155,156,175,176] (Temporal Logic of Actions), VDM [55,56,95,96], and Z [132,133,229,230,242]. Techniques for integrating "different" formal techniques are covered in [7,65, 69,111,216]. The recent book on Logics of Specification Languages [54] covers ASM, B/event B, CafeObj, CASL, DC, RAISE, TLA+, VDM and Z.

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
2. W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. In *Fifth International Conference on Software Reuse, ICSR'98*, Victoria, Canada.
3. Open Mobile Alliance. OMA DRM V2.0 Candidate Enabler. http://www.openmobilealliance.org/release_program/drm_v2_0.html, Sep 2005.
4. Gerald Allwein and Jon Barwise, editors. *Logical Reasoning with Diagrams*. Studies in Logic and Computation (Ed. D. M. Gabbay). Oxford Uniersity Press, 198 Madison Ave., New York, NY 10016, USA, 1996. A collection of 10 papers.
5. Derek Andrews. Report from the BSI Panel for the Standardization of VDM. In L.M̃arshall R.B̃loomfield and R. Jones, editors, *VDM '88 VDM – The Way Ahead*, pages 74–78. VDM-Europe, Springer-Verlag, September 1988.
6. Derek Andrews. How to Read the VDM-SL SC22/WG13 D152/N418. Technical report, University of Leicester, May 1991.

7. Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
8. Yasuhito Arimoto and Dines Bjørner. Hospital Healthcare: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
9. Alapan Arnab and Andrew Hutchison. Fairer Usage Contracts for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
10. Yochai Benkler. Coase's Penguin, or Linux and the Nature of the Firm. *The Yale Law Journal*, 112, 2002.
11. Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.
12. G.M. Birtwistle, O.-J.Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA* `begin`. Studentlitteratur, Lund, Sweden, 1974.
13. Dines Bjøner, Chris W. George, and Søren Prehn. Domain Analysis — a Prerequisite for Requirements Capture. Technical Report 37, UNU/IIST, P.O.Box 3058, Macau, February 1995. .
14. Dines Bjørner. Programming in the Meta-Language: A Tutorial. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language, [55]*, LNCS, pages 24–217. Springer–Verlag, 1978.
15. Dines Bjørner. An Architecture for Running Map Systems. Technical Report db/arch/01, UNU/IIST, the UN University's International Institute for Software Technology, P.O.Box 3058, Macau; E-Mail: `library@iist.unu.edu`, February 1994.
16. Dines Bjørner. Issues in International Cooperative Research — Why not Asian, African or Latin American 'Esprits' ? Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, March 1998. Paper presented at the pre–ICSE'98 Asia Pacific Forum on Software Engineering, Kyoto, Japan, April 20–21, 1998. 25 pages.
17. Dines Bjørner. Synopsis of the AMORE Project at the Technical University of Denmark: AMoR — Abstract Modelling of Railways and AMoRPH: Abstract Models of Railway Planning and Handling. Technical notes, Department of Computing Science, Technical University of Denmark, Building 343, DK–2800 Lyngby, Denmark, November 1999. .
18. Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.
19. Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press.
20. Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Oxford, UK, August 4-6 2003. Elsevier

Science Ltd. Symposium held at Tokyo, Japan. Editors: S. Tsugawa and M. Aoki.
21. Dines Bjørner. New Results and Trends in Formal Techniques for the Development of Software for Transportation Systems. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. Institut für Verkehrssicherheit und Automatisierungstechnik, Techn.Univ. of Braunschweig, Germany, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
22. Dines Bjørner. The Grand Challenge – FAQs of the R&D of a Railway Domain Theory. In *IFIP World Computer Congress, Topical Days: TRain: The Railway Domain*, IFIP, Amsterdam, The Netherlands, 2004. Kluwer Academic Press.
23. Dines Bjørner. The TRain Topical Day. In *Building the Information Society, IFIP 18th World Computer Congress, Tpical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart*, pages 607–611. Kluwer Academic Publishers, August 2004. A Foreword.
24. Dines Bjørner. Towards a Formal Model of CyberRail. In *Building the Information Society, IFIP 18th World Computer Congress, Tpical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart*, pages 657–664. Kluwer Academic Publishers, August 2004. Original report also listed some of DB's students as co–authors.
25. Dines Bjørner. Documents: A Domain Analysis. Techn. Memoranda IS–TM–2006-005, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., 19 December 2006.
26. Dines Bjørner. Documents: A Domain Analysis. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
27. Dines Bjørner. Domains and Problem Frames (Invited keynote at IWAAPF, ICSE 2006 Satellite Event, Shanghai, May 2006). Techn. Memoranda IS–TM–2006-006, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., 27 December 2006.
28. Dines Bjørner. On Domain and Domain Engineering: Prerequisites for Trustworthy Software. A Necessity for Believable Management – A Technology Management Document. Techn. Memoranda IS–TM–2006-001, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., 24 November 2006.
29. Dines Bjørner. Possible Collaborative 'Domain' Projects – ATechnology Management Brief. Techn. Memoranda IS–TM–2006-002, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., 24 November 2006.
30. Dines Bjørner. Public Government, A Domain Analysis [Presented May 1, 2006 at the UNU-IIST/Macau Government Conference on E–Government]. Techn. Memoranda IS–TM–2006-004, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., 19 December 2006.
31. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

32. Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen.
33. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design.* Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
34. Dines Bjørner. Verified Software for Ubiquitous Computing, A Proposed VSTTE/Ubiquitous Computing Foothill Proposal [Presented at 1AWCVS (First Asian Working Conference on Verified Systems), Macau 29-31 Oct., 2006]. Techn. Memoranda IS–TM–2006-003, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., 19 December 2006.
35. Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.
36. Dines Bjørner. MITS: Models of IT Security, Security Rules & Regulations: An Interpretation. Techn. Memoranda IS–TM–2007-004, ISSN 0918-7561, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn., 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., January 2007.
37. Dines Bjørner. The Rôle of Domain Engineering in Software Development. Techn. Memoranda IS–TM–2007-002, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn. ISSN 0918-7561, 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., January 2007.
38. Dines Bjørner. The Triptych Process Model: Process Assessment and Improvement. Techn. Memoranda IS–TM–2007-001, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn. ISSN 0918-7561, 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., January 2007.
39. Dines Bjørner. Transportation Systems Development. In *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation; Special Workshop Theme: Formal Methods in Avionics, Space and Transport*, ENSMA, Futuroscope, France, December 12–14 2007.
40. Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2008. (This is a new journal, published by Taylor & Francis, New York and London, edited by Philip Laplante).
41. Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.
42. Dines Bjørner. Domain Engineering. In *The 2007 Lipari PhD Summer School*, Lecture Notes in Computer Science (eds. E. Börger and A. Ferro), pages 1–102, Heidelberg, Germany, 2009. Springer. To appear. Meanwhile check with http://www2.imm.dtu.dk/~db/container-paper.pdf.
43. Dines Bjørner. *Domain Engineering.* To be submitted to Springer for evaluation in 2009, Expected published 2010. Either this book ms. is submitted or that of [44] is submitted. Decision on this to be made before Summer 2009. This book was the basis for guest lectures at University of the Saarland (Germany), March 2009.
44. Dines Bjørner. *Software Engineering, Vol. I: The Triptych Approach, Vol. II: A Model Development.* To be submitted to Springer for evaluation in 2009,

Expected published 2010. Either this book ms. is submitted or that of [43] is submitted. Decision on this to be made before Summer 2009. This book was the basis for guest lectures at Techn. Univ. of Graz, Oct.–Dec. 2008.

