On Domains and Domain Engineering

Prerequisites for Trustworthy Software
A Necessity for Believable Project Management

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

With the attached document we wish to make the reader aware of a new dimension to software engineering.

- Automotive engineers have their application science include those of mechanics and of thermodynamics and be otherwise based on applied mathematics.
- Mobile phone engineers have their application science 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, manufaccturing, "the market" (including digital rights management), transportation, etc., raises the specter of there now emerging proper software application sciences.

In the attached (still draft) document 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.
- We show how the latter, requirements engineering, changes and becomes more stable and a more well-founded professional activitity.
- And we show, in five appendixes, "small" scale examples of domain descriptions of transportation nets, manufacturing, documents, "the market" and cyberrail.

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

Domain Engineering and Digital Rights Group, April 28, 2006
Graduate School of Information Science
JAIST: Japan Institute of Advanced Science & Technolog

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY
1990

Domain Engineering and Digitial Rights Group*
Graduate School of Information Science

On Domains and Domain Engineering

Prerequisites for Trustworthy Software

A Necessity for Believable Project Management

April 28, 2006

JAISTJapan Advanced Institute of Science and Technology
Graduate School of Information Science
1–1 Asahiday, Tatsunokuchi, Nomi
Ishakawa 923–1292, Japan

* JAIST/DEDR

Document History:

- 6th Version: 8 Marh
- 5th Version: 7 March 2006
- 4th Version: 2 March 2006
- 3rd Version: 26 February 2006
- 2nd Version: 25 February 2006
- 1st Version: 22 February 2006

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 [11–13]. 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 developement 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.

Contents

Part II APPENDICES

Part I

MAIN PART

1

FAQ: Domains and Domain Models

In this chapter we bring in some definitions related to domains (Sect. 1.1), we briefly characterise what a domain description contains (Sect. 1.2), and we overview the triptych dogma of developing software from domain models via requirements to software design (Sect. 1.3). In Sect. 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. 1.4). The chapter ends with a reference to business process engineering (Sect. 1.5).

1.1 Some Definitions

1.1.1 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 given it a name: the *name* domain, and sufficiently well distinguished from "neighbouring" universes or areas to avoid unnecessary overlap and confusion.

1.1.2 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).

1.1.3 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.

1.1.4 Domain Model and Domain Theory

By a domain theory we understand a formal model of a domain such that properties of the domain can be stated and formally verified.

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

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 next.

1.2.1 Entities and Types

They describe the 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 entities:

Financial Service Industry: Bank accounts, whether demand/deposit, savings & loan, or other are 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 entities. States of junctions (open or closed for movement across a junction from a segment to another segment) are conceptual entities.
Documents: A document is an entity.

■

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

1.2.2 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.

■

1.2.3 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!

■

1.2.4 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 between 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.

■

1.2.5 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 C–G (Pages 93–182) shows fragments of realistic domain models of (C) transportation nets, (D) manufacturing, (E) documents, (F) "the market" and (G) a futuristic railway service concept CyberRail.

1.3 The Triptych Dogma

The triptych dogma is the basis for Vol. 3 of [11–13].

The Triptych Dogma

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

1.3.1 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. Aeronautics 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 needs know "all about" the financial service industry domain theory! Similar for software applications within transportation, health care, manufacturing, air traffic, "the market", etcetera.

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.

1.3.2 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.

1.3.3 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"!

1.3.4 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.

1.4 Proper Software Development

So, for us, software development proceeds in phases:

1.4.1 Domain Engineering

If one has not already been established for the wider domain of some application, then establish first a domain model — usually with a scope thta 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:** 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.

Chapter 2 will touch upon some issues of how to construct a domain model.

1.4.2 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:** those requirements which can be expressed solely using terms of the domain. (Usually domain requirements are called functional requirements.)
- The **interface requirements:** 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:** those requirements which can be expressed solely 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.

1.4.3 Software Design

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

Chapters 27–28 of Vol. 3, [13], of [11–13] cover a number of techniques for "deriving" trustworthy software from requirements.

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 role.

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 reengineering.

1.6 Precise Narratives and Formal Specifications

Appendices C–G 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 [39, 41] specifications. We are now, at JAIST, developing corresponding models in CafeOBJ [31, 37, 38].

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.

2

FAQ: Domain Engineering

2.1 With Whom to do Domain Models?

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

Example 2.1 *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. ■

2.2 What Role "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 chapter, *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 similar careful business process descriptions.

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:

- **Intrinsics**
- **Support Technology**
- **Management & Organisation**
- **Rules & Regulations**
- **Scripts**
- **Human Behaviour**

We will briefly touch upon each of these.

2.3.1 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 2.2 *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.

2.3.2 Support Technology

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

Example 2.3 *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.

2.3.3 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 2.4 *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.

2.3.4 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 2.5 *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.

2.3.5 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 2.6 *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.

2.3.6 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 2.7 *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.

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.

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.

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.

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.

3.1 With Whom to do Requirements Models?

Same answer as given in Sect. 2.1 on page 21. See Example 2.1 on page 21.

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.

We repeat from Sect. 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 — as we shall see in the next chapter, FAQ: Business Process Re-engineering — 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.*

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.

3.3.1 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 3.1 *A Projected Transportation Domain:* In Appendix C 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:** Of those parts of a domain description which are projected onto the requirements prescription some 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 3.2 *A Determined Market Domain:* In Appendix F we present a domain description of "the market". That description is 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 3.3 *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 3.4 *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 3.5 *Two Timetable-Fitted Transport Domains:* Two groups of requirements engineers are each working on their transport requirements, one fore 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 involved both train and bus timetables. ▪

3.3.2 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 & Concept Identification:** A line by line inspection of the domain requirements shall reveal and result in a list of all shared phenomena & concepts.
- **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 3.6 *Supply Chain State Initialisation:* In requirements for a supply chain system of consumers, retailers and delivery services it is necessary to establish initial sales catalgues and delivery service tables. ▪

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

Example 3.7 *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 catalgues 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.

Example 3.8 *We refer to the next interface item: MM dialogues, and to the example, Example 3.9 of that item. While successively providing inout for segment after segment, and for junction after junction (not mentioned below) the input provider may decide, occassionally, 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 3.9 *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 named paired with blank fields for providing this kind of information. Etcetera. ▪

- **Man-Machine Physiological Interface:** The MM dialogue, besides GUI, is usually "carried" by tacticle instruments (keyboard, mouse, "pointing to the screen, depressing icons (buttons)"), by sound (microphones and louadspeakers), 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).

3.3.3 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.
- **Maintainability:** There are a number of maintainbility 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.
- **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.

3.4 How to Acquire Requirements?

Principles and techniques very much similar to those covered in Sect. 2.4 apply here. So we just refer the reader to study Page 25 — reading ‘requirements’ wherever Sect. 2.4 wrote ‘domain’.

3.5 How to Validate a Requirements Model?

Principles and techniques very much similar to those covered in Sect. 2.5 apply here. So we first refer the reader to study Page 26 — reading ‘requirements’ wherever Sect. 2.5 wrote ‘domain’. Then validation must be performed not just on the requirements prescription documents but also the underlying domain description documents.

3.6 How to Verify a Requirements Model?

Principles and techniques very much similar to those covered in Sect. 2.6 apply here. So we first refer the reader to study Page 26 — reading ‘requirements’ wherever Sect. 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.

3.7 What About Satisfiability and Feasibility?

3.8 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 [13] provides details.

3.9 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 the implicit/derivate goals of the project**.

Sections 23.3–5 of Vol. 3 [13] provides a few more details.

3.9.1 Technical Feasibility

Technical feasibility amounts to:

- **Feasibility of business process re-engineering:** Can the prescribed business process re-engineering requirements be imoplemented? 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 [13] provides details.

3.9.2 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 [13] provides a few more details.

3.9.3 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 [13] provides a few more details.

4

Conclusion

4.1 Myths and Commandments of Formal Methods

As of the year 2005 some, even respectable software engineers and academics, have problems with what they refer to as formal methods, which we prefer to call formal techniques.[1] One often finds that these sceptics voice various concerns. Some in the form of myths or claims. Other concerns reflect hesitancy with respect to how such formal techniques can be inserted into university curricula and into industry.

In this section we shall discuss these and related topics.

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

4.1.1 First Seven Myths

Anthony Hall [44] lists and dispels the following myths (claims) about formal techniques:[2]

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

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

[2]We list the "myths" and claims as enumerated in [44], but the subsequent indented comments represent our own views.

Of course, use of formal techniques cannot guarantee perfect software. It can, when properly followed, and in most cases indeed does, lead to far more appropriate software.

2. *Formal Techniques are* ***all about program proving.***

   Well, at least in these three volumes of textbooks of software engineering it is not. In these three volumes we have emphasised abstract modelling.

3. *Formal techniques are* ***only useful for safety-critical systems.***

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

4. *Formal techniques* ***require highly trained mathematicians.***

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

5. *Using formal techniques* ***increases the cost of development.***

   No. In numerous projects (some conducted under the auspices of the European Union's IT research programmes, in the 1980s and the 1990s) it has been demonstrated that using formal techniques did not increase cost of development, and in several cases it decreased the cost. For example, consider the Dansk Datamatik Center's (DDC) very successful development of a full `Ada` compiler [3, 20, 29]. DDC spent around 44 man years to develop a United States Department of Defense validated compiler — while another European and several US companies spent at least three–five times the manpower.

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

   Who says users should read formal specifications? In the present volume we have stressed the importance of concurrently developing and maintaining informal as well as formal domain descriptions, requirements prescriptions and software design specifications.

7. *Formal techniques are* ***not used on real, large-scale software.***

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

We encourage the readers to study Anthony Hall's delightful [44].

4.1.2 Seven More Myths

Jonathan P. Bowen and Michael G. Hinchey [24] builds upon Anthony Hall's analysis [44], and add a further seven "myths" and claims:[3]

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

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

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

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

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

    Yes, they are. In particular, software for military applications, in the UK, now demands the use of formal techniques.
13. *Formal techniques are* ***not supported.***

    There are now many software houses, especially in Europe, which offer consultancy advice on the use of formal techniques in other software houses' formal developments.
14. ***"Formal methods" people always use formal methods.***

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

Despite the seeming "outdatedness" of some of the seven more myths, we still encourage the readers to study Bowen and Hinchey's delightful [24].

[3] We list the "myths" and claims as enumerated in [24], but the subsequent indented comments represent our own views.

4.1.3 Ten Formal Methods Commandments

Given that the myths and claims have been disposed of in a trustworthy, believable manner, we can then go on and reiterate what has been said again and again in these volumes: When using formal techniques, please consider carefully the following sound advice from Jonathan P. Bowen and Michael G. Hinchey [25]:[4]

15. **Choose an appropriate notation.**

    Certainly.

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

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

17. **Estimate costs.**

    Always.

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

    See answer to item 13 above.

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

    See answer to item 10 above.

20. **Document sufficiently.**

    This volume in our three-volume series of textbooks on software engineering stresses this point almost to the extreme. In most other engineering practices documentation is far more extensive than what we witness today (year 2005) in software development. So follow the advice of the present volume: *Document, document, document.*

21. **Do not compromise thy quality standards.**

    In fact, tighten your quality standards.

22. **Do not be dogmatic.**

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

23. **Test, test and test again.**

    Also this point has been stressed in the present volume. Besides verification and model checking, it is indeed necessary to test. In this volume we have advocated, in Sect. **??**, an approach to testing which was tied very closely to an approach to software

[4]We list the Ten Commandments as listed in [25], but the subsequent indented comments represent our own views.

development, from demos, via skeletons, as we called them, and prototypes to actual unit code and systems. The testing went hand in hand with that development.

24. **Reuse.** There are two issues here.
    (a) **Reuse of software designs:** Reuse of modules is what is primarily referred to here. Since we have not covered, in Vol. 2, Chap. 10's treatment of 'modularity' that concept to the depth necessary for large-scale specifications and projects, we really cannot give any qualified advice in this area. It seems to this author that "reuse" is sort of a "white elephant," a desideratum that few can live up to.

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

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

    It does, however, make sense in the meaning of that of the Object Management Group's (OMG) guidelines for, say, dictionary components. The kind of dictionaries referred to have a base part which can be reused across compiler, operating system, database, and several application system developments.

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

    (b) **Reuse of domain descriptions and requirements prescriptions:** This is an altogether different matter. The very purpose of developing domain descriptions is that they be reused whenever requirements for software within the application area are being developed.

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

4.2 FAQs: Frequently Asked Questions

4.2.1 General

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

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

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

For a number of software products, insurance companies require that a certified company, like Lloyd's [69] (or such similar companies as Norwegian Veritas [76], Bureau Veritas [26] or TÜV [88]), regularly and irregularly, unannounced, inspect and, in various ways, check the development. Such insurance and "verification" companies are increasingly turning to formal techniques so their staff can understand and professionally evaluate the use of formal descriptions, prescriptions and specifications.

26. *What should/could be the languages of informal descriptions?*

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

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

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

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

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

    Whichever is most appropriate and at hand. For most developments that we know of, i.e., for most problem frames, the `RAISE` Specification `Language, RSL,` is adequate. You can then, when and as needed, augment `RSL` descriptions, prescriptions or specifications with Petri nets [64, 79–82], message sequence charts [60–62], live sequence charts [30, 51, 66], statecharts [47–50, 52] or duration calculus [91, 92] descriptions, prescriptions or specifications — or several of these. These "augmentations" were covered, to some nontrivial depth, in Vol. 2, Chaps. 12–15.

    Or you can use B [1], eventB, VDM-SL [18, 19, 34] or Z [55, 85, 86, 90] — all come, or will soon come, with suitable Petri net, message or live sequence chart, statechart, duration calculus or TLA+ [67, 71] augmentations.

RSL variants of UML's Class Diagrams may also be advisable (Vol. 2, Chap. 10).

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

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

4.2.2 Domains

1. *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.

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

   This is a contentious issue. For a specific requirements development one may be tricked into developing only an instantiated domain description, that is, a domain description that is already instantiated to the specific domain. But, as we have seen in this volume, Part **??**, it is oftentimes far more convenient to develop highly reusable, cf. item 24(b) above, domain descriptions.

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

3. *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 1, 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.

4. *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. 4.3.2).

4.2.3 Requirements

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

1. *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)!

2. *Must we formulate requirements strictly before software design?* This question could also appear in the previous section as: *Must we determine domain descriptions strictly before requirements prescriptions?*

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

4.3 Research and Tool Development

These three volumes of textbooks on software engineering represent a state of the art as of the winter of 2005/2006.

4.3.1 Evolving Principles, Techniques and Tools

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

4.3.2 Grand Challenges

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

Three Dimensions of Grand Challenges

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

Integration of Formal Techniques

Volume 2, Chaps. 10, 12–15 introduced UML class diagrams, Petri nets, message and live sequence charts, statecharts and the duration calculus. The chapters suggested that, when appropriate, these other notational, mostly diagrammatic systems be used in conjunction with, for example, `RSL`. The formal issue is: How does the semantics of `RSL` fit with the semantics of UML class diagrams, Petri nets, message and live sequence charts, statecharts and the duration calculus?

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

Much research and much experimental development still has to be done before we deploy these combinations or integrations in common industrial practice. For now they can be used in carefully monitored and integrated formal techniques guru-tutored industrial developments. We refer to a series of conferences on *IFM: (Integrated Formal Methods)*, which are held annually, for references to ongoing R&D [2, 23, 27, 43].

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

Trustworthy Evolutionary Systems Development

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

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

Domain Theories

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

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

On the Nature of "Grand Challenges"

Tony Hoare has formulated 17 criteria for a research topic to be a grand challenge. We borrow the topic lines from [56], but edit, i.e., shorten Hoare's, as usual, poignant, discussion. In other words, we strongly encourage the reader to study Hoare's paper.

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

1. **Fundamental:** It relates strongly to foundations, and the nature and limits of a discipline.
2. **Astonishing:** It implies constructing something ambitious, heretofore not imagined.
3. **Testable:** It must be objectively decidable whether a grand challenge project endeavour is succeeding or failing.
4. **Revolutionary:** It must imply radical paradigm shifts.
5. **Research-oriented:** It can be achieved by methods of academic research — and is not likely to be met solely by commercial interests.
6. **Inspiring:** Almost the entire research community must support it, enthusiastically, even while not all may be engaged in the endeavour.
7. **Understandable:** Comprehensible by — and captures the imagination of — the general public.
8. **Challenging:** Goes beyond what is initially possible and requires insight, techniques and tools not available at the start of the project.
9. **Useful:** Results in scientific or other rewards — even if the project as a whole may fail.

10. **International:** It has international scope: Participation would increase the research profile of a nation.
11. **Historical:** It will eventually be said: It was formulated years ago, and will stand for years to come.
12. **Feasible:** Reasons for previous failures are now understood and can now be overcome.
13. **Incremental:** Decomposes into identified individual research goals.
14. **Cooperative:** Calls for loosely planned cooperation between research teams.
15. **Competitive:** Encourages and benefits from competition among individuals and teams — with clear criteria on who is winning, or who has won.
16. **Effective:** General awareness and spread of results changes attitudes and activities of scientists and engineers.
17. **Risk-Managed:** Risks of failure are identified and means to meet will be applied.

4.4 Application Areas

The present three volumes on software engineering, of which the one you have in front of you now is the third, have, in their very many examples, hinted at a great number of application areas.

4.4.1 Additional Areas

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

We mention the following books:

- I. Hayes (ed.): *Specification Case Studies* (Prentice Hall, 1987), [53].
- C. Jones, R. Shaw (eds.): *Case Studies in Systematic Software Development* (Prentice Hall, 1990), [65].
- H.D. Van, C. George, T. Janowski, R. Moore (eds.) [89]: *Specification Case Studies in RAISE* (Springer, April 2002), [89].

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

4.4.2 The Examples

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

1. **The UNIX Filing System:** Chap. 4 [53] [73]
   Title explains the application.
2. **CAVIAR: Visitor Information System:** Chap. 5 [53] [36]
   A reasonably sophisticated company visitor and meeting (room reservation) system is developed.
3. **The IBM CICS Transaction System:** Chaps. 14–17 [53] [54]
   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 [65] [72]
   The design of a proof assistant system, with theorem store, proof verification, etc., is carefully argued.
5. **Unification:** Chaps. 5, 6 [65] [35]
   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 [65] [42]
   Two papers investigate heap storage and garbage collection.
7. **Graphics:** Chap. 13 [65] [70]
   Paper investigates and formalises line representations on graphics devices.
8. **A University Library System:** Chap. 3 [89] [78]
   A reasonably sophisticated library system is developed.
9. **A Radio Communications-Based Telephone Switching System:** Chap. 4 [89] [32]
   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 [89] [68]
    Developed for the Vietnam Ministry of Finance, this system involves the Taxation, the Budget and the Treasury Departments as well as all the actions within and between them: from assessment of tax bases, via the budgeting for all ministries, to the collection of taxes.
    We refer to exercise item **??** of Sect. **??** for an abstract description of the domain of this project.
11. **Multilingual Document Processing:** Chap. 6 [89] [33]
    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 [89] [77]
    A manufacturing system is developed, one which involves production cells, stock handling and all the related processes.
13. **Travel Planning:** Chap. 8 [89] [84]
    A reasonably sophisticated travel planning system is developed.
14. **Authentication:** Chap. 9 [89] [87]
    Some safety properties of authentication protocols are formulated and proven.
15. **Spatial Graphics:** Chap. 10 [89] [75]
    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.

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.
2. Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors. *IFM 99: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer–Verlag. Proceedings of 1st Intl. Conf. on IFM.
3. D. Bjørner and O. Oest. The DDC Ada Compiler Development Project. *[20]*, pages 1–19, 1980.
4. Dines Bjørner. Programming in the Meta-Language: A Tutorial. In Dines Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language, [18]*, LNCS, pages 24–217. Springer–Verlag, 1978.
5. Dines Bjørner. Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In Dines Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language, [18]*, LNCS, pages 337–374. Springer–Verlag, 1978.
6. Dines Bjørner. The Vienna Development Method: Software Abstraction and Program Synthesis. In *Mathematical Studies of Information Processing*, volume 75 of *LNCS*. Springer–Verlag, 1979. Proceedings of Conference at Research Institute for Mathematical Sciences (RIMS), University of Kyoto, August 1978.
7. 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.
8. 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.
9. 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.

10. Dines Bjørner. *Software Engineering*, volume 1: Abstraction and Modelling, vol. 2: Specification of Systems and Languages, vol. 3: Domains, Requirements and Software Design of *Texts in Theoretical Computer Science, the EATCS Series*. Springer, 2006. Chapters 12–14 of vol.2 primarily authored by Christian Krog Madsen.
11. Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
12. 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.
13. Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
14. Dines Bjørner, Yu Lin Dong, and S. 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.
15. 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.
16. 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.
17. Dines Bjørner, C.W. George, and S. 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.
18. Dines Bjørner and C.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*. [4–6].
19. Dines Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
20. Dines Bjørner and O. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer–Verlag, 1980.
21. 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.
22. 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.
23. 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–Verlag. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.
24. 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.
25. 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.
26. Bureau Veritas. The Bureau Veritas Home Page. Electronically, on the Web: `http://www.bureauveritas.com/homepage`$_f$*rameset.html*, 2005.
27. 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–Verlag. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.
28. 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: [31, 40, 55, 71, 74, 83] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
29. 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.
30. 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.
31. 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: [28, 40, 55, 71, 74, 83] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
32. Roderick W. Durmiendo and Chris W. George. Development of a Distributed Telephone Switch. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 99–130. Springer–Verlag, April 2002.
33. Myatav Erdenechimeg, Yumbayar Namstrai, and Richard C. Moore. Multi–lingual Document Processing. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 155–186. Springer–Verlag, April 2002.
34. John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using* `VDM-SL`. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.
35. John S. Fitzgerald and Sunil Vadera. Unification: Specification and Development, and: Building a Theory of Unification. In *[65]*, pages 127–162 and 163–194. Prentice-Hall International, 1990.
36. Bill Flinn and Ib Holm Sørensen. CAVIAR: a case study in specification. In *[53]*, pages 79–110. Prentice-Hall International, 1987.

37. 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.
38. 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.
39. 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.
40. 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: [28, 31, 55, 71, 74, 83] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
41. 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.
42. Chris W. George and Mario I. Wolczko. Heap Storage, and Garbage Collection. In *[65]*, pages 195–210, and 21–233. Prentice-Hall International, 1990.
43. 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–Verlag. Proceedings of 2nd Intl. Conf. on IFM.
44. Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.
45. 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.
46. 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.
47. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
48. David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.
49. David Harel and Eran Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
50. 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.
51. David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
52. David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

53. I. Hayes, editor. *Specification Case Studies*. Prentice-Hall International, 1987.
54. Ian Hayes and Steve King. Chapters 13–17 on the formal modelling of the IBM CICS Transaction Processing System. In *[53]*, pages 179–243. Prentice-Hall International, 1987.
55. 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: [28, 31, 40, 71, 74, 83] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
56. C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50:63–69, January 2003.
57. Charles Anthony Ricahrd Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004. Second edition of [58]. See also `http://www.usingcsp.com/`.
58. Charles Anthony Richard Hoare. *Communicating Sequential Processes*. Charles Anthony Richard Hoare Series in Computer Science. Prentice-Hall International, 1985. See also [57].
59. V. Daniel Hunt. *Process Mapping: How to Reengineer Your Business Processes*. John Wiley & Sons, Inc., New York, N.Y., USA, 1996.
60. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
61. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
62. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.
63. J. Mike Jacka and Paulette J. Keller. *Business Process Mapping: Improving Customer Satisfaction*. John Wiley & Sons, Inc., New York, N.Y., USA, 2002.
64. 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.
65. C. B. Jones and R. C. Shaw. *Case Studies in Systematic Sotware Development*. Prentice-Hall International, 1990.
66. 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.
67. Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
68. 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 *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 131–. Springer–Verlag, April 2002.
69. Lloyd's Register. The Lloyd's Register Home Page. Electronically, on the Web: `http://www.lr.org/code/home.htm`, 2005.
70. Lynn C. Marshall. Line Representation on Graphics Devices. In *[65]*, pages 337–364. Prentice-Hall International, 1990.
71. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [28, 31, 40, 55, 74, 83] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
72. Richard C. Moore. Muffin: A Proof Assistant. In *[65]*, pages 91–126. Prentice-Hall International, 1990.
73. Carroll C. Morgan and Bernhard Suffrin. Specification of the UNIX filing system. In *[53]*, pages 45–78. Prentice-Hall International, 1987.

74. Till Mossakowski, Anne E. Haxthausen, Don Sanella, and Andzrej Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [28, 31, 40, 55, 71, 83] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
75. Quoc Tao Ngo and Hung Dang Van. Formalsiation of Realm–Based Spatial Data Types. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 259–286. Springer–Verlag, April 2002.
76. Norske Veritas. The DNV (Det Norske Veritas) Home Page. Electronically, on the Web: `http://www.dnv.com/`, 2005.
77. Ardegboyega Ojo and Tomasz Janowski. Formalsing Production Processes. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 187–217. Springer–Verlag, April 2002.
78. Pak Jong Ok, Ri Hyon Sul, and Chris W. George. A University Library System. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 81–98. Springer–Verlag, April 2002.
79. Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
80. Wolfang Reisig. *A Primer in Petri Net Design.* Springer Verlag, March 1992. 120 pages.
81. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.
82. Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998. xi + 302 pages.
83. Wolfgang Reisig. The Expressive Power of Abstract–Sate Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [28, 31, 40, 55, 71, 74] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
84. Nitesh Shresta and Tomasz Janowski. Model–Based Travel Planning. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 219–242. Springer–Verlag, April 2002.
85. 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.
86. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
87. Toshiyuki Tanaka and Chris W. George. Proving Safety of Authentication Protocols. In *[89]*, FACIT: Formal Approaches to Computing and Information Technology, pages 243–258. Springer–Verlag, April 2002.
88. TÜV. The TÜV Certification Home Page. Electronically, on the Web: `http://www.tuev-cert.de/index`$_e$*n.html*, 2005.
89. 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.
90. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.
91. 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.

92. Chao Chen Zhou, Charles Anthony Richar Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

Part II

APPENDICES

A

Business Processes

A.1 Business Process Engineering

We rough-sketch a number of examples. In each example we start, according to the principles and techniques enunciated above, with identifying behaviours, events, and hence channels and the type of entities communicated over channels, i.e. participating in events. Hence we shall emphasise, in these examples, the behaviour, or process diagrams. We leave it to other examples to present other aspects, so that their totality yields the principles, the techniques and the tools of domain description.

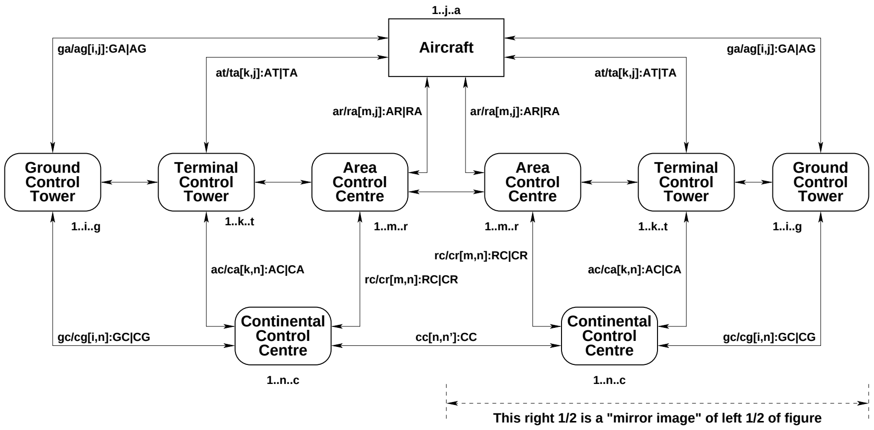


Fig. A.1. An air traffic behavioural system abstraction

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 (Fig. A.1 on the preceding page).

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 role in handling aircraft approaching airports with intention to land. They may direct these to temporarily wait in a holding area. They — eventually — guide the aircraft down, usually "stringing" them into an ordered landing queue. In doing this the terminal control towers take over the monitoring and control of landing aircraft from regional control centres, and pass their monitoring and control on to the ground control towers.

(iv) The area control centres handle aircraft flying over their territory: taking over their monitoring and control either from ground control towers, or from neighbouring area control centres. Area control centres shall help ensure smooth flight, that aircraft are allotted to appropriate air corridors, if and when needed (as one business process), and are otherwise kept informed of "neighbouring" aircraft and weather conditions en route (other business processes). Area control centres hand over aircraft either to terminal control towers (as yet another business process), or to neighbouring area control centres (as yet another business process).

(v) The continental control centres monitor and control, in collaboration with regional and ground control centres, overall traffic in an area comprising several regional control centres (as a major business process), and can thus monitor and control whether contracted (landing) slot allocations and schedules can be honoured, and, if not, reschedule these (landing) slots (as another major business process).

From the above rough sketches of behaviours the domain engineer then goes on to describe types of messages (i.e., entities) between behaviours, types of entities specific to the behaviours, and the functions that apply to or yield those entities.

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). A detailed description for each of the freight logistics business process behaviours listed above should now follow. We leave this as an exercise to the reader to complete.

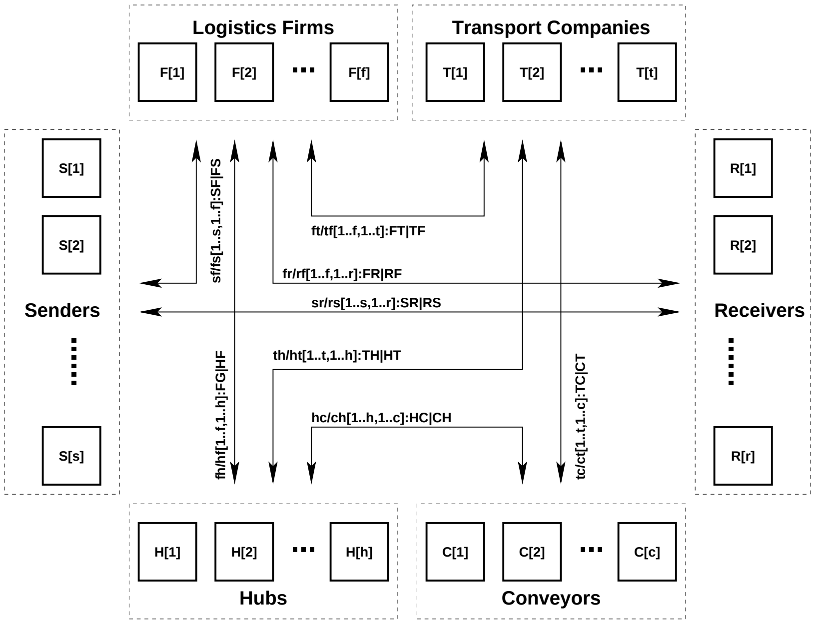


Fig. A.2. A freight logistics behavioural system abstraction

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). There may be other parts of a harbour: a holding area for ships to wait before being allowed to properly enter the harbour and be berthed at a buoy or a quay, or for ships to rest before proceeding; as well as buoys at which ships may be anchored while unloading and loading. We shall assume that the reader can properly complete an appropriate, realistic harbour domain.

A detailed description for each of the harbour business process behaviours listed above should now follow. We leave this as an exercise to the reader to complete.

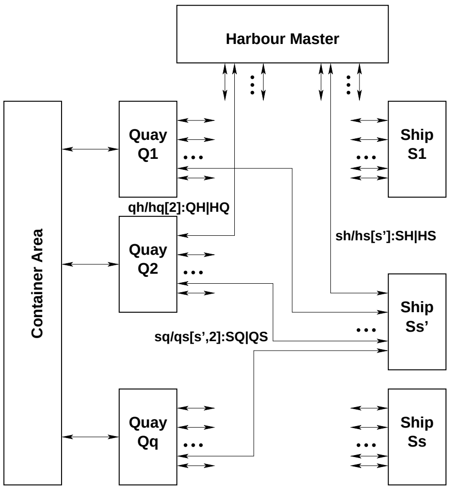


Fig. A.3. A harbour behavioural system abstraction

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.

(ii) Banks engage with clients, portfolio managers, and brokers and traders in exchanges related to client transactions with banks, portfolio managers, and brokers and traders, as well as with these on their own behalf, as clients.

(iii) Securities instrument brokers and traders engage with clients, portfolio managers and the stock exchange(s) in exchanges related to client transactions with brokers and traders, and, for traders, as well as with the stock exchange(s) on their own behalf, as clients.

(iv) Portfolio managers engage with clients, banks, and brokers and traders in exchanges related to client portfolios.

(v) Stock exchanges engage with the financial service industry watchdog, with brokers and traders, and with the stock listed enterprises, reinforcing trading practices, possibly suspending trading of stocks of enterprises, etc.

(vi) Stock incorporated enterprises engage with the stock exchange: They send reports, according to law, of possible major acquisitions, business developments, and quarterly and annual stockholder and other reports.

(vii) The financial industry watchdog engages with banks, portfolio managers, brokers and traders and with the stock exchanges.

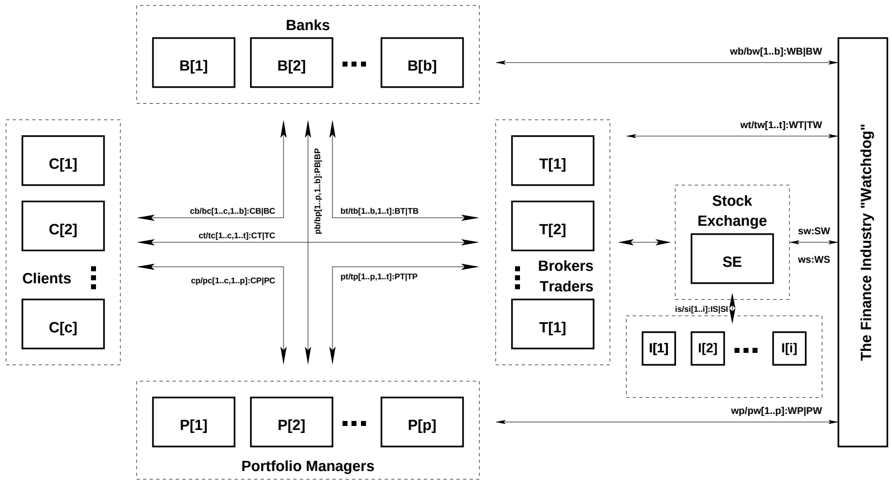


Fig. A.4. A financial behavioural system abstraction

A.2 Business Process Reengineering Requirements

Characterisation. By *business process reengineering* we understand the reformulation of previously adopted business process descriptions, together with additional business process engineering work. ▪

Business process reengineering (BPR) is about *change*, and hence BPR is also about *change management*. The concept of workflow is one of these "hyped" as well as "hijacked" terms: They sound good, and they make you "feel" good. But they are often applied to widely different subjects, albeit having some phenomena in common. By workflow we shall, very loosely, understand the physical movement of people, materials, information and "centre ('locus') of control" in some organisation (be it a factory, a hospital or other). We have, in Vol. 1, Chap. 12 (Petri Nets), in Sect. 12.5.1 covered the notion of *work flow systems*.