45. Dines Bjørner. *Domain Engineering.* JAIST Press, March 2009. This Research Monograph is based on the following 2006 Technical Memoranda from JAIST School of Information Science: [25, 27–30, 34, 36–38, 62].
46. Dines Bjørner and XiaoYi Chen. Public Government: A Domain Analysis. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
47. Dines Bjørner, Yu Lin Dong, and Søren Prehn. Domain Analyses: A Case Study of Station Management. In KICS'94: *Kunming International CASE Symposium, Yunnan Province, P.R.of China.* Software Engineering Association of Japan, 16–20 November 1994.
48. Dines Bjørner and Aser Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering. In *Festschrift for Prof. Willem Paul de Roever (Eds. Martin Steffen, Dennis Dams and Ulrich Hannemann*, volume [not known at time of submission of the current paper] of *Lecture Notes in Computer Science (eds. Martin Steffen, Dennis Dams and Ulrich Hannemann)*, pages 1–12, Heidelberg, July 2008. Springer.
49. Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup, and Søren Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997. .
50. Dines Bjørner, Chris W. George, Anne Eliabeth Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Pěnička. "UML"–ising Formal Techniques. In *INT 2004: Third International Workshop on Integration of Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 423–450. Springer–Verlag, 28 March 2004, ETAPS, Barcelona, Spain. To be published in INT–2004 Proceedings, Springer–Verlag.
51. Dines Bjørner, Chris W. George, and Søren Prehn. Scheduling and Rescheduling of Trains. Research Report 52, UNU/IIST, P.O.Box 3058, Macau, December 1995. .
52. Dines Bjørner, Chris W. George, and Søren Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice*, Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999.
53. Dines Bjørner, Chris W. George, and Søren Prehn. Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson*, P.O.Box 1299, Grand View, Texas 76050-1299, USA, 24–28 June 2002. Society for Design and Process Science.
54. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [71, 90, 95, 101, 120, 132, 176, 184, 214].* EATCS Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
55. Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS.* Springer–Verlag, 1978. This was the first monograph on *Meta-IV.* .
56. Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development.* Prentice-Hall, 1982.

57. Dines Bjørner, Dong Yu Lin, and Søren Prehn. Domain Analyses: A Case Study of Station Management. Research Report 23, UNU/IIST, P.O.Box 3058, Macau, 9 November 1994. Presented at the *1994 Kunming International CASE Symposium*: KICS'94, Yunnan Province, P.R.of China, 16–20 November 1994. .
58. Dines Bjørner and Ole N. Oest. The DDC Ada Compiler Development Project. *[59]*, pages 1–19, 1980.
59. Dines Bjørner and Ole N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer–Verlag, 1980.
60. Dines Bjørner, Søren Prehn, and Chris W. George. Formal Models of Railway Systems: Domains. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM.
61. Dines Bjørner, Søren Prehn, and Chris W. George. Formal Models of Railway Systems: Requirements. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM.
62. Dines Bjørner, Arimoto Yasuhito, Chen Xiaoyi, and Xiang Jianwen. A Family of License Languages. Techn. Memoranda IS–TM–2007-003, Graduate School of Information Science & Technology, JAIST: Japan Adv. Inst. of Sci. and Techn. ISSN 0918-7561, 1-1 Asahidai, Nomi, Ishikawa 923-1292, Hokuriku, Japan., January 2007.
63. Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.
64. B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ., USA, 1981.
65. Eerke A. Boiten, John Derrick, and Graeme Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, England, April 4-7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.
66. J.P. Bowen and M. Hinchey. Seven More Myths of Formal Methods. Technical Report PRG–TR–7–94, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, June 1994. Shorter version published in LNCS Springer Verlag FME'94 Symposium Proceedings.
67. J.P. Bowen and M. Hinchey. Ten Commandments of Formal Methods. Technical report, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, 1995.
68. Bureau Veritas. The Bureau Veritas Home Page. Electronically, on the Web: `http://www.bureauveritas.com/homepage_frameset.html`, 2005.
69. Michael J. Butler, Luigia Petre, and Kaisa Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15-18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.
70. Dominique Cansell and Dominique Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003. This paper is one of

a series: [89, 105, 133, 175, 185, 213] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

71. Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages 47–152 in [54]. Springer, 2008.
72. C.E.C. Digital Rights: Background, Systems, Assessment. Commission of The European Communities, Staff Working Paper, 2002. Brussels, 14.02.2002, SEC(2002) 197.
73. XiaoYi Chen and Dines Bjørner. Public Government: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
74. C. N. Chong, R. J. Corin, J. M. Doumen, S. Etalle, P. H. Hartel, Y. W. Law, and A. Tokmakoff. LicenseScript: a logical language for digital rights management. *Annals of telecommunications special issue on Information systems security*, 2006.
75. C. N. Chong, S. Etalle, and P. H. Hartel. Comparing Logic-based and XML-based Rights Expression Languages. In *Confederated Int. Workshops: On The Move to Meaningful Internet Systems (OTM)*, number 2889 in LNCS, pages 779–792, Catania, Sicily, Italy, 2003. Springer.
76. Cheun Ngen Chong, Ricardo Corin, and Sandro Etalle. LicenseScript: A novel digital rights languages and its semantics. In *Proc. of the Third International Conference WEB Delivering of Music (WEDELMUSIC'03)*, pages 122–129. IEEE Computer Society Press, 2003.
77. G.B. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, pages 430–440. IEEE, 1984.
78. CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer–Verlag, 2004.
79. David Crystal. *The Cambridge Encyclopedia of Language*. Cambridge University Press, 1987, 1988.
80. Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.
81. Jim Davies. Announcement: Electronic version of Communicating Sequential Processes (CSP). Published electronically: `http://www.usingcsp.com/`, 2004. Announcing revised edition of [137].
82. J.W. de Bakker. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.
83. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul De Roever, editors. *FMCO 2005: Formal Methods For Components and Objects, IV*. Fairleigh Dickinson University Press, 1–4 November, 2005.
84. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul De Roever, editors. *FMCO 2006: Formal Methods For Components and Objects, V*, volume to be assigned. to be assigned, 7–10 November, 2006.
85. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul De Roever, editors. *FMCO 2004: Formal Methods For Components and Objects,*

III, volume 3657 of *Revised Lecture Series, Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, November 2–5, 2004.

86. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul De Roever, editors. *FMCO 2003: Formal Methods For Components and Objects, II*, volume 3188 of *Revised Lecture Series, Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, November 4–7, 2003.
87. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul De Roever, editors. *FMCO 2002: Formal Methods For Components and Objects, I*, volume 2852 of *Revised Lecture Series, Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, November 5–8, 2002.
88. Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 29, 2001. ISBN 0 52180608 9, xxiv+776 pages, 5 tables, 84 figures, and 156 excercises.
89. Ražvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. CafeOBJ: Logical Foundations and Methodology. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [70, 105, 133, 175, 185, 213] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
90. Răzvan Diaconescu. *Logics of Specification Languages*, chapter A Methodological Guide to the CafeOBJ Logic, pages 153–240 in [54]. Springer, 2008.
91. Roderick W. Durmiendo and Chris W. George. Development of a Distributed Telephone Switch. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 99–130. Springer–Verlag, April 2002.
92. Myatav Erdenechimeg, Yumbayar Namstrai, and Richard C. Moore. Multi–lingual Document Processing. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 155–186. Springer–Verlag, April 2002.
93. Maria Fahlén, editor. *FME Rail Workshop # 3*, volume 3 of *FME Rail Seminars*, Falun, Sweden, May 12–14 1998. FME: Formal Methods Europe, Banverket. ESSI Project 26538. Workshop venue: Stockholm, Sweden. Organised by Banverket (Swedish Rail's Infrastructure Division), Falun, Sweden.
94. Richard Feynmann, Robert Leighton, and Matthew Sands. *The Feynmann Lectures on Physics*, volume Volumes I–II–II. Addison-Wesley, California Institute of Technology, 1963.
95. John S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages 453–487 in [54]. Springer, 2008.
96. John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using `VDM-SL`.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.
97. John S. Fitzgerald and Sunil Vadera. Unification: Specification and Development, and: Building a Theory of Unification. In *[151]*, pages 127–162 and 163–194. Prentice-Hall International, 1990.
98. Bill Flinn and Ib Holm Sørensen. CAVIAR: a case study in specification. In *[130]*, pages 79–110. Prentice-Hall International, 1987.
99. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial-–Strength Algebraic Formal Method*, Sara Burgerhartstraat 25, P.O. Box 211,

NL–1000 AE Amsterdam, The Netherlands, 2000. Elsevier. Proceedings from an April 1998 Symposium, Numazu, Japan.

100. Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing – Vol. 6. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, SINGAPORE 596224. Tel: 65-6466-5775, Fax: 65-6467-7667, E-mail: wspc@wspc.com.sg, 1998.
101. Chris George and Anne E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages 349–399 in [54]. Springer, 2008.
102. Chris W. George. A Theory of Distributing Train Rescheduling. Research Report 51, UNU/IIST, P.O.Box 3058, Macau, December 1995. Published in: Marie-Claude Gaudel and James Woodcock (eds.), *FME'96: Industrial Benefit and Advances in Formal Methods*, LNCS 1051, Springer-Verlag, 1996, pp. 499–517. .
103. Chris W. George. Distributed Train Rescheduling. Research Report 42, UNU/IIST, P.O.Box 3058, Macau, April 1995. .
104. Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
105. Chris W. George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [70, 89, 133, 175, 185, 213] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
106. Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
107. Chris W. George, Hung Dang Van, Tomasz Janowski, and Richard Moore. *Case Studies using The RAISE Method*. FACTS (Formal Aspects of Computing: Theory and Software) and FME (Formal Methods Europe). Springer–Verlag, London, 2002. This book reports on a number of case studies using RAISE (Rigorous Approach to Software Engineering). The case studies were done in the period 1994–2001 at UNU/IIST, the UN University's International Institute for Software Technology, Macau (till 20 Dec., 1997, Chinese Teritory under Portuguese administration, now a Special Administrative Region (SAR) of (the so–called People's Republic of) China).
108. Chris W. George and Mario I. Wolczko. Heap Storage, and Garbage Collection. In *[151]*, pages 195–210, and 21–233. Prentice-Hall International, 1990.
109. Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2002. 2nd Edition.
110. M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. In *Lecture Notes in Computer Science, Vol. 78*. Springer-Verlag, 1980.
111. Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1-3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
112. C.A. Gunter and J.C. Mitchell. *Theoretical Aspects of Object-oriented Programming*. The MIT Press, Cambridge, Mass., USA, 1994.

113. Carl A. Gunter, Stephen T. Weeks, and Andrew K. Wright. Models and Languages for Digtial Rights. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 4034–4038, Maui, Hawaii, USA, January 2001. IEEE Computer Society Press.
114. C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.
115. P.L. Haff, editor. *The Formal Definition of CHILL*. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1981.
116. Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.
117. Joseph Y. Halpern and Vicky Weissman. A Formal Foundation for XrML. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.
118. Michael Hammer and James A. Champy. *Reengineering the Corporation: A Manifesto for Business Revolution*. HarperCollins*Publishers*, 77–85 Fulham Palace Road, Hammersmith, London W6 8JB, UK, May 1993. 5 June 2001, Paperback.
119. Michael Hammer and Stephen A. Stanton. *The Reengineering Revolutiuon: The Handbook*. HarperCollins*Publishers*, 77–85 Fulham Palace Road, Hammersmith, London W6 8JB, UK, 1996. Paperback.
120. Michael R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages 299–347 in [54]. Springer, 2008.
121. Michael Reichhardt Hansen and Hans Rischel. *Functional Programming in Standard ML*. Addison Wesley, 1997.
122. David Harel. *Algorithmics —The Spirit of Computing*. Addison-Wesley, 1987.
123. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
124. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
125. David Harel. *The Science of Computing — Exploring the Nature and Power of Algorithms*. Addison-Wesley, April 1989.
126. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
127. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.
128. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
129. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.
130. I. Hayes, editor. *Specification Case Studies*. Prentice-Hall International, 1987.
131. Ian Hayes and Steve King. Chapters 13–17 on the formal modelling of the IBM CICS Transaction Processing System. In *[130]*, pages 179–243. Prentice-Hall International, 1987.
132. Martin C. Henson, Moshe Deutsch, and Steve Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages 489–596 in [54]. Springer, 2008.

133. Martin C. Henson, Steve Reeves, and Jonathan P. Bowen. Z Logic and its Consequences. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [70, 89, 105, 175, 185, 213] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
134. Michael G. Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995.
135. C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50:63–69, January 2003.
136. Charles Anthony Richard Hoare and Ji Feng He. *Unifying Theories of Programming*. Prentice Hall, 1997.
137. Tony Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985.
138. Tony Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [137]. See also `http://www.usingcsp.com/`.
139. Watts Humphrey. *Managing The Software Process*. Addison-Wesley, 1989. ISBN 0201180952.
140. V. Daniel Hunt. *Process Mapping: How to Reengineer Your Business Processes*. John Wiley & Sons, Inc., New York, N.Y., USA, 1996.
141. ContentGuard Inc. XrML: Extensible rights Markup Language. http://www.xrml.org, 2000.
142. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
143. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
144. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
145. J. Mike Jacka and Paulette J. Keller. *Business Process Mapping: Improving Customer Satisfaction*. John Wiley & Sons, Inc., New York, N.Y., USA, 2002.
146. Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
147. Michael A. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. ACM Press, Pearson Education. Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England, 2001.
148. Kurt Jensen. *Coloured Petri Nets*, volume 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science*. Springer–Verlag, Heidelberg, 1985, revised and corrected second version: 1997.
149. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986. Superceded by [150].
150. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.
151. C. B. Jones and R. C. Shaw. *Case Studies in Systematic Sotware Development*. Prentice-Hall International, 1990.
152. C. B. Jones and R.C.F. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.
153. Jochen Klose and Hartmut Wittke. An automata based interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *TACAS 2001*, LNCS 2031, pages 512–527. Springer-Verlag, 2001.
154. R.H. Koenen, J. Lacy, M. Mackay, and S. Mitchell. The long march to interoperable digital rights management. *Proceedings of the IEEE*, 92(6):883–897, June 2004.