A.2.1 Michael Hammer's Ideas on BPR

Michael Hammer, a guru of the business process reengineering "movement", states [46]:

1. *Understand a method of reengineering before you do it for serious.*

So this is what this chapter is all about!

1. *One can only reengineer processes.*

Clearly Hammer utters an untenable dogma!

1. *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.

1. *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".

1. *Reengineering requires radical, breakthrough ideas about process design. Reengineering leaders must encourage people to pursue stretch goals*[1] *and to think out of the box; to this end, leadership must reward creative thinking and be willing to consider any new idea.*

[1] A 'stretch goal' is a goal, an objective, for which, if one wishes to achieve that goal, one has to stretch oneself.

This is clearly an example of the US guru, "new management"-type 'speak'!

1. *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".

1. *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.

1. *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".

1. *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. *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.

A.2.5 Intrinsics Review and Replacement

Characterisation. By *intrinsics review and replacement* we understand an evaluation as to whether current intrinsics stays or goes, and as to whether newer intrinsics need to be introduced. ∎

Example 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.

A.2.6 Support Technology Review and Replacement

Characterisation. By *support technology review and replacement* we understand an evaluation as to whether current support technology as used in the enterprise is adequate, and as to whether other (newer) support technology can better perform the desired services. ▪

Example 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. ▪

A.2.7 Management and Organisation Reengineering

Characterisation. By *management and organisation reengineering* we understand an evaluation as to whether current management principles and organisation structures as used in the enterprise are adequate, and as to whether other management principles and organisation structures can better monitor and control the enterprise. ▪

Example 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 designwise differently formatted paper documents: *requisition form, order form, purchase order, delivery inspection form, rejection and return form*, and *payment form.* The supplier had corresponding forms: *order acceptance and quotation form, delivery form, return acceptance form, invoice form, return verification form*, and *payment acceptance form.* The current concern is only the procurement forms, not the supplier forms. The proposed domain requirements are mandating that all procurer forms disappear in their paper version, that basically only one, the procurement document, represents all phases of procurement, and that order, rejection and return notification slips, and payment authorisation notes, be effected by electronically communicated and duly digitally signed messages that represent appropriate subparts of the one, now electronic procurement document. The business process reengineering part may now "short-circuit" previous staff's review and acceptance/rejection of former forms, in favour of fewer staff interventions.

The new business procedures, in this case, subsequently find their way into proper domain requirements: those that support, that is monitor and control all stages of the reengineered procurement process. ▪

A.2.8 Rules and Regulations Reengineering

Characterisation. By *rules and regulations reengineering* we understand an evaluation as to whether current rules and regulations as used in the enterprise are adequate, and as to whether other rules and regulations can better guide and regulate the enterprise. ▪

Here it should be remembered that rules and regulations principally stipulate business engineering processes. That is, they are — i.e., were — usually not computerised.

Example 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. ▪

A.2.9 Human Behaviour Reengineering

Characterisation. *Human Behaviour Reengineering:* By *human behaviour reengineering* we understand an evaluation as to whether current human behaviour as experienced in the enterprise is acceptable, and as to whether partially changed human behaviours are more suitable for the enterprise. ∎

Example 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. ∎

A.2.10 Script Reengineering

On one hand, there is the engineering of the contents of rules and regulations, and, on another hand, there are the people (management, staff) who script these rules and regulations, and the way in which these rules and regulations are communicated to managers and staff concerned.

Characterisation. By *script reengineering* we understand evaluation as to whether the way in which rules and regulations are scripted and made known (i.e., posted) to stakeholders in and of the enterprise is adequate, and as to whether other ways of scripting and posting are more suitable for the enterprise. ∎

Example 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 therefore called for, one that is really first to be done in the basic business processes of a bank offering these services to its customers. ∎

A.2.11 Discussion: Business Process Reengineering

A.2.12 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.

A.2.13 General

Business process reengineering is based on the premise that corporations must change their way of operating, and, hence, must "reinvent" themselves. Some corporations (enterprises, businesses, etc.) are "vertically" structured along functions, products or geographical regions. This often means that business processes "cut across" vertical units. Others are "horizontally" structured along coherent business processes. This often means that business processes "cut across" functions, products or geographical regions. In either case adjustments may need to be made as the business (i.e., products, sales, markets, etc.) changes. We otherwise refer to currently leading books on business process reengineering: [45, 46, 59, 63].

B

RSL: The RAISE Specification Language

B.1 Types

This is a very brief refresher on the `RAISE` Specification Language `RSL`. The reader is kindly asked to study first the decomposition of this section into its subparts and sub-subparts.

B.1.1 Type Expressions

`RSL` has a number of *built-in* types. There are the Booleans, integers, natural numbers, reals, characters and texts. From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc. Let A, B and C be any type names or type expressions, then the following (save the [i] line numbers) are generic type expressions:

Formal Expressions

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

[7] A-**set**
[8] A-**infset**
[9] $\mathrm{A} \times \mathrm{B} \times \ldots \times \mathrm{C}$
[10] A^*
[11] A^ω
[12] $\mathrm{A} \underset{m}{\overrightarrow{\;}} \mathrm{B}$
[13] $\mathrm{A} \to \mathrm{B}$

[14] $A \xrightarrow{\sim} B$
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

Annotations:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., $-2, -1, 0, 1, 2, \ldots$.
3. The natural number type of positive integer values $0, 1, 2, \ldots$.
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values "a", "b", ...[1]
6. The text type of character string values "aa", "aaa", ..., "abc", ...
7. The set type of finite set values, see below.
8. The set type of infinite set values.
9. The Cartesian type of Cartesian values, see below.
10. The list type of finite list values, see below.
11. The list type of infinite list values.
12. The map type of finite map values, see below.
13. The function type of total function values, see below.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf. 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., $(A \underset{m}{\rightarrow} B)$, or (A*)**-set**, or (A**-set**)list, or $(A|B) \underset{m}{\rightarrow} (C|D|(E \underset{m}{\rightarrow} F))$, etc.
16. The (postulated disjoint) union of types A, B, . . . , and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, . . . , and bv are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, . . . , and bv are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

[1] RSL uses double quotes " on both sides of a character and a character string rather than usual balanced quotes "..."

B.1.2 Type Definitions

Concrete Types:

Types can be concrete, in which case the structure of the type is specified by type expressions:

Formal Expressions

```
type
   A = Type_expr
```

Some schematic type definitions are:

Formal Expressions

```
[1]  Type_name = Type_expr /* without |s or sub-types */
[2]  Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3]  Type_name ==
          mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
          ... |
          mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4]  Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[5]  Type_name = {| v:Type_name′ • 𝒫(v) |}
```

where a form of [2–3] is provided by combining the types:

Formal Expressions

```
   Type_name = A | B | ... | Z
   A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
   B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
   ...
   Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Subtypes

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$ constitutes the subtype A:

Formal Expressions

```
type
   A = {| b:B • 𝒫(b) |}
```

Sorts (Abstract Types)

Types can be sorts (abstract) in which case their structure is not specified:

Formal Expressions

type
  A, B, ..., C

B.2 The RSL Predicate Calculus

B.2.1 Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values. Then

Formal Expressions

false, **true**
a, b, ..., c
$\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a=b, a$\neq$b

are propositional expressions having Boolean values. $\sim$, $\wedge$, $\vee$, $\Rightarrow$, and = are Boolean connectives (i.e., operators). They are read: *not*, *and*, *or*, *if-then* (or *implies*), *equal* and *not-equal*.

B.2.2 Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values, and let i, j, ..., k designate number values, then

Formal Expressions

false, **true**
a, b, ..., c
$\sim$a, a$\wedge$b, a$\vee$b, a$\Rightarrow$b, a=b, a$\neq$b
x=y, x$\neq$y,
i<j, i$\leq$j, i$\geq$j, i>j, ...

are simple predicate expressions.

B.2.3 Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then

Formal Expressions

$\forall$ x:X • $\mathcal{P}(x)$
$\exists$ y:Y • $\mathcal{Q}(y)$
$\exists$! 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.*

B.3 Concrete RSL Types

B.3.1 Set Enumerations

Let the below as denote values of type A, then the below designate simple set enumerations:

Formal Expressions

{{}, {a}, {a_1,a_2,...,a_m}, ...} $\in$ A-**set**
{{}, {a}, {a_1,a_2,...,a_m}, ..., {a_1,a_2,...}} $\in$ A-**infset**

The expression, last line below, to the right of the $\equiv$, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is highly abstract in the sense that it does not do so by following a concrete algorithm.

Formal Expressions

type
  A, B
  P = A $\rightarrow$ **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** $\times$ P $\times$ Q $\rightarrow$ B-**infset**
  comprehend(s,$\mathcal{P}$,$\mathcal{Q}$) $\equiv$ { $\mathcal{Q}$(a) | a:A • a $\in$ s $\wedge$ $\mathcal{P}$(a) }

B.3.2 Cartesian Enumerations

Let e range over values of Cartesian types involving A, B, …, C (allowing indexing for solving ambiguity), then the below expressions are simple Cartesian enumerations:

Formal Expressions

type
  A, B, ..., C
  A × B × ... × C
value
  ... (e1,e2,...,en) ...

B.3.3 List Enumerations

Let a range over values of type A (allowing indexing for solving ambiguity), then the below expressions are simple list enumerations:

Formal Expressions

  $\{\langle\rangle, \langle a\rangle, ..., \langle a1,a2,...,am\rangle, ...\} \in A^*$
  $\{\langle\rangle, \langle a\rangle, ..., \langle a1,a2,...,am\rangle, ..., \langle a1,a2,...,am,... \rangle, ...\} \in A^\omega$

  $\langle$ ei .. ej $\rangle$

The last line above assumes e_i and e_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former then the list is empty.

The last line below expresses list comprehension.

Formal Expressions

type
  A, B, P = A → **Bool**, Q = A $\stackrel{\sim}{\rightarrow}$ B
value
  comprehend: A^ω × P × Q $\stackrel{\sim}{\rightarrow}$ B^ω
  comprehend(lst,$\mathcal{P}$,$\mathcal{Q}$) ≡
    $\langle$ $\mathcal{Q}$(lst(i)) | i **in** $\langle$1..**len** lst$\rangle$ • $\mathcal{P}$(lst(i)) $\rangle$

B.3.4 Map Enumerations

Let a and b range over values of type A and B, respectively (allowing indexing for solving ambiguity).Then the below expressions are simple map enumerations:

Formal Expressions

type
  A, B
  M = A $\overrightarrow{m}$ B
value
  a,a1,a2,...,a3:A, b,b1,b2,...,b3:B

  [ ], [a↦b], ..., [a1↦b1,a2↦b2,...,a3↦b3] ∀ ∈ M

The last line below expresses map comprehension:

Formal Expressions

type
  A, B, C, D
  M = A $\overrightarrow{m}$ B
  F = A $\stackrel{\sim}{\to}$ C
  G = B $\stackrel{\sim}{\to}$ D
  P = A → **Bool**
value
  comprehend: M×F×G×P → (C $\overrightarrow{m}$ D)
  comprehend(m,$\mathcal{F}$,$\mathcal{G}$,$\mathcal{P}$) ≡
    [$\mathcal{F}$(a) ↦ $\mathcal{G}$(m(a)) | a:A • a ∈ **dom** m ∧ $\mathcal{P}$(a)]

B.3.5 Set Operations

Formal Expressions

value
  ∈: A × A-**infset** → **Bool**
  ∉: A × A-**infset** → **Bool**
  ∪: A-**infset** × A-**infset** → A-**infset**
  ∪: (A-**infset**)-**infset** → A-**infset**
  ∩: A-**infset** × A-**infset** → A-**infset**
  ∩: (A-**infset**)-**infset** → A-**infset**
  \: A-**infset** × A-**infset** → A-**infset**
  ⊂: A-**infset** × A-**infset** → **Bool**

$\subseteq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
$=$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
$\neq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
card: A-**infset** $\xrightarrow{\sim}$ **Nat**

examples
a $\in$ {a,b,c}
a $\notin$ {}, a $\notin$ {b,c}
{a,b,c} $\cup$ {a,b,d,e} = {a,b,c,d,e}
$\cup${{a},{a,b},{a,d}} = {a,b,d}
{a,b,c} $\cap$ {c,d,e} = {c}
$\cap${{a},{a,b},{a,d}} = {a}
{a,b,c} $\setminus$ {c,d} = {a,b}
{a,b} $\subset$ {a,b,c}
{a,b,c} $\subseteq$ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} $\neq$ {a,b}
card {} = 0, **card** {a,b,c} = 3

Annotations:

- $\in$ The membership operator expresses that an element is member of a set.
- $\notin$ The nonmembership operator expresses that an element is not member of a set.
- $\cup$ The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
- $\cap$ The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- $\setminus$ The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
- $\subseteq$ The proper subset operator expresses that all members of the left operand set are also in the right operand set.
- $\subset$ The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
- $=$ The equal operator expresses that the two operand sets are identical.
- $\neq$ The nonequal operator expresses that the two operand sets are *not* identical.
- **card** The cardinality operator gives the number of elements in a (finite) set.

The operations can be defined as follows:

Formal Expressions

value
$s' \cup s'' \equiv \{ a \mid a{:}A \bullet a \in s' \vee a \in s'' \}$
$s' \cap s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \in s'' \}$
$s' \setminus s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \notin s'' \}$
$s' \subseteq s'' \equiv \forall a{:}A \bullet a \in s' \Rightarrow a \in s''$
$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists a{:}A \bullet a \in s'' \wedge a \notin s'$
$s' = s'' \equiv \forall a{:}A \bullet a \in s' \equiv a \in s'' \equiv s \subseteq s' \wedge s' \subseteq s$
$s' \neq s'' \equiv s' \cap s'' \neq \{\}$
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 ∗/

B.3.6 Cartesian Operations

Formal Expressions

type
A, B, C
g0: G0 = A × B × C
g1: G1 = (A × B × C)
g2: G2 = (A × B) × C
g3: G3 = A × (B × C)

value
va:A, vb:B, vc:C, vd:D
(va,vb,vc):G0,
(va,vb,vc):G1
((va,vb),vc):G2
(va3,(vb3,vc3)):G3

decomposition expressions
let (a1,b1,c1) = g0,
    (a1′,b1′,c1′) = g1 **in** .. **end**
let ((a2,b2),c2) = g2 **in** .. **end**
let (a3,(b3,c3)) = g3 **in** .. **end**

B.3.7 List Operations

Formal Expressions

value

$\mathbf{hd}$: $A^\omega \xrightarrow{\sim} A$
$\mathbf{tl}$: $A^\omega \xrightarrow{\sim} A^\omega$
$\mathbf{len}$: $A^\omega \xrightarrow{\sim} \mathbf{Nat}$
$\mathbf{inds}$: $A^\omega \rightarrow \mathbf{Nat\text{-}infset}$
$\mathbf{elems}$: $A^\omega \rightarrow A\textbf{-infset}$
$.(.)$: $A^\omega \times \mathbf{Nat} \xrightarrow{\sim} A$
$\frown$: $A^* \times A^\omega \rightarrow A^\omega$
$=$: $A^\omega \times A^\omega \rightarrow \mathbf{Bool}$
$\neq$: $A^\omega \times A^\omega \rightarrow \mathbf{Bool}$

examples

$\mathbf{hd}\langle a1,a2,...,am\rangle=a1$
$\mathbf{tl}\langle a1,a2,...,am\rangle=\langle a2,...,am\rangle$
$\mathbf{len}\langle a1,a2,...,am\rangle=m$
$\mathbf{inds}\langle a1,a2,...,am\rangle=\{1,2,...,m\}$
$\mathbf{elems}\langle a1,a2,...,am\rangle=\{a1,a2,...,am\}$
$\langle a1,a2,...,am\rangle(i)=ai$
$\langle a,b,c\rangle^\frown\langle a,b,d\rangle = \langle a,b,c,a,b,d\rangle$
$\langle a,b,c\rangle=\langle a,b,c\rangle$
$\langle a,b,c\rangle \neq \langle a,b,d\rangle$

Annotations:

- **hd** Head gives the first element in a nonempty list.
- **tl** Tail gives the remaining list of a nonempty list when Head is removed.
- **len** Length gives the number of elements in a finite list.
- **inds** Indices gives the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems** Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$ Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$ Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$ The equal operator expresses that the two operand lists are identical.
- $\neq$ The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

Formal Expressions

value
is_finite_list: $A^\omega \rightarrow$ **Bool**

len q $\equiv$
case is_finite_list(q) **of**
true $\rightarrow$ **if** q = $\langle\rangle$ **then** 0 **else** 1 + **len tl** q **end**,
false $\rightarrow$ **chaos end**

inds q $\equiv$
case is_finite_list(q) **of**
true $\rightarrow$ { i | i:**Nat** • 1 $\leq$ i $\leq$ **len** q },
false $\rightarrow$ { i | i:**Nat** • i$\neq$0 } **end**

elems q $\equiv$ { q(i) | i:**Nat** • i $\in$ **inds** q }

q(i) $\equiv$
if i=1
then
if q$\neq\langle\rangle$
then let a:A,q$'$:Q • q=$\langle$a$\rangle$$^\frown$q$'$ **in** a **end**
else chaos end
else q(i$-$1) **end**

fq $^\frown$ iq $\equiv$
$\langle$ **if** 1 $\leq$ i $\leq$ **len** fq **then** fq(i) **else** iq(i $-$ **len** fq) **end**
| i:**Nat** • **if len** iq$\neq$**chaos then** i $\leq$ **len** fq+**len end** $\rangle$
pre is_finite_list(fq)

iq$'$ = iq$''$ $\equiv$
inds iq$'$ = **inds** iq$''$ $\wedge$ $\forall$ i:**Nat** • i $\in$ **inds** iq$'$ $\Rightarrow$ iq$'$(i) = iq$''$(i)

iq$'$ $\neq$ iq$''$ $\equiv$ $\sim$(iq$'$ = iq$''$)

B.3.8 Map Operations

Formal Expressions

value
m(a): M $\rightarrow$ A $\xrightarrow{\sim}$ B, m(a) = b

dom: M $\rightarrow$ A-**infset** [domain of map]
dom [a1$\mapsto$b1,a2$\mapsto$b2,...,an$\mapsto$bn] = {a1,a2,...,an}

rng: M $\rightarrow$ B-**infset** [range of map]
rng [a1$\mapsto$b1,a2$\mapsto$b2,...,an$\mapsto$bn] = {b1,b2,...,bn}

†: M $\times$ M $\rightarrow$ M [override extension]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″] † [a′$\mapsto$b″,a″$\mapsto$b′] = [a$\mapsto$b,a′$\mapsto$b″,a″$\mapsto$b′]

$\cup$: M $\times$ M $\rightarrow$ M [merge $\cup$]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″] $\cup$ [a‴$\mapsto$b‴] = [a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″,a‴$\mapsto$b‴]

\: M $\times$ A-**infset** $\rightarrow$ M [restriction by]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″]\{a} = [a′$\mapsto$b′,a″$\mapsto$b″]

/: M $\times$ A-**infset** $\rightarrow$ M [restriction to]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″]/{a′,a″} = [a′$\mapsto$b′,a″$\mapsto$b″]

=,≠: M $\times$ M $\rightarrow$ **Bool**

°: (A $\overrightarrow{m}$ B) $\times$ (B $\overrightarrow{m}$ C) $\rightarrow$ (A $\overrightarrow{m}$ C) [composition]
[a$\mapsto$b,a′$\mapsto$b′] ° [b$\mapsto$c,b′$\mapsto$c′,b″$\mapsto$c″] = [a$\mapsto$c,a′$\mapsto$c′]

Annotations:

- $m(a)$ Application gives the element of which a maps to in the map m.
- **dom** Domain/definition set gives the set of values which *maps to* in a map.
- **rng**: Range/image set gives the set of values which *are mapped to* in a map.
- † Override/extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- $\cup$ 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.
- ≠ The nonequal operator expresses that the two operand maps are *not* identical.

- ° Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

The map operations can also be defined as follows:

Formal Expressions

value
  rng m ≡ { m(a) | a:A • a ∈ **dom** m }

  m1 † m2 ≡
    [a↦b | a:A,b:B •
      a ∈ **dom** m1 \ **dom** m2 ∧ b=m1(a) ∨ a ∈ **dom** m2 ∧ b=m2(a)]

  m1 ∪ m2 ≡ [a↦b | a:A,b:B •
        a ∈ **dom** m1 ∧ b=m1(a) ∨ a ∈ **dom** m2 ∧ b=m2(a)]

  m \ s ≡ [a↦m(a) | a:A • a ∈ **dom** m \ s]
  m / s ≡ [a↦m(a) | a:A • a ∈ **dom** m ∩ s]

  m1 = m2 ≡
    dom m1 = **dom** m2 ∧ ∀ a:A • a ∈ **dom** m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [a↦c | a:A,c:C • a ∈ **dom** m ∧ c = n(m(a))]
    pre rng m ⊆ **dom** n

B.4 λ-Calculus and Functions

RSL supports function expressions for λ-abstraction.

B.4.1 The λ-Calculus Syntax

Formal Expressions

type /∗ A BNF Syntax: ∗/
  ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | (⟨A⟩)
  ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
  ⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩

$\langle A \rangle$ ::= ($\langle L \rangle\langle L \rangle$)
value /* Examples */
$\langle L \rangle$: e, f, a, ...
$\langle V \rangle$: x, ...
$\langle F \rangle$: λ x • e, ...
$\langle A \rangle$: f a, (f a), f(a), (f)(a), ...

B.4.2 Free and Bound Variables

Formal Expressions

Let x, y be variable names and e, f be λ-expressions.

- $\langle V \rangle$: Variable x is free in x.
- $\langle F \rangle$: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- $\langle A \rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

B.4.3 Substitution

In RSL, the following rules for substitution apply:

Formal Expressions

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,
  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x]($\lambda x \bullet P$)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ $\lambda y \bullet$ **subst**([N/x]P),
  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),
  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

B.4.4 α-Renaming and β-Reduction

Formal Expressions

- α-renaming: λx•M
  If x y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M): We can rename the formal parameter of a λ-function

expression provided that no free variables of its body M thereby become bound.

- β-reduction: (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)

B.4.5 Function Signatures

For some functions, we want to abstract from the function body:

Formal Expressions

value
obs_Pos_Aircraft: Aircraft $\rightarrow$ Pos,
move: Aircraft $\times$ Dir $\rightarrow$ Aircraft,

B.4.6 Function Definitions

Functions — with body — can be defined explicitly

Formal Expressions

value
f: A $\times$ B $\times$ C $\rightarrow$ D
f(a,b,c) $\equiv$ Value_Expr

g: B-**infset** $\times$ (D $\overrightarrow{m}$ C-**set**) $\stackrel{\sim}{\rightarrow}$ A^*
g(bs,dm) $\equiv$ Value_Expr
pre $\mathcal{P}$(dm)

or implicitly

Formal Expressions

value
f: A $\times$ B $\times$ C $\rightarrow$ D
f(a,b,c) **as** d
post $\mathcal{P}_1$(d)

g: B-**infset** $\times$ (D $\overrightarrow{m}$ C-**set**) $\stackrel{\sim}{\rightarrow}$ A^*
g(bs,dm) **as** al

pre $\mathcal{P}_2$(dm)
post $\mathcal{P}_3$(al)

The symbol $\stackrel{\sim}{\to}$ 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.

B.5 Further Applicative Expressions

B.5.1 Let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Formal Expressions

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of

Formal Expressions

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

Recursive **let** expressions are written as:

Formal Expressions

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **Y**F **in** B(f,a) **end**

where:

F $\equiv$ λg•λa•(E(g)) and YF = F(YF)

Predicative **let** expressions:

Formal Expressions

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}(\mathsf{a})$ for evaluation in the body $\mathcal{B}(\mathsf{a})$.

Patterns and *wild cards* can be used:

Formal Expressions

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩^ℓ = list **in** ... **end**
let ⟨a,_,b⟩^ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

B.5.2 Conditionals

Various kinds of conditional expressions are offered by RSL:

Formal Expressions

if b_expr **then** c_expr **else** a_expr **end**

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
    if b_expr **then** c_expr **else** **skip** **end**

if b_expr_1 **then** c_expr_1
elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...
elsif b_exprt_n **then** c_expr_n **end**

case expr **of**
    choice_pattern_1 → expr_1,
    choice_pattern_2 → expr_2,
    ...
    choice_pattern_n_or_wild_card → expr_n
end

B.5.3 Operator/Operand Expressions

Formal Expressions

$\langle$Expr$\rangle$ ::=
  $\langle$Prefix_Op$\rangle$ $\langle$Expr$\rangle$
  | $\langle$Expr$\rangle$ $\langle$Infix_Op$\rangle$ $\langle$Expr$\rangle$
  | $\langle$Expr$\rangle$ $\langle$Suffix_Op$\rangle$
  | ...
$\langle$Prefix_Op$\rangle$ ::=
  $-$ | $\sim$ | $\cup$ | $\cap$ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**
$\langle$Infix_Op$\rangle$ ::=
  $=$ | $\neq$ | $\equiv$ | $+$ | $-$ | $*$ | $\uparrow$ | $/$ | $<$ | $\leq$ | $\geq$ | $>$ | $\wedge$ | $\vee$ | $\Rightarrow$
  | $\in$ | $\notin$ | $\cup$ | $\cap$ | $\backslash$ | $\subset$ | $\subseteq$ | $\supseteq$ | $\supset$ | $\hat{}$ | $\dagger$ | $\circ$
$\langle$Suffix_Op$\rangle$ ::= !

B.6 Imperative Constructs

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

B.6.1 Variables and Assignment

Formal Expressions

0. **variable** v:Type := expression
1. v := expr

B.6.2 Statement Sequences and skip

Sequencing is done using the ";" operator. **skip** is the empty statement having no value or side effect.

Formal Expressions

2. **skip**
3. stm_1;stm_2;...;stm_n

B.6.3 Imperative Conditionals

Formal Expressions

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

B.6.4 Iterative Conditionals

Formal Expressions

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

B.6.5 Iterative Sequencing

Formal Expressions

8. **for** b **in** list_expr • P(b) **do** S(b) **end**

B.7 Process Constructs

B.7.1 Process Channels

Let A, B stand for types of channel messages and KIdx stand for channel array indexes. Then

Formal Expressions

channel c:A
channel { k[i]:B • i:KIdx }

declare a channel, c, and an array of channels, k, whose individual channels, k[i], are able to communicate values of the designated types.

B.7.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels.

Let P() and Q(i) stand for process expressions[2] then

Formal Expressions

P() ‖ Q(i)	Parallel composition
P() ⎕ Q(i)	Nondeterministic External Choice (either/or)
P() ⌈⌉ Q(i)	Nondeterministic Internal Choice (either/or)
P() ╫ Q()	Interlock Parallel composition

expresses the parallel (‖) of two processes, the nondeterministic choice between two processes, either external (⎕) or internal (⌈⌉). The interlock (╫) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

B.7.3 Input/Output Events

Let c and k[i] designate channels of type A, and let e designate an expression also of type A. Then

Formal Expressions

c ?, k[i] ?	Input expression (a clause)
c ! e, k[i] ! e	Output clause (a statement)

expresses the willingness to engage in an event that "reads" an input, and respectively "writes" an output.

B.7.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Formal Expressions

value

[2] Both expressions (P() and (Q(i)) name process definitions (P respectively Q). P has no formal parameters. Q has, as only parameter, a channel array index. The former, P(), thus invokes P with no arguments and the latter, Q(i), invokes Q with a channel array index argument.

P: **Unit** → **in** c **out** k[i] **Unit**
Q: i:KIdx → **out** c **in** k[i] **Unit**

P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

B.8 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, one or more values (including functions), zero, one or more variables, zero, one or more channels and zero, one or more axioms listed under respective **type, variable, channel, value**and **axiom** "headers". We prefer to list their order as shown:

Formal Expressions

type
 ...
variable
 ...
channel
 ...
value
 ...
axiom
 ...

In practice a full specification repeats the above listings many times, once for each "module" (i.e., aspect, facet, view) of specification. Each of these modules may be "wrapped" into scheme, class or object definitions.[3]

[3]For schemes, classes and objects we refer to Vol. 2, Chap. 10 of this series of textbooks.

C

On a Domain Model of Transportation

We bring a fragment of a domain model of transportation nets.

C.1 Net Topology

We conceptualise as segments the physically manifest phenomena of roads (between adjacent street intersections), rail tracks (between adjacent train stations), airlanes (between adjacent airports) and shpping lanes (between adjacent harbours). We likewise conceptualise as junctions street intersections, train stations, airports and harbours.

C.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 $\rightarrow$ S-**set**
  obs_Js: N $\rightarrow$ **J-set**
axiom
  $\forall$ n:N • **card** obs_Ss(n) $\geq 1 \wedge$ **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 $\rightarrow$ 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.

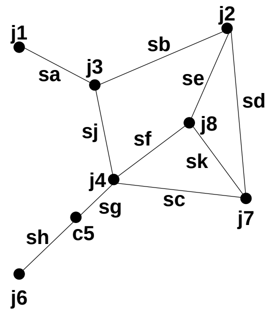


Fig. C.1. A simple net of segments and junctions

Applying the observer functions to the net of Fig. C.1 yields:

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.

C.1.2 Segment and Junction Identifications

1. 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.

[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, airlanes and shipping lanes, respectively the interesting (associated) street intersections, trains stations, airports and harbours.

2. 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.).

C.1.3 Segment and Junction Reference Identifications

1. 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.
2. 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 alon observe the connected segments. One can only refer to them. The identifications serve the role of being referents.

3. 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′.

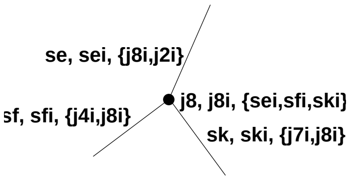


Fig. C.2. One junction and its connected segments

Figure C.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.

4. 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.

C.1.4 Paths and Routes

1. 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.
2. 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.

C.1.5 Segment and Junction Identifications of Routes

1. For future purposes we need be able to identifiy 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)}

$\text{xtr\_C: N} \times \text{Ji} \to \text{C, xtr\_S: N} \times \text{Si} \to \text{S}$
$\text{xtr\_C(n,ji)} \equiv \textbf{let}\ \text{j:J} \bullet \text{j} \in \text{obs\_Js(n)} \wedge \text{ji=obs\_Ji(j)}\ \textbf{in}\ \text{j}\ \textbf{end}$
$\text{xtr\_S(n,si)} \equiv \textbf{let}\ \text{s:S} \bullet \text{s} \in \text{obs\_Ss(n)} \wedge \text{si=obs\_Si(s)}\ \textbf{in}\ \text{s}\ \textbf{end}$

$\text{first\_Ji: R} \stackrel{\sim}{\to} \text{Ji, last\_Ji: R} \stackrel{\sim}{\to} \text{Ji}$
$\text{first\_Ji(r)} \equiv \textbf{case}\ \text{r}\ \textbf{of}\ \langle\rangle \to \textbf{chaos}, \langle(\text{ji},\_,\_)\rangle^\frown \text{r}' \to \text{ji}\ \textbf{end}$
$\text{last\_Ji(r)} \equiv \textbf{case}\ \text{r}\ \textbf{of}\ \langle\rangle \to \textbf{chaos}, \text{r}'^\frown\langle(\_,\_,\text{ji})\rangle \to \text{ji}\ \textbf{end}$

$\text{first\_Si: R} \stackrel{\sim}{\to} \text{Si, last\_Si: R} \stackrel{\sim}{\to} \text{Si}$
$\text{first\_Si(r)} \equiv \textbf{case}\ \text{r}\ \textbf{of}\ \langle\rangle \to \textbf{chaos}, \langle(\_,\text{si},\_)\rangle^\frown \text{r}' \to \text{si}\ \textbf{end}$
$\text{last\_Si(r)} \equiv \textbf{case}\ \text{r}\ \textbf{of}\ \langle\rangle \to \textbf{chaos}, \text{r}'^\frown\langle(\_,\text{si},\_)\rangle \to \text{si}\ \textbf{end}$

$\text{first\_J: R} \times \text{N} \stackrel{\sim}{\to} \text{J, last\_J: R} \times \text{N} \stackrel{\sim}{\to} \text{J}$
$\text{first\_J(r,n)} \equiv \text{xtr\_J(first\_Ji(r),n)}$
$\text{last\_J(r,n)} \equiv \text{xtr\_J(last\_Ji(r),n)}$

$\text{first\_S: R} \times \text{N} \stackrel{\sim}{\to} \text{S, last\_S: R} \times \text{N} \stackrel{\sim}{\to} \text{S}$
$\text{first\_S(r,n)} \equiv \text{xtr\_S(first\_Si(r),n)}$
$\text{last\_S(r,n)} \equiv \text{xtr\_S(last\_Si(r),n)}$

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. (Similarely 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.

C.1.6 Circular and Pendular Routes

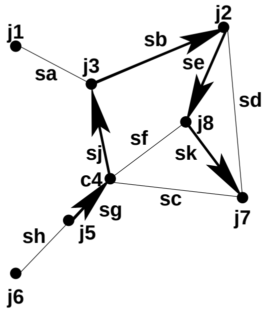
1. A route is circular if the same junction identification either occurs more than twice in the route, or if it occurs as both the first and the last junction identification of the route. Given a net we can compute the set of all non-circular routes by omitting from the above pairs of routes, r and r′, 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 contrained 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.
2. 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.

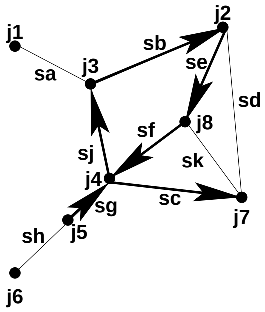
   value
       reverse: P → P
       reverse(jif,si,jit) ≡ (jit,si,jif)

       reverse: R → R



<(j5i,sgi,j4i),(j4i,sji,j3i),(j3i,sbi,j2i),(j2i,sei,j8i)>

Fig. C.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. C.4. A circular route, graphically and as an expression

reverse(r) $\equiv$
  case r **of**
    $\langle\rangle \to \langle\rangle$,
    $\langle(\text{jif,si,jit})\rangle^\frown r' \to \text{reverse}(r')^\frown\langle(\text{jit,si,jif})\rangle$
  end

reverse(r) $\equiv \langle \text{reverse}(r(i)) | i$ **in** $[\,n..1\,]\rangle$

pendular: R $\to$ R
pendular(r) $\equiv r^\frown\text{reverse}(r)$

is_pendular(r) $\equiv \exists\, r',r''\!:\!R \bullet r'^\frown r'' = r \wedge r''=\text{reverse}(r')$

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 constuct 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.

C.1.7 Connected Nets

1. 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′.

C.1.8 Net Decomposition

1. 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} ∧

$\{\} = \cap\{\text{obs\_Ss}(n')|n':N\bullet n' \in ns\} \wedge$
$\{\} = \cap\{\text{obs\_Js}(n')|n':N\bullet n' \in ns\} \wedge$
$\forall\ n':N\bullet n' \in ns \Rightarrow \text{connected}(n') \wedge \ldots$

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:

$\forall\ n:N\ \bullet$
  let ns = decompose (n) **in**
  $\forall\ n',n'':N \bullet \{n',n''\}\subseteq ns \wedge n'\neq n'' \Rightarrow$
    $\text{obs\_Ss}(n') \cap \text{obs\_Ss}(n'') = \{\} \wedge$
    $\text{obs\_Js}(n') \cap \text{obs\_Js}(n'') = \{\}$ **end**

The above 1 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.

C.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.

C.2.1 General Issues

1. 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

C.2.2 Segment and Junction Modes

1. 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 $\rightarrow$ M
   obs_Ms: J $\rightarrow$ M-**set**
   axiom
   $\forall$ n:N, j:J $\bullet$ j $\in$ obs_Js(n) $\Rightarrow$
   let ss = xtr_Ss(n,obs_Ji(j)) **in**
   obs_Ms(j) = {obs_M(s)|s:S $\bullet$ s $\in$ ss} **end**
   $\forall$ n:N, s:S $\bullet$ s $\in$ obs_Ss(n) $\Rightarrow$
   let {ji,ji′} = obs_Jis(s) **in**
   let {j,j′} = {xtr_J(n,ji),xtr_J(n,ji′)} **in**
   obs_M(s) $\in$ obs_Ms(j) $\cap$ 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.

C.2.3 Single-Modal Nets and Net Projection

1. Given a multi-modal net one can project it onto a set og 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 subsnet 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.

C.3 Segment and Junction Attributes

C.3.1 Segment and Junction Attribute Observations

We now enrich our segments and junctions.

1. Segments have lengths.
2. Junctions have modality-determined lengths between pairs of (same such modality) segments connected to the junction.
3. 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.
4. Junctions have standard transfer time per modality of transport between pairs of segments connected to the junction.
5. Junctions have standard arrival time per modality of transport.
6. Junctions have standard departure times per modality of transport.
7. Segments have standard costs of transporting a unit of freight from one end of the segment to the other end.
8. 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

$$
\begin{array}{ll}
\text{obs\_TI: S} \rightarrow \text{TI} & \\
\text{obs\_TI: Si} \times \text{J} \times \text{Si} \rightarrow \text{TI} & \\
\text{obs\_TI: J} \times \text{M} \xrightarrow{\sim} \text{TI}, & \textbf{pre } \text{obs\_TI(j,m): m} \in \text{obs\_Ms(j)} \\
\text{obs\_TI: J} \times \text{M} \times \text{M} \xrightarrow{\sim} \text{TI}, & \textbf{pre } \text{obs\_TI(j,m,m}'\text{): \{m,m}'\text{\}} \subseteq \text{obs\_Ms(j)} \\
\text{obs\_arr\_TI: J} \times \text{M} \xrightarrow{\sim} \text{TI}, & \textbf{pre } \text{obs\_arr\_TI(j,m): m} \in \text{obs\_Ms(j)} \\
\text{obs\_dep\_TI: J} \times \text{M} \xrightarrow{\sim} \text{TI}, & \textbf{pre } \text{obs\_dep\_TI(j,m): m} \in \text{obs\_Ms(j)} \\
+\text{: L} \times \text{L} \rightarrow \text{L} & \\
+\text{: TI} \times \text{TI} \rightarrow \text{TI} &
\end{array}
$$

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 lenths.
- One can add time durations.

C.3.2 Route Lengths

1. One can compute the length of a route of a net and one can find the shortest such route between two identified junctions.

   value
   $\quad$length: $\text{R} \rightarrow \text{N} \xrightarrow{\sim} \text{L}$
   $\quad$length(r)(n) $\equiv$
   $\quad\quad$**case** r **of**
   $\quad\quad\quad\langle\rangle \rightarrow 0$,
   $\quad\quad\quad\langle(\text{jf,si,jt})\rangle \rightarrow \text{obs\_L(xtr\_S(si,n))}$,

$\langle$(ji1,sii,ji2),(jj1,sij,jj2)$\rangle$⌢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($\langle$(jj1,sij,jj2)$\rangle$⌢r′) **end**
end
pre: r ∈ routes(n) ∧ ji2=jj1

value
shortest_route: Ji × Ji → N $\tilde{\to}$ 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 connnection 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.

C.3.3 Route Traversal Times

1. 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
```