155. Leslie Lamport. The Temporal Logic of Actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995.
156. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
157. P.J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.
158. P.J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation (in 2 parts). *Communications of the ACM*, 8(2-3):89–101 and 158–165, Feb.-March 1965.
159. P.J. Landin. A Formal Description of ALGOL 60. In *[233]*, pages 266–294, 1966.
160. P.J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.
161. Peter Gorm Larsen and Wiesław Pawłowski. The formal semantics of iso vdm–sl. *"Computer Standards and Interfaces"*, 17(5–6):585–602, September 1995.
162. P.G. Larsen, editor. *FME Rail Workshop # 1*, volume 1 of *FME Rail Seminars*, Forskerparken, DK–6000 Odense, Denmark, 8–9 June 1998. FME: Formal Methods Europe, IFAD. ESSI Project 26538. Workshop venue: Breukelen, The Netherlands. Organised by Origin Nederland, a member of the Philips group of companies, P.O.Box 1444, NL–3430 BK Nieuwegein, The Netherlands.
163. Thierry Lecomte and Peter Gorm Larsen, editors. *FME Rail Workshop # 5*, volume 5 of *FME Rail Seminars*. FME: Formal Methods Europe, Springer Verlag, September 20–24 1999. ESSI Project 26538. Workshop venue: Toulouse, France. Organised as part of FM'99: World Congress of Formal Methods.
164. Tran Mai Lien, Le Linh Chi, Phung Phuong Nam, Do Tien Dung, Nguyen Le Tyhu, and Chris W. George. Developing a National Financial Information System. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 131–. Springer–Verlag, April 2002.
165. Lloyd's Register. The Lloyd's Register Home Page. Electronically, on the Web: `http://www.lr.org/code/home.htm`, 2005.
166. IPR Systems Pty Ltd. Open Digital Rights Language (ODRL). http://odrl.net, 2001.
167. Gordon E. Lyon. Information Technology: A Quick-Reference List of Organizations and Standards for Digital Rights Management. NIST Special Publication 500-241, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce, Oct 2002.
168. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.
169. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.
170. Lynn C. Marshall. Line Representation on Graphics Devices. In *[151]*, pages 337–364. Prentice-Hall International, 1990.
171. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
172. John McCarthy. Towards a Mathematical Science of Computation. In C.M. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
173. John McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*. North-Holland Publ.Co., Amsterdam, 1963.