C.3.4 Function Lifting

1. 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
```

2. Similarely 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.

C.3.5 Transporation Costs

1. 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)
```

C.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

[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 maing it into a bill of ladings notion: one that prescribes possible reloading at junctions.

of abstraction we consider certain road junctions to designate road nets of smaller or larger communities.

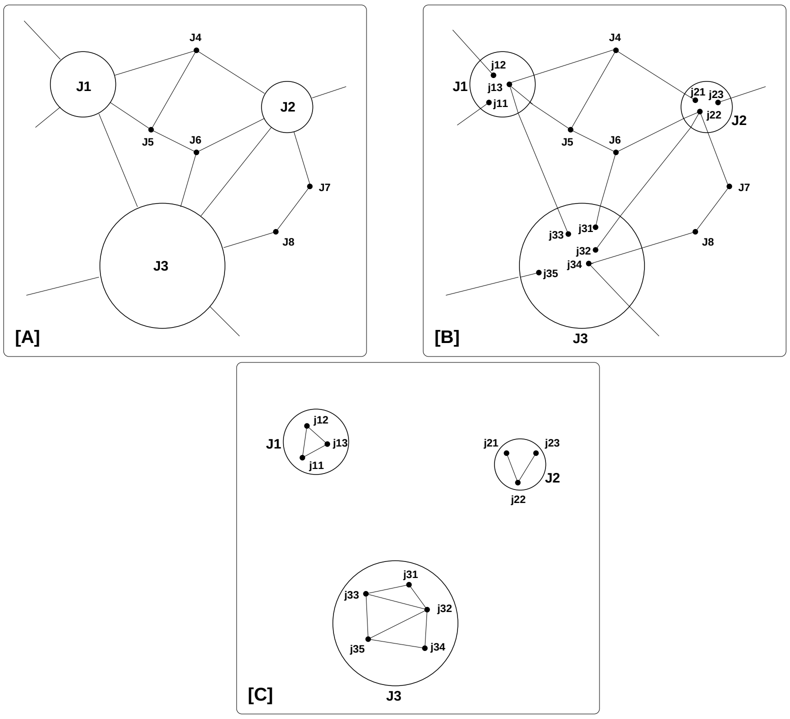


Fig. C.5. Gross [A] versus semi-detailed [B] road net — and community road nets [C]

1. Figure [A] C.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] C.5.
2. Junctions J1, J2 and J3 are considered composite, that is, to represent communities.
3. We may consider the road net of Fig.[A] C.5 to be an abstraction of the road net hinted at in Fig.[B] C.5.
4. Junctions j11, j12, ..., j35 are considered simple embedded junctions.
5. 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).
6. In Fig. [B] C.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.
7. The internal segments of junctions J1, J2 and J3 are shown in Fig. [C] C.5 on the previous page. They are to be considered complete nets "in and by" themselves.
8. We may consider the implied junction identifications Ji1, Ji2 and Ji3 to be names of communities.
9. 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.
10. We shall assume that from these junction identifications, say jikℓ, one can observe the more abstract junction identifications, i.e., Jik.
11. 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.
12. 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] C.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] C.5 on the previous 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
```

C.5 Railway Nets

C.5.1 General

A transportation net of modality railway has segments be lines between stations and have junctions be stations.

1. We concretise the concept of modes. Mode m=railway will now designate railway nets:

   ```
   type
       M == road | railway | ...
   ```

2. From a multi-modal transportation net we can project the railway net, rn:RN:

   ```
   value
       proj: N × {railway} → RN
   ```

3. Junctions of a transportation net of modality railway have sub-junctions which are stations:

   ```
   value
       proj: J × {railway} → ST
   ```

4. Segments of a transportation net of modality railway become lines:

value
proj: S × {railway} → LI

C.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.

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.

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

C.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 (udually different) segment emanating from that junction"*. Segments connected to a junction are both incident upon that junction and emanates from that junction.

C.6.1 Segment and Junction States

1. 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.
2. 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.
3. Junctions may direct traffic from any subset of incident segments to any subset of emanating segments.
4. 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
  ...

1. The junction of Fig. C.6 shows four segments, identified by A, B, C and D.
2. The figure also suggests a state in which traffic lights prohibit movements from A into J, from B into J,

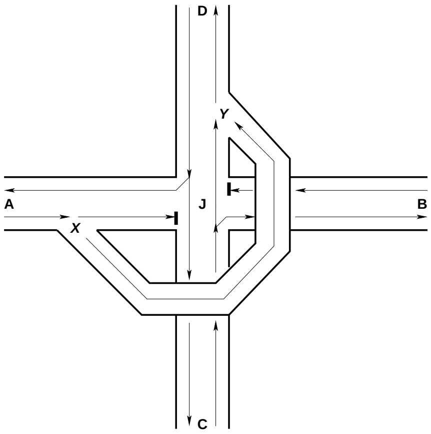


Fig. C.6. A Special "Carrefour" Junction

3. from C via J into A, and from D via J into B.
4. The "bypass" from A/X into Y/D appears to be such that traffic can always pass from A into D.
5. The current state alluded to in Fig. C.6 appears to be:

$$j\sigma_J : \{(A, D), (C, B), (C, D), (D, A), (D, C)\}$$

6. (A, D) is potentially a member of every state that the junction can possibly be in — see next section.

C.6.2 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)

Etcetera!

D

On a Domain Model of Manufacturing

We bring a fragment of a domain model of manufacturing.

D.1 Introduction

D.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).

D.1.2 Examples of Machines

The concept of machine in this book 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!

D.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. D.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], RAISE's RSL [39,41], VDM-SL [18,19,34] or Z [55,85,86,90] can skip the annotations. The formal notation that is used is RSL [39]. We refer to [11–13] for a thorough introduction to abstract and modelling using RSL.

D.2 Parts

1. An atomic part is a smallest unit of man-made production.

type
  P
value
  is_atomic_P: P $\rightarrow$ **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.

1. 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.

1. 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)

1. 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, $\pi : \mathit{\Pi}$.

type
  Π
  COMPS = P-**set**
value
  obs_*Π*: P → *Π*
  obs_COMPS: P → COMPS

  no_of_occurrences: P × P $\stackrel{\sim}{\to}$ **Nat**
  no_of_occurrences(cp,p) ≡
    card {p′|p′:P • p′ ∈ obs_COMPS(cp) ∧ 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_Π(p′) $\neq$ obs_Π(p″) $\wedge$
obs_Π(p) $\neq$ obs_Π(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 yeilds 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.

1. 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 comcepts.[1]

D.3 Machines

1. 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$\neq${} |}

[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).

  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.

1. 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 n = i(pn_j) then the machine function to which the machine signature will be associated (see below) shall require n occurrences of parts $p_1, p_2, \ldots, p_n$ — all having the same part number pn_j.

1. 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
  $\mathrm{MACH'} = \mathrm{Input} \times \mathrm{MSig} \times \mathrm{MOp} \times \mathrm{Output}$
  $\mathrm{MACH} = \{|\ \mathrm{mach{:}MACH'} \bullet \mathrm{wf\_MACH(mach)}\ |\}$
value
  $\mathrm{obs\_MACH{:}\ MC} \to \mathrm{MACH}$
  $\mathrm{wf\_MACH(i,(isig,osig),op,o)} \equiv$
    $\mathbf{dom}\ \mathrm{i} \subseteq \mathbf{dom}\ \mathrm{isig} \wedge \mathbf{dom}\ \mathrm{o} \subseteq \mathbf{dom}\ \mathrm{osig} \wedge$
    $\mathbf{dom}\ \mathrm{i} = \{\mathrm{pn} \mid \mathrm{pn{:}Pn} \bullet \exists\ \mathrm{p{:}P} \bullet \mathrm{p} \in \mathcal{D}(\mathrm{op}) \wedge \mathrm{pn=obs\_Pn(p)}\} \wedge$
    $\mathbf{dom}\ \mathrm{o} = \{\mathrm{pn} \mid \mathrm{pn{:}Pn} \bullet \exists\ \mathrm{p{:}P} \bullet \mathrm{p} \in \mathcal{R}(\mathrm{op}) \wedge \mathrm{pn=obs\_Pn(p)}\}$

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.
- 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 actal 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.

[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".

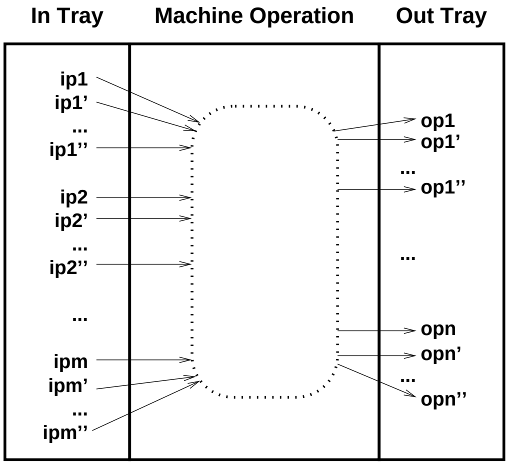


Fig. D.1. A schematic machine

1. Figure D.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.

D.4 Machine Operation

1. Figure D.1 also intends to show that a machine operation consumes one or more occurrences ($\mathsf{ip1}$, $\mathsf{ip1}'$, ..., $\mathsf{ip1}''$) of parts of one part number (ip_1), one or more occurrences ($\mathsf{ip2}$, $\mathsf{ip2}'$, ..., $\mathsf{ip2}''$) 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),
2. 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.

1. 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: $\text{isig}' = \text{isig} \wedge \text{osig}' = \text{osig} \wedge \text{op}' = \text{op} \wedge$
      $\text{input}' = \text{input} \setminus \mathcal{D}\,\text{op} \wedge \text{output}' = \text{output} \cup \mathcal{R}\,\text{op}$

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 reequired 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.

D.5 Production Floors

1. A production floor of a manufacturing plant, MP, consists of a non-zero number of uniquely identified machines.

type
  MP, PFId, MId
  $\text{PFs}' = \text{PFId} \overrightarrow{m} \text{PF}$, $\text{PFs} = \{|\text{pfs:PFs}' \bullet \text{pfs} \neq [\,]|\}$
  $\text{PF}' = \text{Mid} \overrightarrow{m} \text{MC}$, $\text{PF} = \{|\text{pf:PF}' \bullet \text{pf} \neq [\,]|\}$
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 idenfied 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.

1. We can think of a production floor as shown in Fig. D.2.

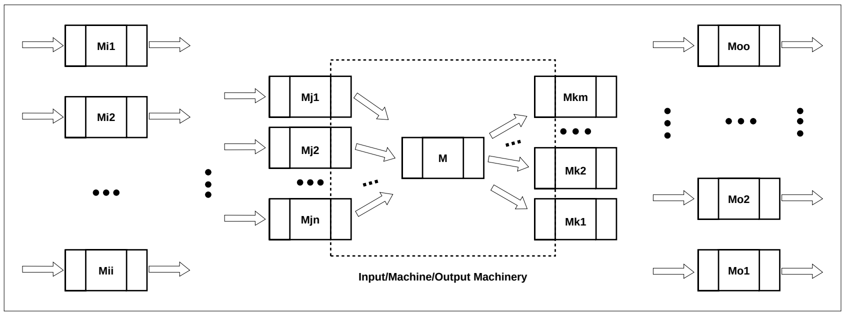


Fig. D.2. A schematic production floor

1. Let us focus on the entral — 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.
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.
2. Figure D.3 shows a more general situation for the input/machine/output machinery of Fig. D.2.

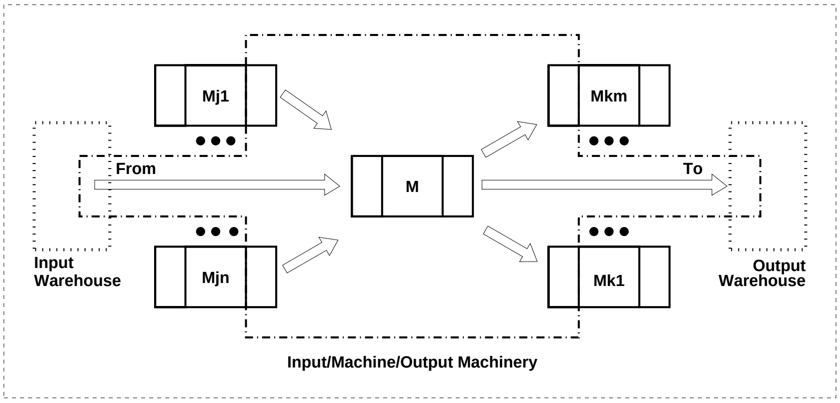


Fig. D.3. A general input/machine/output machinery

1. Whereas Fig. D.2 could be construed as expressing that all inputs to the central machine M came from other machines, Fig. D.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.
2. It is thus that we arrive at a production unit consisten 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 manufactutng 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.

D.6 Production Plans

D.6.1 Production Layouts

1. By a production layout we mean any arbitrary composition (i.e., set) of machines on a production floor, pf:PF.
2. The definition of PF given in item 1 is a model of a production layout.
3. Assume, as a thought experiment, that there is such a pf:PF. It consists on n machines: m_1, m_2, $\ldots$, m_n.
4. 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 4(a–4(c).

5. 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.
6. Instead we decide to model this aspect of production in the form of a production plan.

D.6.2 Production Targets

1. By a production target we mean the number of products of a finite number of distinct part numbers
2. A production target could be expressed, as intimated in Fig. D.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.
3. 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.
4. 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 $(n_{M_i} \times m_{M_i} : p)$, $(n_{M_i} \times m'_{M_i} : p')$, ..., $(n_{M_i} \times m''_{M_i} : p'')$.
5. And so on.

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.

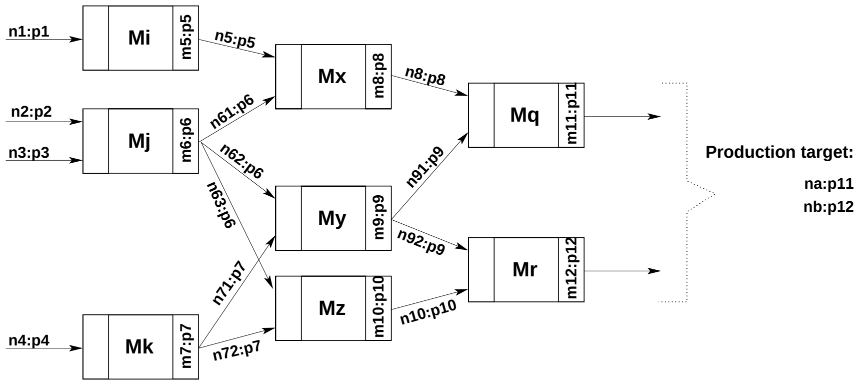


Fig. D.4. A production layout

D.6.3 Part Dependencies

1. Figure D.4 instantiates a rather general layout. Same kind (i.e., part number) input parts may come from different machines, etc.
2. 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?
3. 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).
4. Let us examine Fig. D.3 on page 128 and Fig. D.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. D.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 $m'^{\cdots\prime}$ which provides parts to machine m.
5. To express this formally, and thus precisely, we simplify some aspects of machines: The abstract from machine signatues 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
  $\mathrm{PFs}' = \mathrm{PFId} \xrightarrow[m]{} \mathrm{PF}$, $\mathrm{PFs} = \{|\mathrm{pfs}{:}\mathrm{PFs}' \bullet \mathrm{pfs} \neq [\,]|\}$

$\mathrm{PF}' = \mathrm{Mid} \underset{m}{\overrightarrow{\ }} \mathrm{MC}$, 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

D.6.4 Production Plans

Etcetera.

D.7 Interpretations of the Model — So Far!

We present a few examples of “near”-realistic production facilities.

D.7.1 A Matchbox Factory

1. Figure D.5 intends to show the production floor of a simple minded match factory.
2. Descriptions of machines are onky indicative of what “goes on”!
3. The reader may wish to make these descriptions more complete.

1. Leftmost and rightmost “dangling” arrows designate input from the in warehouse, respectively output to the out warehouse.
2. The reader may ponder about such questions as:
   (a) How is the output production capacity of machine Mt “tuned” to the input production capacty of machine Ms,
   (b) How is the output production capacity of machine Ms “tuned” to the input production capacty of machine Mmb,
   (c) How is the output production capacity of machine Mb “tuned” to the input production capacty of machine Mmb and

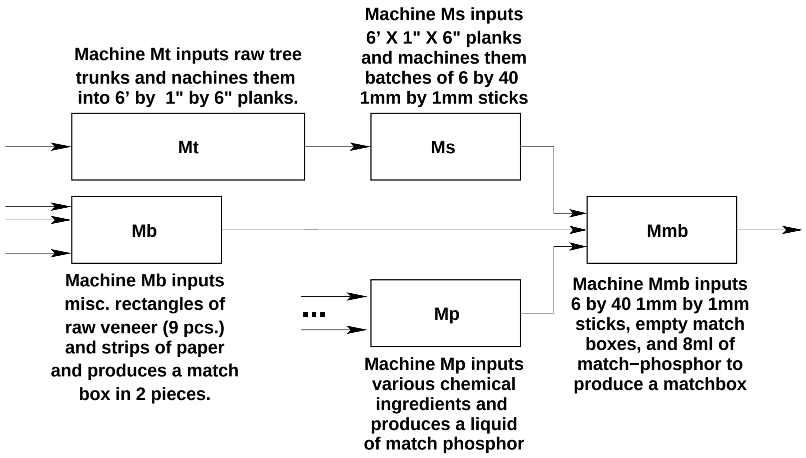


Fig. D.5. A match factory

   (d) How is the output production capacity of machine Mp “tuned” to the input production capacty of machine Mmb.
3. So do we!
4. Please not that this is just one interpretation of the concept of machines, warehouses and production floors.
5. For more realistics 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

D.7.2 A Hot Strip Mill

For an intuition of hot strip tubular and sheet production, please see:
http://www.ussteel.com/corp/sheet/hr/pmcpline.htm
http://www.wcisteel.com/operations/main.html

D.7.3 Cog Wheel Factory

For an intuition of cog wheel production, please see:http://www.hero.dk/engelsk/
Click successive images in left quadrangle

E

On a Domain Model of Documents

We bring a fragment of a domain model of documents.

E.1 A RAISE Model of Documents

E.1.1 Originals, Copies and Versions

There are documents. Documents are either created, edited or copied. One can claim that a document is either an original, or an edited version, for short, a version, of a document, or a copy of a document. One can claim that a document can either only (say: "most recently") be an original, or only (say: "most recently") be an edited document (i.e., a version), or only (say: "most recently") be a copy of a document. The pragmatic intention of documents is to embody document content. We leave the notion of document content undefined. There is information. Information is either document content, or the absence of such. We use the special literal void to designate absence of content. 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: D → Bool
      is_eD: D → Bool
      is_cD: D → Bool
    axiom
```

```
  ∀ d:D •
     is_oD(d)∨is_eD(d)∨is_cD(d) ∧
     is_oD(d)⇒∼is_eD(d)∧∼is_cD(d) ∧
     is_eD(d)⇒∼is_oD(d)∧∼is_cD(d) ∧
     is_cD(d)⇒∼is_oD(d)∧∼is_eD(d) ∧ ... or, which is the same:
  ∀ d:D,e:E •
     is_oD(create()) ∧ ∼is_eD(create()) ∧ ∼is_cD(create()) ∧
     is_cD(copy(d)) ∧ ∼is_oD(copy(d)) ∧ ∼is_cD(edit(e,d))
     is_eD(edit(e,d)) ∧ ∼is_oD(edit(e,d)) ∧ ∼is_cD(edit(e,d))
value
  obs_I: D → I
axiom
  obs_I(create()) = void ∧
  ∀ d:D • is_oD(d) ⇒ 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. E.1.2 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 (... → ...) that they are all function values.
  - ⋆ create designates the create function. It is a type **Unit** → 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 × D → 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 → 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".[1]
  - ⋆ 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.

[1] Adding 3 and 7, yielding 10, does not, in any way, destroy or influence 3 and 7.

  - ⋆ is_eD designates a predicate observer function. It is a type $D \to \textbf{Bool}$. 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 $D \to \textbf{Bool}$. 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 $\forall$ 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 ... .*
  - ⋆ The proposition *is_oD(create()) ∧ ∼is_eD(create()) ∧ ∼is_cD(create())* "reads" *for all documents and editing values, the value resulting from a proper create operation is an original and is not a edited version and is not a copy.*
  - ⋆ The predicate ∀ d:D,e:E • is_eD(edit(e,d)) ∧ ∼is_oD(edit(e,d)) etcetera "reads" *a document that has been properly edited is an edited version and is not an original and is not a copy.*
- The signature obs_I: $D \to I$ 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.

E.1.2 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
$\quad E' = FE \times BE$

FE,BE = I → I
E = {|(fe,be):E • ∀ i:I • be(fe(i))=i |}
value
obs_E: D $\stackrel{\sim}{\to}$ 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.

E.1.3 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 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 $\stackrel{\sim}{\to}$ 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 book.

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 ith 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: $D \rightarrow D^*$
  obs_doc_trace(d) $\equiv$
    if is_oD(d) **then** $\langle d \rangle$ **else** $\langle d \rangle$^obs_doc_trace(obs_Pre(d)) **end**

Annotations:

- We name the document trace function obs_doc_trace since it is 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 $\wedge$ ....

axiom
  $\forall$ d:D •

$\forall$ i:**Nat** • i $\in$ **inds** obs_doc_trace(d) $\Rightarrow$
  is_eD(obs_doc_trace(d)(i)) $\Rightarrow$
    $\forall$ j:**Nat** • j $\in$ **inds** obs_doc_trace(d) $\wedge$ j<i $\wedge$
      $\forall$ k:**Nat** • k $\in$ {j,i−1} $\wedge$ is_cD(obs_doc_trace(d)(k)) $\Rightarrow$
        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 doucments
- 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.

E.1.4 Annotated Original Documents

We modify the copy function and the notion of an original document, od:oD. We now annotate original document by a trace of "has been copied" markers. The 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 $\rightarrow$ hbc_Mark*
axiom
  obs_hbc_Marks(create()) $\equiv$ $\langle\rangle$ $\wedge$
  $\forall$ od:oD • obs_hbc_Marks(copy(od)) = $\langle$hbc$\rangle$$^\frown$obs_hbc_Marks(od)
value
  disregard_Marks: D $\rightarrow$ D
  disregard_Marks(d) **as** d′
    obs_hbc_Marks(d′) = $\langle\rangle$ $\wedge$ obs_Pre(d) = obs_Pre(d′)
  differ_by_1_Mark: D $\times$ D $\rightarrow$ **Bool**
  differ_by_1_Mark(d,d′) $\equiv$
    obs_hbc_Marks(d) = **tl** obs_hbc_Marks(d′) $\wedge$
    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 "bass been copied" marks. That is the function of the disregard_Marks function.
- Two documents are, in a sense, the same if they differ only by one or more marks.

We now redefine the predecessor observer function.

value
  obs_Pre: D $\stackrel{\sim}{\to}$ D
axiom
  obs_Pre(copy(d)) = d $\wedge$
  $\forall$ d:D,e:E $\bullet$
    obs_Pre(edit(e,d)) = d $\wedge$
  $\forall$ od:oD $\bullet$
    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.

E.1.5 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 the number of documents remain the same). Given an original document one can establish the family tree of documents descending from the originally created document.
A document family tree consists of nodes and stems (i.e., branches). Nodes, other than the root node, designate operations performed on documents. The root node designates the "moment" before "creation"! Stems designate documents. 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 new document resulting from the node oeration. The output stem is, or the output stems are said to emanate from the node. The root node designates the create operation. Any other node designates either an edit or a copy operation. If a node designates an edit operation then it has one output stem and that stem designates the edited version of the document designated by the stem incident upon the edit node. If a node designates a copy operation then

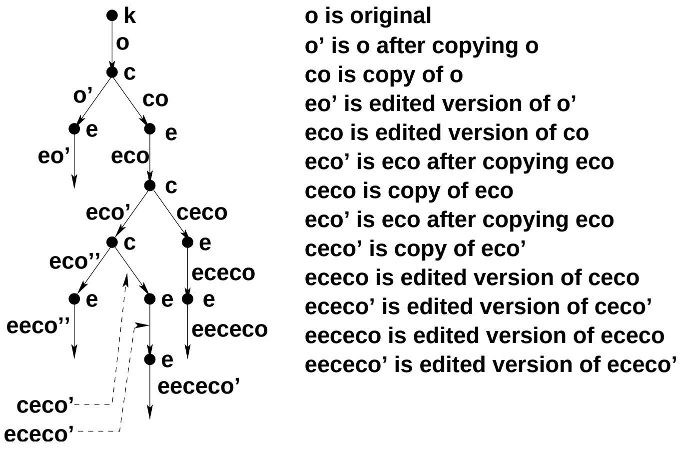


Fig. E.1. A document family tree

it has two output stems: one of these stems designate the (input) 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 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 sub-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[2] may have one or two subtrees, i.e., document trees: one of the (sub-)root designates the [create] (edit) [3] 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) ≡

[2]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.*

[3]The phrase: *(sub-)root designates the [create] (edit)* reads as follows: *root designates the create or sub-root designates the edit.*

```
    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′))
            → preEpost((fe,be),d,ed) ∧ wfDT(dt′)(ed),
        mkCT(mkCopy(),(d′,dt′),(cd,dt″))
            → preCpost(d,d′) ∧ wfDT(dt′)(d′) ∧ wfDT(dt″)(cd)
    end

preEpost:  E × D × eD → Bool
preEpost((fe,be),d,ed) ≡ ...
    /∗ see postcondition of the edit function on page 138 ∗/

preCpost: D × D → Bool
preCpost(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 amanating from the mkCreate() node.
- DT designates the possibly emty 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)) snd 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 likewsie 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 preEpost((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 preCpost(d,d′) where d′ is the document being copied — and after copying,
    - · the well-formedness of the master[4] document sub-tree wfDT(dt′)(d′), and
    - · the well-formedness of the copied document sub-tree wfDT(dt″)(cd).

E.1.6 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. E.2 shows 11 states of a document family tree.

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 state, here s10, designates a notion of final documents. A final state means that no further operations are to be performed on members of a set of documents. ("Case closed.") Please note that the final state of any document family tree is unique — as is the initial state. Please also note that a void document, i.e., a copy of a copy of … a

[4]We shall move this notion way back, towards the front of this book: the master document is the document being copied.

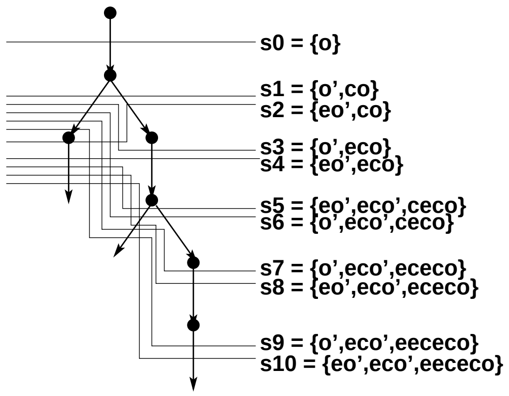


Fig. E.2. Document family states

copy of an original document may be a final document.[5] 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
  $\Sigma = \{|\sigma{:}\mathrm{D}\textbf{-set}{\bullet}\sigma{\neq}\{\}|\}$
value
  States: DFT $\to$ Σ**-set**
  Traversal: DFT $\to$ Σ^*

  States(dft) $\equiv$ ...
  Traversal(dft) $\equiv$ ...

E.1.7 Document Community

By a document community we mean a set of uniquely identified document family trees.

[5]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.

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 → **Bool**
  wfDoCo(doco) ≡ ...

E.1.8 Discussion of First Model of Document Intrinsics

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. 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, in the next two sections propose concrete models of locations and time of operation invocation.

E.1.9 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.

We now extend our simplistic model of document intrinsics. 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
  create: L → oD
  copy: D × L → D × cD
  edit: E × D → eD
  move: D × L $\xrightarrow{\sim}$ D, **pre**: move(d,l): obs_L(d)≠l
  obs_L: D → L
axiom
  ∀ l:L,e:E •
    obs_L(create(l)) = l ∧
    let (d′,cd) = copy(d,l) **in**
    preCpost′(d,d′) ∧ obs_L(cd)=l **end** ∧
    let ed = edit(e,d) **in** obs_L(ed)=obs_L(d) **end** ∧
    obs_L(move(d,l)) = l

E.1.10 A Basic Concrete Model of Time

We introduce a temporal notion of time. 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).

We now extend our simplistic model of document intrinsics. 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. The original document being created "receives" the given time. The document copy being established likewise "receives" the given time. The document from which the copy was made retains its time. The document resulting from an edit "receives" the given time. The move shall result in a moved document 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. Previous documents of any documents retain their times of operation applications.

type
  T
value
  obs_T: $D \to T$
  create: $L \times T \to oD$
  copy: $D \times L \times T \to D \times cD$
  edit: $E \times D \times T \to eD$
  move: $D \times L \times T \xrightarrow{\sim} D$, **pre**: move(d,l): obs_L(d)$\neq$l
axiom
  $\forall$ t:T,e:E •
    obs_T(create(_,t)) = t $\wedge$
    let (d′,cd) = copy(d,_,t) **in**
    preCpost′(d,d′) $\wedge$ obs_T(cd)=t **end** $\wedge$
    let ed = edit(e,d,t) **in** obs_T(ed)=t **end** $\wedge$
    obs_T(move(d,l,t)) = t

E.1.11 Located and Timed Documents

We wish to record that for every document that has been copied the fact that it has been copied: tie and place.

value
  has_been_copied: $D \to$ **Bool**
  when_where_copied: $D \xrightarrow{\sim} L \times T$
  otherwise_the_same: $D \times D \to$ **Bool**
axiom
  $\forall$ d:D •
    $\sim$has_been_copied(d) $\equiv$ when_where_copied(d)=**chaos** $\wedge$
    $\forall$ l:L,t:T •
      let (c′,cd) = copy(d,l,t) **in**
      has_been_copied(d′) $\wedge$ when_where_copied(d′)=(l,t) $\wedge$
      otherwise_the_same(d,d′) **end**

Time and Space (Locations)

No document can at the same time be in two different locations:

axiom
  $\forall$ d,d′:D •
    let (l,t) = (obs_L(d),obs_T(d)),(l′,t′) = (obs_L(d′),obs_T(d′)) **in**
    (t=t′ $\wedge$ l=l′ $\equiv$ d=d′) $\wedge$ (t=t′ $\wedge$ l$\neq$l′ $\equiv$ d$\neq$d′) **end**

E.1.12 Unique Document Identifiers

Given that creation and copying times and locations are unique we can introduce a notion of unique document identifications. For a document family that is simple enough:

type
  Uid
value
  obs_Uid: D → Uid
axiom
  let od = create() **in**
  let uid = obs_Uid(d) **in**
  ∀ d:D, e:E, t:T, l:L •
    let dt = doc_trace(d) **in** dt(**len** dt)=od ⇒
      ∀ i:**Nat** • i ∈ **inds** dt ⇒
        is_oD(dt(i)) ⇒ obs_Uid(dt(i))=uid ∧
        is_eD(dt(i)) ⇒ obs_Uid(dt(i))≠uid ∧
        is_cD(dt(i)) ⇒ obs_Uid(dt(i))≠uid
  end end end

For a community we need to distnguish two originals by their distinct document family identifiers. That is: The Uid is some concoction of a document family identifier, a ocopy or edit operation time and a copy or edit operation location.