174. John McCarthy. A Formal Description of a Subset of ALGOL. In *[233]*, 1966.
175. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [70, 89, 105, 133, 185, 213] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
176. Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages 401–451 in [54]. Springer, 2008.
177. S. Michiels, K. Verslype, W. Joosen, and B. De Decker. Towards a Software Architecture for DRM. In *Proceedings of the Fifth ACM Workshop on Digital Rights Management (DRM'05)*, pages 65–74, Alexandria, Virginia, USA, Nov 2005.
178. R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, Halsted Press/John Wiley, New York, 1976.
179. R. Milner. Models of LCF. In K. Apt, editor, *Mathematical Centre Tracts 82: Found. of Comp. Sci. II, part 2*, pages 49–63. Mathematisch Centrum, 1976.
180. R. Milner. LCF: A way of doing proofs with a machine. In *Proceedings: Math. Found. of Computer Sci.*, pages 146–159. Vol. 74 of Lecture Notes in Computer Science, Springer-Verlag, 1979.
181. Markus Montigel, editor. *FME Rail Workshop #4*, volume 4 of *FME Rail Seminars*, Herzogenburgerstr. 68, A-3100 St. Pölten, Austria, February 17–19 1999. FME: Formal Methods Europe, Fachhochschulstudiengang St. Pölten. ESSI Project 26538. Workshop venue: St. Pölten, Austria. Organised by Fachhochschulstudiengang St. Pölten and Alcatel, Austria.
182. Richard C. Moore. Muffin: A Proof Assistant. In *[151]*, pages 91–126. Prentice-Hall International, 1990.
183. Carroll C. Morgan and Bernhard Suffrin. Specification of the UNIX filing system. In *[130]*, pages 45–78. Prentice-Hall International, 1987.
184. T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. *Logics of Specification Languages*, chapter CASL – the Common Algebraic Specification Language, pages 241–298 in [54]. Springer, 2008.
185. Till Mossakowski, Anne E. Haxthausen, Don Sanella, and Andrzej Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [70,89,105,133,175,213] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
186. Peter D. Mosses. *Action Semantics.* Cambridge University Press: Tracts in Theoretical Computer Science, 1992. .
187. Ben C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, England, 1986.
188. D. Mulligan and A. Burstein. Implementing copyright limitations in rights expression languages. In *Proc. of 2002 ACM Workshop on Digital Rights Management*, volume 2696 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, 2002.
189. Deirdre K. Mulligan, John Han, and Aaron J. Burstein. How DRM-Based Content Delivery Systems Disrupt Expectations of "Personal Use". In *Proc. of The 3rd International Workshop on Digital Rights Management*, pages 77–89, Washington DC, USA, Oct 2003. ACM.
190. Quoc Tao Ngo and Hung Dang Van. Formalisation of Realm–Based Spatial Data Types. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 259–286. Springer–Verlag, April 2002.

191. J. Nievergelt. Thoughts on Traffic Scheduling and Time Table Design. Technical Report dyl/9/2, ETH, Zürich, Switzerland, Informatik, ETH, CH-8092 Zurich, Switzerland, January 1994.
192. Norske Veritas. The DNV (Det Norske Veritas) Home Page. Electronically, on the Web: `http://www.dnv.com/`, 2005.
193. Ardegboyega Ojo and Tomasz Janowski. Formalsing Production Processes. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 187–217. Springer–Verlag, April 2002.
194. Pak Jong Ok, Ri Hyon Sul, and Chris W. George. A University Library System. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 81–98. Springer–Verlag, April 2002.
195. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
196. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
197. Sam Owre and N. Shankar. Writing PVS proof strategies. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/-Tactics in Higher Order Logics (STRATA 2003)*, number CP-2003-212448 in NASA Conference Publication, pages 1–15, Hampton, VA, September 2003. NASA Langley Research Center. The complete proccedings are available at http://research.nianet.org/fm-at-nia/STRATA2003/.
198. Lawrence Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.
199. Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
200. Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice–Hall, 2nd edition, 2001.
201. Nico Plat and Peter Gorm Larsen. An overview of the iso/vdm–sl standard. *Sigplan Notices*, 27(8):76–82, August 1992.
202. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
203. Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, IEEE CS FoCS, pages 46–57. Providence, Rhode Island, IEEE CS, 1977. .
204. S. Prehn. Distributed Train Time-tables and Dispatching. Technical Report SP/13/2, UNU/IIST, the UN University's International Institute for Software Technology, P.O.Box 3058, Macau; E-Mail: `library@iist.unu.edu`, July 1994. .
205. Roger S. Pressman. *Software Engineering, A Practitioner's Approach*. International Edition, Computer Science Series. McGraw–Hill, 5th edition, 1981–2001.
206. Riccardo Pucella and Vicky Weissman. A Logic for Reasoning about Digital Rights. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.
207. Riccardo Pucella and Vicky Weissman. A Formal Foundation for ODRL. In *Proc. of the Workshop on Issues in the Theory of Security (WIST'04)*, 2004.
208. Martin Pěnička and Dines Bjørner. From Railway Resource Planning to Train Operation — a Brief Survey of Complementary Formalisations. In *Building the*

Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August, 2004, Toulouse, France — Ed. Renéne Jacquart, pages 629–636. Kluwer Academic Publishers, August 2004.