E.2 A CafeOBJ Model of Documents

This model and its analysis (inclunding proofs) has been worked out by Kazuhiro Ogata.

```
--
-- Sort Absence corresponds to the type whose values are only void.
-- Constant void corresponds to value void.
--
mod! ABSENCE {
  [Absence]
  op void : -> Absence .
  op _=_ : Absence Absence -> Bool {comm}
  var A : Absence
  eq (A = A) = true .
}

--
-- Sort Content corresponds to type C.
--
mod* CONTENT {
  [Content]
  op _=_ : Content Content -> Bool {comm}
  var C : Content
  eq (C = C) = true .
}
```

```
--
-- Sort Info is declared as a supersort of Void and Content.
-- The sort corresponds to type I == void | C.
--
mod* INFO {
  pr(ABSENCE + CONTENT)
  [Absence Content < Info]
  op _=_ : Info Info -> Bool {comm}
  op _=_ : Absence Content -> Bool
  op _=_ : Content Absence -> Bool
  var I : Info
  var A : Absence
  var C : Content
  eq (I = I) = true .
  eq (A = C) = false .
  eq (C = A) = false .
}
--
-- Visible sort Editing corresponds to type E.
--
mod* EDITING {
  [Editing]
  op _=_ : Editing Editing -> Bool {comm}
  var E : Editing
  eq (E = E) = true .
}
--
-- Sort Document is declared as a supersort of sorts ODoc, EDoc and CDoc.
-- Sort Document corresponds to type D.
-- Sort ODoc corresponds to type oD.
-- Sort EDoc corresponds to type eD.
-- Sort CDoc corresponds to type oD.
--
mod! DOCUMENT {
  pr(INFO + EDITING)
  [ODoc EDoc CDoc < Document]
  op create : -> ODoc
  op edit : Editing Document -> EDoc
  op copy : Document -> CDoc
  op isODoc : Document -> Bool -- corresponds to is_oD
  op isEDoc : Document -> Bool -- corresponds to is_eD
  op isCDoc : Document -> Bool -- corresponds to is_cD
  op obsInfo : Document -> Info -- corresponds to obs_I
  op _=_ : Document Document -> Bool {comm}
  vars D D1 : Document
  var OD : ODoc
  var ED : EDoc
  var CD : CDoc
  vars E E1 : Editing
  --
  eq isODoc(OD) = true .
  eq isODoc(ED) = false .
  eq isODoc(CD) = false .
  eq isEDoc(OD) = false .
  eq isEDoc(ED) = true .
  eq isEDoc(CD) = false .
  eq isCDoc(OD) = false .
  eq isCDoc(ED) = false .
  eq isCDoc(CD) = true .
  --
  -- The following four equations surely hold because of the definitiong.
  -- A term of sort Document is one of create, edit(e,d) or cpoy(d),
  -- where e is a term of sort Editing and d is a temr of sort Document.
  --
  eq isODoc(D) or isEDoc(D) or isCDoc(D) = true .
  ceq (not isEDoc(D)) and (not isCDoc(D)) = true  if isODoc(D) .
```

```
  ceq (not isODoc(D)) and (not isCDoc(D)) = true  if isEDoc(D) .
  ceq (not isODoc(D)) and (not isEDoc(D)) = true  if isCDoc(D) .
  --
  eq obsInfo(create) = void .
  ceq obsInfo(copy(D)) = void if isODoc(D) .
  --
  eq (D = D) = true .
  eq (create = edit(E,D)) = false .
  eq (create = copy(D)) = false .
  eq (edit(E,D) = copy(D1)) = false .
  eq (edit(E,D) = edit(E1,D1)) = (E = E1) and (D = D1) .
  eq (copy(D) = copy(D1)) = (D = D1) .
}

--
-- Since CafeOBJ does not provide any facilities that deal with higher-order
-- functions directly, we have to come up with something, which can simulate
-- higher-order functions.
--

--
-- This module is used to constrain paramters of the coming parameterized modules.
--
mod* EQTRIV {
  [Elt]
  op _=_ : Elt Elt -> Bool {comm}
}

--
-- This module deals with higher-order functions to some extent.
-- The module has two paramters named D and R, each of which is constrained
-- by module EQTRIV.
--
mod* FUNCTION(D :: EQTRIV, R :: EQTRIV) {
  [Fun]
  op app : Fun Elt.D -> Elt.R
  op _=_ : Fun Fun -> Bool {comm}
  vars F F1 F2 : Fun
  vars D1 D2 : Elt.D
  eq (F = F) = true .
  ceq (app(F1,D1) = app(F2,D2)) = true if (F1 = F2) and (D1 = D2) .
}

--
-- This view can be used to instantiate module FUNCTION with module INFO
-- that is used as actual parameters.
--
view VIEW1 from EQTRIV to INFO {
  sort Elt -> Info,
  op _=_ -> _=_
}

--
-- An editing is represented by a pair of two functions such that
-- the two functions satisfy equation (*1).
-- Sort Chaos is declared as a supersort of sort Editing.
-- A term of sort Chaos is either chaos or <fe,be>, where fe and be are
-- terms of sort Fun.
-- A partial function f : Document ~> Editing can be dealt with as
-- a total function f' : Document -> Chaos such that f'(d) is f(d) if f is
-- defined on d, and f'(d) is chaos otherwise.
--
mod! EXTENDED-EDITING {
  ex(EDITING)
  pr(FUNCTION(D <= VIEW1, R <= VIEW1)) -- instantiating
  [Editing < Chaos]
  op <_,_> : Fun Fun -> Editing
```

```
  ops fe be : Editing -> Fun
  op chaos : -> Chaos
  op _=_ : Chaos Chaos -> Bool {comm}
  op _=_ : Editing Chaos -> Bool
  op _=_ : Chaos Editing -> Bool
  vars F1 F2 F11 F21 : Fun
  var E : Editing
  var I : Info
  var C : Chaos
  eq fe(< F1, F2 >) = F1 .
  eq be(< F1, F2 >) = F2 .
  eq (< F1, F2 > = < F11, F21 >) = (F1 = F11) and (F2 = F21) .
  --
  eq app(be(E),app(fe(E),I)) = I . -- (*1)
  --
  -- In addition to equation (*1), some more equations, which are
  -- equivalent to it, may have to be added so that rewriring can be used to
  -- reason about documents.
  --
  eq (chaos = chaos) = true .
  eq (E = chaos) = false .
  eq (chaos = E) = false .
}

--
-- Sort Chaos2 is declared as a supersort of sort Document.
--
mod! EXTENDED-DOCUMENT {
  pr(EXTENDED-EDITING)
  ex(DOCUMENT)
  [Document < Chaos2]
  op obsEdit : Document -> Editing
  op obsPre : Document -> Document
  op chaos2 : -> Chaos2
  op _=_ : Chaos2 Chaos2 -> Bool {comm}
  op _=_ : Document Chaos2 -> Bool
  op _=_ : Chaos2 Document -> Bool
  var E : Editing
  var D : Document
  var C : Chaos2
  eq obsEdit(create) = chaos .
  eq obsEdit(edit(E,D)) = E .
  --
  -- eq obsEdit(copy(edit(E,D))) = E .
  -- -- This equation is immplied by the following one.
  --
  eq obsEdit(copy(D)) = obsEdit(D) .
  --
  eq obsInfo(edit(E,D)) = app(fe(E),obsInfo(D)) .
  --
  -- eq  obsInfo(D) = app(be(E),obsInfo(edit(E,D))) .
  -- -- This equation cannot be used as a left-to-right rewrite rules
  -- -- b/c variable E, which appears on the right-hand side, does not
  -- -- appear on the left-hand side.
  -- -- So, instead of it, the following one is declared.
  --
  eq app(be(E),obsInfo(edit(E,D))) = obsInfo(D) .
  --
  eq obsPre(create) = chaos2 .
  eq obsPre(edit(E,D)) = D .
  eq obsPre(copy(D)) = D .
  --
  eq (chaos2 = chaos2) = true .
  eq (D = chaos2) = false .
  eq (chaos2 = D) = false .
}
```

```
--
-- Sort Nat denotes natural numbers.
--
mod! PNAT {
  [Nat]
  op 0 : -> Nat
  op s : Nat -> Nat
  op p : Nat -> Nat
  op _=_ : Nat Nat -> Bool {comm}
  op _<_ : Nat Nat -> Bool
  op _=<_ : Nat Nat -> Bool
  vars N N1 : Nat
  eq p(s(N)) = N .
  eq (N = N) = true .
  eq (0 = s(N)) = false .
  eq (s(N) = s(N1)) = (N = N1) .
  eq (N < N) = false .
  eq (0 < s(N)) = true .
  eq (N < 0) = false .
  eq (s(N) < s(N1)) = (N < N1) .
  eq (N =< N) = true .
  eq (0 =< N) = true .
  eq (s(N) =< 0) = false .
  eq (s(N) =< s(N1)) = (N =< N1) .
}

--
-- Sort List denotes generic lists.
--
mod! LIST(D :: EQTRIV) {
  pr(PNAT)
  [List]
  op nil : -> List
  op _,_ : Elt.D List -> List
  op hd : List -> Elt.D
  op tl : List -> List
  op len : List -> Nat
  op nth : List Nat -> Elt.D
  op _=_ : List List -> Bool {comm}
  vars L L1 : List
  vars E E1 : Elt.D
  var N : Nat
  eq (L = L) = true .
  eq ((E, L) = (E1, L1)) = (E = E1) and (L = L1) .
  eq len(nil) = 0 .
  eq len((E, L)) = s(len(L)) .
  eq nth((E, L),0) = E .
  eq nth((E, L),s(N)) = nth(L,N) .
}

--
-- This view can be used to instantiate LIST with EXTENDED-DOCUMENT.
--
view VIEW2 from EQTRIV to EXTENDED-DOCUMENT {
  sort Elt -> Document,
  op _=_ -> _=_
}

--
-- LIST(D <= VIEW2) is the module for lists of documents and
-- "* {sort List -> DocTrace}" says that sort List is renamed DocTrace.
-- So, LIST(D <= VIEW2) * {sort List -> DocTrace} is the module for lists of
-- documents in which DocTrace, which denotes lists of documents, is used instead of List.
--
mod! DOC-TRACE {
  pr(LIST(D <= VIEW2) * {sort List -> DocTrace})
  op obsDocTrace : Document -> DocTrace
```

```
  vars D D1 : Document
  ceq obsDocTrace(D) = D, nil if isODoc(D) .
  ceq obsDocTrace(D) = D, obsDocTrace(obsPre(D)) if not isODoc(D) .
  --
  eq nth(obsDocTrace(D),0) = D . --  is deduced from the above.
}

--
-- A simulation is conducted.
--
open DOC-TRACE
  ops e1 e2 : -> Editing .
  op d : -> Document .
  eq d = edit(e2,copy(edit(e1,create))) .
  red obsDocTrace(d) .
  --
  -- The result is
  -- edit(e2,copy(edit(e1,create))) , copy(edit(e1,create)) , edit(e1,create) , create , nil
  --
close

--
-- A(d,i) = i \in inds obs_doc_trace(d)
-- B(d,i) = is_eD(obs_doc_trace(d)(i))
-- C(d,j) = j \in inds obs_doc_trace(d)
-- D(i,j) = j<i
-- E(i,j,k) = k \in {j,i-1}
-- F(d,k) = is_cD(obs_doc_trace(d)(k))
-- G(d,i,k) = obs_E(obs_E(obs_doc_trace(d)(i)) = obs_E(obs_doc_trace(d)(k))
--
-- \A{d:D}.\A{i:Nat}.A(d,i) => B(d,i) =>
      \A{j:Nat}.C(d,j) /\ D(i,j) /\ \A{k:Nat}.E(i,j,k) /\ F(d,k) => G(d,i,k) ... (*1)
--
-- where \A is \forall.
--
-- How should we parse formula (*1)?
--
-- (1) \A{d:D}.\A{i:Nat}.[A(d,i) => B(d,i) =>
      \A{j:Nat}.[C(d,j) /\ D(i,j) /\ \A{k:Nat}.[E(i,j,k) /\ F(d,k) => G(d,i,k)]]]
--   It is easy to find d,i,j such that the formula does not hold. So, this is not reasonable.
--
-- (2) \A{d:D}.\A{i:Nat}.[A(d,i) => B(d,i) =>
      \A{j:Nat}.[C(d,j) /\ D(i,j) /\ \A{k:Nat}.[E(i,j,k) /\ F(d,k)] => G(d,i,k)]]
--   The last occurrence of k is free. So, this is not reasonable.
--
-- It seems that the last occurrence of k should be j.
--
-- (3) \A{d:D}.\A{i:Nat}.[A(d,i) => B(d,i) =>
      \A{j:Nat}.[C(d,j) /\ D(i,j) /\ \A{k:Nat}.[E(i,j,k) /\ F(d,k)] => G(d,i,j)]]
--   It is easy to find k such that E(i,j,k) does not hold. For
--   example, let k be i. So, (3) clearly holds. It seems nonsense.
--
-- So, it seems that formula (*1) should be modified as follows:
--
-- \A{d:D}.\A{i:Nat}.[A(d,i) => B(d,i) =>
      \A{j:Nat}.[C(d,j) /\ D(i,j) /\ \A{k:Nat}.[E(i,j,k) => F(d,k)] => G(d,i,j)]]  ... (*2)
--
--

--
-- We are going to prove that formula (*2) is deduced from the definition.
--

-- PROOF OF (*2):
--
-- All quantifiers are moved to the head position as follows:
--
```

```
-- (*2) = \A{d:D}.\A{i:Nat}.\A{j:Nat}.
      [[A(d,i) /\ B(d,i) /\ C(d,j) /\ D(i,j) /\ \A{k:Nat}.[E(i,j,k) => F(d,k)]] => G(d,i,j)]
--      = \A{d:D}.\A{i:Nat}.\A{j:Nat}.\E{k:Nat}.
            [[A(d,i) /\ B(d,i) /\ C(d,j) /\ D(i,j) /\ [E(i,j,k) => F(d,k)]] => G(d,i,j)]
--      = \A{d:D}.\A{i:Nat}.\A{j:Nat}.\E{k:Nat}.
            [[ABCD(d,i,j) /\ [E(i,j,k) => F(d,k)]] => G(d,i,j)]
--
-- where \E is \exists and ABCD(d,i,j) is A(d,i) /\ B(d,i) /\ C(d,j) /\ D(i,j).
-- And the existentially quantified variable k is replaced with
-- f(d,i,j) where f is a skolem function as follows:
--
--      = \A{d:D}.\A{i:Nat}.\A{j:Nat}.
          [[ABCD(d,i,j) /\ [E(i,j,f(d,i,j)) => F(d,f(d,i,j))]] => G(d,i,j)]
--
-- Prove by structural induction on d.
--
-- I) Base case:
-- We are going to prove the formula
--
--   G1: \A{i:Nat}.\A{j:Nat}.[[ABCD(create,i,j) /\ [E(i,j,f(create,i,j)) =>
           F(create,f(create,i,j))]] => G(create,i,j)]
--
-- II) Induction cases:
-- We suppose the induction hypothesis
--
--   IH: \A{i:Nat}.\A{j:Nat}.[[ABCD(d,i,j) /\ [E(i,j,f(d,i,j)) =>
           F(d,f(d,i,j))]] => G(d,i,j)]
--
-- where d is an arbitrary document, and prove the two formulas
--
--   G2: \A{i:Nat}.\A{j:Nat}.[[ABCD(edit(e1,d1),i,j) /\ [E(i,j,f(edit(e1,d1),i,j)) =>
           F(edit(e1,d1),f(edit(e1,d1),i,j))]] => G(edit(e1,d1),i,j)]
--   G3: \A{i:Nat}.\A{j:Nat}.[[ABCD(copy(d1),i,j) /\ [E(i,j,f(copy(d1),i,j)) =>
           F(copy(d1),f(copy(d1),i,j))]] => G(copy(d1),i,j)]
--
-- where d1 is an arbitrary document and e1 is an arbitrary editing.
--
-- Thoese proofs are supported by CafeOBJ.
--
-- Note that i \in inds obs_doc_trace(d) can be expressed by 0 =< i
-- and i < len(obsDocTrace(d)) in CafeOBJ and
-- k \in {j,i-1} can be expressed by j =< k and k < i in CafeOBJ.
--
--
mod LEMMA {
  pr(DOC-TRACE)
  -- arbitrary values
  op d : -> Document
  ops i j : -> Nat
  -- operators representing lemmas or theorems
  op lemma1 : Document Nat Nat -> Bool
  -- skolem constants and functions
  op f : Document Nat Nat -> Nat
  -- CafeOBJ variables
  var D : Document
  vars I J K : Nat
  -- Lemmas or theorems to prove
  eq lemma1(D,I,J) =
      ((0 =< I and I < len(obsDocTrace(D))) and isEDoc(nth(obsDocTrace(D),I)) and
      (0 =< J and J < len(obsDocTrace(D))) and J < I and
      ((J =< f(D,I,J) and f(D,I,J) < I) implies isCDoc(nth(obsDocTrace(D),f(D,I,J)))))
                     implies
                     obsEdit(nth(obsDocTrace(D),I)) = obsEdit(nth(obsDocTrace(D),J)) .
}
--
-- I) Base case: G1 is proved.
--
```

```
-- G1: create
-- By case splitting.
-- G1.a: i = 0
open LEMMA
  eq d = create .
  eq i = 0 .
  red lemma1(d,i,j) .
close
-- G1.b: i = s(n)
open LEMMA
  op n : -> Nat . -- an arbitrary nat
  eq d = create .
  eq i = s(n) .
  red lemma1(d,i,j) .
close
--
-- II) Induction cases: G2 and G2 are proved assuming IH.
--
-- G2: edit(e1,d1)
-- By case splitting.
-- G2.a: i = 0
open LEMMA
  op e1 : -> Editing .
  op d1 : -> Document .
  eq d = edit(e1,d1) .
  eq i = 0 .
  red lemma1(d,i,j) .
close
-- G2.b.a: i = s(n) /\ nth(obsDocTrace(d1),n) = create
open LEMMA
  op e1 : -> Editing .
  op d1 : -> Document .
  op n : -> Nat .
  eq d = edit(e1,d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = create .
  red lemma1(d,i,j) .
close
-- G2.b.b: i = s(n) /\ nth(obsDocTrace(d1),n) = copy(d2)
open LEMMA
  op e1 : -> Editing .
  ops d1 d2 : -> Document .
  op n : -> Nat .
  eq d = edit(e1,d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = copy(d2) .
  red lemma1(d,i,j) .
close
-- G2.b.c.a: i = s(n) /\ nth(obsDocTrace(d1),n) = edit(e2,d2) /\ j = 0
open LEMMA
  ops e1 e2 : -> Editing .
  ops d1 d2 : -> Document .
  op n : -> Nat .
  eq d = edit(e1,d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = edit(e2,d2) .
  eq j = 0 .
  eq f(edit(e1,d1),s(n),0) = 0 .
  red lemma1(d,i,j) .
close
-- G2.b.c.b: i = s(n) /\ nth(obsDocTrace(d1),n) = edit(e2,d2) /\ j = s(n1)
open LEMMA
  ops e1 e2 : -> Editing .
  ops d1 d2 : -> Document .
  ops n n1 : -> Nat .
  eq d = edit(e1,d1) .
  eq i = s(n) .
```

```
  eq nth(obsDocTrace(d1),n) = edit(e2,d2) .
  eq j = s(n1) .
  --
  eq f(edit(e1,d1),s(n),s(n1)) = s(f(d1,n,n1)) .
  --
  red lemma1(d1,n,n1) implies lemma1(d,i,j) .
close
--
-- G3: copy(d1)
-- By case splitting.
-- G3.a: i = 0
open LEMMA
  op d1 : -> Document .
  eq d = copy(d1) .
  eq i = 0 .
  red lemma1(d,i,j) .
close
-- G3.b.a: i = s(n) /\ nth(obsDocTrace(d1),n) = create
open LEMMA
  op d1 : -> Document .
  op n : -> Nat .
  eq d = copy(d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = create .
  red lemma1(d,i,j) .
close
-- G3.b.b: i = s(n) /\ nth(obsDocTrace(d1),n) = copy(d2)
open LEMMA
  ops d1 d2 : -> Document .
  op n : -> Nat .
  eq d = copy(d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = copy(d2) .
  red lemma1(d,i,j) .
close
-- G3.b.c.b: i = s(n) /\ nth(obsDocTrace(d1),n) = edit(e2,d2) /\ j = s(n1)
open LEMMA
  op e2 : -> Editing .
  ops d1 d2 : -> Document .
  ops n n1 : -> Nat .
  eq d = copy(d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = edit(e2,d2) .
  eq j = s(n1) .
  --
  eq f(copy(d1),s(n),s(n1)) = s(f(d1,n,n1)) .
  --
  red lemma1(d1,n,n1) implies lemma1(d,i,j) .
close
-- G3.b.c.a.a: i = s(n) /\ nth(obsDocTrace(d1),n) = edit(e2,d2) /\
      j = 0 /\ len(obsDocTrace(d1)) = 0
open LEMMA
  op e2 : -> Editing .
  ops d1 d2 : -> Document .
  op n : -> Nat .
  eq d = copy(d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = edit(e2,d2) .
  eq j = 0 .
  eq len(obsDocTrace(d1)) = 0 .
  red lemma1(d,i,j) .
close
-- G3.b.c.a.b.a: i = s(n) /\ nth(obsDocTrace(d1),n) = edit(e2,d2) /\
      j = 0 /\ len(obsDocTrace(d1)) = s(n1) /\ n = 0
open LEMMA
  op e2 : -> Editing .
  ops d1 d2 : -> Document .
```

```
  ops n n1 : -> Nat .
  eq d = copy(d1) .
  eq i = s(n) .
  eq nth(obsDocTrace(d1),n) = edit(e2,d2) .
  eq j = 0 .
  eq len(obsDocTrace(d1)) = s(n1) .
  eq n = 0 .
  red lemma1(d,i,j) .
close
-- G3.b.c.a.b.b: i = s(n) /\ nth(obsDocTrace(d1),n) = edit(e2,d2) /\
      j = 0 /\ len(obsDocTrace(d1)) = s(n1) /\ n = s(n2)
open LEMMA
  op e2 : -> Editing .
  ops d1 d2 : -> Document .
  ops n n1 n2 : -> Nat .
  eq d = copy(d1) .
  eq i = s(n) .
  -- eq nth(obsDocTrace(d1),n) = edit(e2,d2) .
  eq nth(obsDocTrace(d1),s(n2)) = edit(e2,d2) .
  --
  eq j = 0 .
  eq len(obsDocTrace(d1)) = s(n1) .
  eq n = s(n2) .
  --
  eq f(copy(d1),s(s(n2)),0) = s(f(d1,s(n2),0)) .
  --
  red lemma1(d1,n,j) implies lemma1(d,i,j) .
close
--
-- Q.E.D.
--
```

F

On a Domain Model of "The Market"

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.

F.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.

F.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 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 F.1 attempts to illustrate possible transaction transitions between buyers and sellers.

Fig. F.1. Buyer / Seller Protocol

F.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 F.2 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. F.2. Trader=Buyer+Seller

F.1.3 Supply Chains

Figure F.3 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.

Fig. F.3. A Network of Traders and Supply Chains

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 ...).

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*.

F.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 F.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. F.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 F.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 F.4.)

Similarly for seller–agents (the right–hand side of Figure F.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.

Figure F.5 on the next page 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 F.6 on the following 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.

Fig. F.5. A Simple ("One Stage") Broker

Fig. F.6. A Multiple (here: Three) Stage Broker

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 deployed, and that these are based on the all the transaction kinds as first sketched.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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` [39].

type
[0] Θ, MSG
[1] Idx = {| 1..n |}

value
[2] sys: (Idx $\overrightarrow{m}$ Θ) → **Unit**
[3] sys(mθ) ≡ ‖ { tra(i)(mθ(i)) | i:Idx }

channels {tc[i,j]:MSG | i,j:Idx • i< j}

value
[4] tra: i:Idx → Θ → **in** {tc[j,i]|j:Idx•i≠j} **out** {tc[i,j]|j:Idx•i≠j} **Unit**
[5] tra(i)(θ) ≡ tra(i)(nxt(i)(θ))

[6] nxt: i:Idx → Θ → **in** {tc[j,i]|j:Idx•i≠j} **out** {tc[i,j]|j:Idx•i≠j} Θ
[7] nxt(i)(θ) ≡
[8] **let** choice = rcv ⌈⌉ snd **in**
[9] **cases** choice **of** rcv→receive(i)(θ), snd→send(i)(θ) **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, $\lceil\rceil$) "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 → Θ → in {tc[j,i]|j:Idx•i≠j} Θ
receive(i)(θ) ≡
    [] {let msg=tc[j,i]? in  update_rcv_state(msg,j)(θ) end | j:Idx}
```

Once the internal non–deterministic choice ($\lceil\rceil$) 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.

```
update_rcv_state: MSG × i:Idx → Θ → Θ
update_rcv_state(msg,j)(θ) ≡
  cases obs_Trans(msg) of
    mk_Del(_,_)
      → upd_rcv(msg,j)(upd_del(msg,j)(θ)),
    mk_Ret(_,_)
      → upd_rcv(msg,j)(upd_ret(msg,j)(θ)),
```

```
[9]      _ → upd_rcv(msg,j)(θ)
[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)(θ) ≡ deposit_trans((sU(msg),j),msg)(cond_rec(msg,j)(θ))
[1] upd_del(msg,j)(θ) ≡ deposit_delivery((sU(msg),j),msg)(θ)
[2] upd_ret(msg,j)(θ) ≡ remove_return((sU(msg),j),msg)(θ)

[3] cond_rcv(msg,j)(θ) ≡
[4]   if intial_trans(msg)(θ)
[5]     then θ
[6]     else remove_prior_trans(sU(msg),j)(θ) 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)(θ) ≡
[2]   let choice = ini ⌈⌉ res ⌈⌉ nor in
[3]   cases choice of
[4]     ini → send_initial(i)(θ),
[5]     res → send_response(i)(θ),
[6]     nor → remove_received_msg(θ) 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)(θ) ≡
  let choice = buy ⌈⌉ sell **in**
  cases choice **of**
    buy → send_init_buy(i)(θ),
    sell → send_init_sell(i)(θ) **end end**

send_response: i:Idx → Θ → **out** {tc[i,j]|j:Idx•i≠j} Θ
send_response(i)(θ) ≡
  let choice = buy ⌈⌉ sell **in**
  cases choice **of**
    buy → send_res_buy(i)(θ),
    sell → send_res_sell(i)(θ) **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 → Θ → **out** {tc[i,j]|j:Idx•i≠j} Θ
send_init_buy(i)(θ) ≡
  let choice = inq ⌈⌉ ord ⌈⌉ pay ⌈⌉ ret ⌈⌉ ... **in**
  let (j,msg,θ') = prepare_init_buy(choice)(i)(θ) **in**
  tc[i,j]!msg ; θ' **end end**

send_init_sell: i:Idx → Θ → **out** {tc[i,j]|j:Idx•i≠j} Θ
send_init_sell(i)(θ) ≡
  let choice = quo ⌈⌉ con ⌈⌉ del ⌈⌉ inv ⌈⌉ ... **in**
  let (j,msg,θ') = prepare_init_sell(choice)(i)(θ) **in**
  tc[i,j]!msg ; θ' **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 → Θ → **out** {tc[i,j]|j:Idx•i≠j} Θ
send_res_buy(i)(θ) ≡
  let (θ',msg)=sel_update_buy_state(θ), j=obs_trader(msg) **in**
  let (θ'',msg$'$) = response_buy_msg(msg)(θ') **in**
  tc[i,j]!msg$'$; θ'' **end end**

send_res_sell: i:Idx → Θ → **out** {tc[i,j]|j:Idx•i≠j} Θ
send_res_sell(i)(θ) ≡

let $(\theta'$,msg)=sel_update_sell_state(θ), j=obs_trader(msg) **in**
let $(\theta''$,msg$')$ = response_sell_msg(msg)(θ') **in**
tc[i,j]!msg$'$; θ'' **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.

F.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: (U $\times$ Idx) $\times$ MSG $\rightarrow \Theta \rightarrow \Theta$
deposit_delivery: (U $\times$ Idx) $\times$ MSG $\rightarrow \Theta \rightarrow \Theta$
remove_return: (U $\times$ Idx) $\times$ MSG $\rightarrow \Theta \rightarrow \Theta$
initial_trans: MSG $\times$ Idx $\rightarrow \Theta \rightarrow$ **Bool**
remove_prior_trans: U $\times$ Idx $\rightarrow \Theta \rightarrow \Theta$
remove_received_msg: $\Theta \rightarrow \Theta$

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 $\rightarrow$ Idx $\rightarrow \Theta \rightarrow$ Idx $\times$ MSG $\times\ \Theta$
prepare_init_sell: Choice $\rightarrow$ Idx $\rightarrow \Theta \rightarrow$ Idx $\times$ MSG $\times\ \Theta$

The above operations internal deterministically chooses which prior transactions to respond to.

obs_trader: MSG $\rightarrow$ 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 $\times$ Idx $\rightarrow \Theta \rightarrow \Theta$
upd_del: MSG $\times$ Idx $\rightarrow \Theta \rightarrow \Theta$
upd_ret: MSG $\times$ Idx $\rightarrow \Theta \rightarrow \Theta$
cond_rcv: MSG $\times$ Idx $\rightarrow \Theta \rightarrow \Theta$

sel_update_buy_state: $\Theta \rightarrow \Theta\ \times$ MSG
sel_update_sell_state: $\Theta \rightarrow \Theta\ \times$ MSG

response_buy_msg: MSG $\rightarrow \Theta \rightarrow \Theta\ \times$ MSG
response_sell_msg: MSG $\rightarrow \Theta \rightarrow \Theta\ \times$ 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.

F.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'.

G

On a Domain Model of CyberRail

G.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 [14] [17] [21] [22] [7] [16] [9] [8] [15] — using abstraction and modelling principles and techniques extensively covered in three forthcoming software engineering textbooks [10]. (ii) A term project with four MSc students. And (iii) Some fascination as whether one cold formalise an essence of the novel ideas of CyberRail. We strongly believe that we can capture one crucial essence of CyberRail — such as this paper will show.

The formalisation of CyberRail is expressed in the RAISE [41] Specification Language, RSL [39]. RAISE stands for Rigorous Approach to Industrial Software Engineering. In the current abstract model we especially make use of RSL's parallel process modeling capability. It builds on, ie., borrows from Tony Hoare's algebraic process concept of Communicating Sequential Processes, CSP [58].

G.2 A Rough Sketch Formal Model

G.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, Σ, Ω, CT, TC
  MΣ = Tx $\overrightarrow{m}$ Σ
channel
  {ct[p]:CT,tc[p]:TC|p:Tx}, cr:CR, rc:RC
value
  mσ_i:MΣ, ω_i:Ω

  cyberrail_system: **Unit** → **Unit**
  cyberrail_system() ≡ ‖ { traveller(p)(mσ_i(p)) | p:Tx } ‖ cyber(ω)

  cyber: Ω → **in** {tc[p]|p:Tx},cr **out** {ct[p]|p:Tx},rc **Unit**
  cyber(ω) ≡
    cyber_as_server(ω) ⌈⌉ cyber_as_proactive(ω) ⌈⌉ cyber_as_co_director(ω)

  traveller: p:Tx → Σ → **in** ct[p] **out** tc[p] **Unit**
  traveller(p)(σ) ≡ active_traveller(p)(σ) ⌈⌉ passive_traveller(p)(σ)

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.

G.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**
  ...
  updateΣ: Update → Σ → Σ
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 → Σ → **in** ct[p] **out** tc[p] **Unit**
  passive_traveller(p)(σ) ≡ **let** res = ct[p]? **in** updateΣ(res)(σ) **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 → Σ → **out** tc[p] Σ
  magic_f: $\Sigma \to \Sigma \times$ Request

  f(p)(σ) ≡ **let** (σ',req) = magic_f(σ) **in** tc[p]!req;σ' **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.

G.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: Ω → **in** {tc[p]|p:Tx} **out** {ct[p]|p:Tx} **Unit**
  cyber_rail_as_server(ω) ≡
    ⌷ {**let** req = tc[p]? **in** cyber(serve_traveller(p,req)(ω)) **end** | p:Tx}

  serve_traveller: p:Tx × Req → Ω → **in** {tc[p]|p:Tx} **out** {ct[p]|p:Tx} Ω
  serve_traveller(p,req)(ω) ≡
    case req **of**
      mkIniPl(pl) →
        let (ω',pls) = sugg_pls(p,pl)(ω) **in** ct[p]!pls;cyberrail(ω') **end**
      mkSelPl(pl) →
        let (ω',res) = res_pl(p,pl)(ω) **in** ct[p]!book;cyberrail(ω') **end**
      mkChgPl(pl) →

```
        let (ω′,pl′) = chg_pl(p,pl)(ω) in ct[p]!pl′;cyberrail(ω′) end
      mkBTrav(pl) → ...
      mkBTrai(pl) → ...
      ...
      mkLeTr(pl) → ...
      mkEnTr(pl) → ...
    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 G.2.3 rough sketch–formalises one of these.

```
type
  CR_act == gu_tr | no_tr | co_tr | wa_tr | ...
value
  cyber_as_proactive: Ω → out {ct[p]|p:Tx} Unit
  cyber_as_proactive(ω) ≡
    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 $\times$ Plan $\to \Omega \to \Omega \times$ Plan-**set**
  res_pl: p:Tx $\times$ Plan $\to \Omega \to \Omega \times$ Plan
  chg_pl: p:Tx $\times$ Plan $\to \Omega \to \Omega \times$ 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: $\Omega \to$ **out** {ct[p]|p:Tx} Ω
  guide_traveller(ω) $\equiv$
    let (ω',(ps,guide)) = any_guide(ω) **in** broadcast(ps,guide) ; ω' **end**

  any_guide: $\Omega \to \Omega \times$ (Tx-**set** $\times$ Guide)

  notify_traveller: $\Omega \to$ **out** {ct[p]|p:Tx} Ω
  commercial_to_travellers: $\Omega \to$ **out** {ct[p]|p:Tx} Ω
  warn_traveller: $\Omega \to$ **out** {ct[p]|p:Tx} Ω
  ...

  broadcast: Tx-**set** $\times$ CT $\to$ **Unit**
  broadcast(ps,msg) $\equiv$
    case ps **of** {}$\to$**skip**,{p}$\cup$ ps$'\to$ct[p]!msg;broadcast(ps$'$,msg) **end**

type
  CT = Guide | Notification | Commercial | Warning | ...
  Guide == mkGui(...)
  Notification == mkNot(...)
  Commercial == mkCom(...)
  Warning == mkWar(...)
  ...

G.3 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 what the title of [**?**] refers to as the *Information Infrastructure*. We look forward to report on that in a near future.

The aim 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.

G.4 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)