209. Martin Pěnička, Albena Kirilova Strupchanska, and Dines Bjørner. Train Maintenance Routing. In *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
210. Wolfang Reisig. *A Primer in Petri Net Design*. Springer Verlag, March 1992. 120 pages.
211. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.
212. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.
213. Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [70,89,105,133,175,185] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
214. Wolfgang Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [54]. Springer, 2008.
215. John C. Reynolds. *The Semantics of Programming Languages*. Cambridge University Press, 1999.
216. Judi M.T. Romijn, Graeme P. Smith, and Jaco C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM. ISBN 3-540-30492-4.
217. A. W. Roscoe, editor. *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice Hall International, January 1994.
218. A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997. Now available on the net: http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.
219. A. W. Roscoe and J. C. P. Woodcock, editors. *A Millenium Perspective on Informatics*. Palgrave, November 2001. Festschrift for Sir Tony Hoare.
220. Pamela Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, Apr 2003.
221. David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.
222. Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.
223. Nitesh Shresta and Tomasz Janowski. Model–Based Travel Planning. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 219–242. Springer–Verlag, April 2002.
224. C. Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In *NATO Science Committee Informatics Conference*, 1995. http://scholar.google.com/url?sa=U&q=http://www.cs.wpi.edu/˜gpollice/cs509-s04/Readings/simonyi95death.pdf.
225. C. Simonyi. The Future is Intentional. *Computer*, 32(5):56–57, May 1999.

226. C. Simonyi. Intentional Programming: Asymptotic Fun? In *Position Paper, SDP Workshop, Vanderbilt University*, December, 2001. http://scholar.-google.com/url?sa=U&q=http://www.nitrd.% -gov/subcommittee/sdp/vanderbilt/position_papers/simonyi.pdf.
227. Jens Ulrik Skakkebæk. *A Larger Case Study: Railway Crossing*, chapter 7. Dept. of Computer Science, Techn. Univ. of Denmark, 1992.
228. Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 1982–2001.
229. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.
230. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
231. J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.
232. Staff of Merriam Webster. Online Dictionary: `http://www.m-w.com/home.htm`, 2004. Merriam–Webster, Inc., 47 Federal Street, P.O. Box 281, Springfield, MA 01102, USA.
233. T. B. Steel, editor. *Formal Language Description Languages,* IFIP TC-2 Work. Conf., Baden. North-Holland Publ.Co., Amsterdam, 1966.
234. Albena Kirilova Strupchanska, Martin Pěnička, and Dines Bjørner. Railway Staff Rostering. In *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems*. L'Harmattan Hongrie, 15–16 May 2003. Conf. held at Techn.Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany.
235. Toshiyuki Tanaka and Chris W. George. Proving Safety of Authentication Protocols. In *[238]*, FACIT: Formal Approaches to Computing and Information Technology, pages 243–258. Springer–Verlag, April 2002.
236. Robert Tennent. *The Semantics of Programming Languages*. Prentice–Hall Intl., 1997.
237. TÜV. The TÜV Certification Home Page. Electronically, on the Web: `http://www.tuev-cert.de/index_en.html`, 2005.
238. Hung Dang Van, Chris George, Tomasz Janowski, and Richard Moore, editors. *Specification Case Studies in RAISE*. FACIT: Formal Approaches to Computing and Information Technology. Springer–Verlag, April 2002. ISBN 1-85233-359-6.
239. Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.
240. Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 2000. 2nd Edition.
241. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.
242. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
243. J. C. P. Woodcock and M. Loomes. *Software Engineering Mathematics*. Pitman, London, 1988.

244. J.C.P. Woodcock, editor. *FME Rail Workshop # 2*, volume 2 of *FME Rail Seminars*, Parks Road, Oxford OX1 3QD, England, October 1998. FME: Formal Methods Europe, Oxford Univ., Computing Lab. ESSI Project 26538. Workshop venue: Canary Wharf, London Docklands, England. Organised by Formal Systems Ltd., Oxford. Hosted by London Underground.
245. JianWen Xiang and Dines Bjørner. The Electronic Media Industry: A Domain Analysis and a License Language. Technical note, JAIST, School of Information Science, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292, Summer 2006.
246. Yu Xinyiao. Stability of Railway Systems. Technical Report 28, UNU/IIST, P.O.Box 3058, Macau, May 1994.
247. Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.
248. Chao Chen Zhou, Charles Anthony Richard Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.
249. Chao Chen Zhou and Huiqun Yu. A Duration Model for Railway scheduling. Technical Report 24b, UNU/IIST, P.O.Box 3058, Macau, May 1994.

L

Indexes

L.1 Concept Index

L.2 Example Index

L.3 Referenced Author Index

L.4 Symbol Index