Informatics Models of Infrastructure Domains*

Dines Bjørner†
Informatics and Mathematical Modelling, Technical University of Denmark
Building 322, Richard Petersens Plads, DK–2800 Lyngby, Denmark
URI: www.imm.dtu.dk/~db, E–Mail: db@imm.dtu.dk

May 18, 2001

Abstract

By infrastructure informatics we understand the confluence of mathematical modelling, computing science and applications of computing in public administration, private business and industry, and in semi–public utilities such as for example transport and health–care.

In this invited paper we wish to alert the audience at the CSIT'01 held in Yerevan in September 2001, and the readers of its proceedings, to some new avenues of computing science, and some new demands for tighter relations between mathematical modelling and computing science, and between these and software engineering.

We hope to achieve our aim by showing a number of sketch domain models of central aspects or such large scale infrastructure components as railway systems, logistics, E–business, health–care, and financial service industries (such as banks, insurance companies, and the brokers, traders and exchanges of the securities industry).

The paper is quite long.

We have been careful in deciding this fact. In order to get the message across: Loud and clearly, namely that we have something new, something different, to offer. A new approach to software engineering, one that entails working out large, very large descriptions indeed, of domains, and from these carefully developing the requirements — and finally software — it is indeed necessary to bring large, believable, trustworthy "realistic" examples. Now, since the domains are indeed very large scale infrastructure components, the examples could be expected to be very much larger than they actually are in this paper. Hence we achieve a secondary purpose: To show that even very large scale infrastructure components can be comfortably described by a handful of professional software engineers.

*This is the text of a paper invited for presentation at CSIT'01, Yerevan, Armenia, 17–21 September, 2001

†The occasion for the delivery of this paper is the induction, of its author, as a Member of the Yerevan Branch of the Russian Academy of Natural Sciences

Contents

1 Introduction

The aims & objectives of this paper is to alert the reader to some new possibilities for the trustworthy development of indeed very large scale computing systems for indeed very large societal infrastructure components. The developments alluded to would typically involve close collaboration between computer & computing scientists, software engineers, mathematical modellers, and the clients: Customers representing one or another infrastructure component.

The wording of some subsections below has been predicated by the assumption that this paper reaches an audience rather different from the audiences reached through my other papers. The wording referred to may formulate certain attitudes towards current computer science cum software engineering academic teachings and current software engineering practices in a part of the world with which its author is more familiar than most coming from Europe, having travelled extensively in the Russias, and in the many republics that surround Russia.

1.1 Infrastructure and Infrastructure Components

By a country's infrastructure we mean all those component institutions, public utilities and facilities that serve to help create social and economic conditions for the well–being of its citizens. Examples of such components are: The transportation industry (roads, rails, air and sea lanes; trucking companies, railway companies, airlines, shipping companies), the health–care sector, the financial service industry, and the like. Infrastructure components (and their subcomponents) are geographically widely distributed, each of its enterprises has tens of thousands of concurrent (loosely or tightly control coupled, inter–communicating) activities going on at any one time, and they represent very large enterprises indeed.

Computing and communications systems, of these infrastructure component enterprises, when closely, minutely inspected, are very highly fragmented. Consists of many hundreds, if not thousands, of independently developed and operating computer (etc.) applications which, if they share anything, is some data on some hard disks — but with no assurance of a common semantic understanding of these data.

It is for such kind of systems that the current paper suggests ways and means of tackling their fragmentation. We believe, but it is, it must be stated, just a claim, merely a postulate, but it seems, an obvious one — we believe — that significant economic and social gains could be made when future systems are deployed, systems that embody infrastructure–wide knowledge and sharing of processes and data. The domain models of this paper shows how to start the development of such systems.

1.2 Trustworthy Development of Computing Systems

We have all too often heard about "EDP Scandals": Software systems that took 200%–400% more time and monies to develop, systems which, when deployed, brought about immense disappointments: They simply did not deliver what was expected. Their understanding of the application domain was appalling, their human computer user interfaces were dismal, and they were, to top it off, full of bugs, failing again and again !

The triptych software development sketched in Section 2 reflects more than 25 years of research, development and wide usage in Northern Europe. So—called "formal methods" are

certainly being increasingly widely accepted, as standard topics in leading universities, and in leading edge, high IT technology companies. In 1999 a world congress on Formal Methods drew 530 participants from North America, Europe and The Far East.

The author is currently readying a number of papers, and a rather comprehensive Software Engineering text book, summarising a quarter century's work, incl. some 75 previously published papers and around a dozen co–authored and co–edited books on the subject of formal techniques in software development. We refer to these papers and the book for a more full account of trustworthy software engineering [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

1.3 Informatics Collaboration

Informatics, to us, is the confluence of mathematical modelling (ie. the use of existing mathematics), computer & computing science, software engineering and computing applications. For far too long have the teaching of software engineering (cum programming) been far too isolated from mathematical modelling. As many examples of Section 3 will show: There are "zillions" of fascinating needs for mature interplays between these disciplines.

A closing section Section 4.1 will review the examples of Section 3 in the light of these postulated opportunities.

1.4 Methodological Approach

By a *method* we understand a set of *principles* for `selecting` and `applying` a set of *techniques,* using a set of *tools,* in `analysing` a problem in order `efficiently` to `construct` an `efficient artifact` (here software[1]).

By *methodology* we understand the study and knowledge of one or more *methods.*

Since the principles of a method are guidelines to be used by people they cannot be formalised. A great number of techniques can. And tools based on such techniques likewise. But to call the methods formal is unfortunate. Better would be to use such double terms as formal techniques, precise techniques or logic techniques — as imprecise or illogic techniques are probably not desired, but informal are !

1.5 Structure of Paper

The paper is long because of its Section 3's many and long examples. We decided to break all conventional traditions for standard length conference proceedings papers to deliver this 63 paper. We did so because it is important for the reader to see, in one place, ie. this paper, a sufficient variety of examples, each with their different modelling approaches, yet with the same aims: Models of what is indeed conceived of as very large scale actual life application domains.

So, after Section 2's capsule view of the three main engineerings of software, Section 3 will illustrate five different domains: Railway systems, logistics, electronic commerce, health–care systems and financial services. A final section will conclude.

The formal specifications of this paper makes free use of the `RAISE` Specification Language, `RSL` [12, 13] but could as well have used combinations of `CSP` [14, 15, 16] with either of `VDM-SL` [17, 18, 19, 20], or `Z` [21, 22, 23], or `B` [24].

[1] In general: *computing & communication systems.*

2 A Triptych of Software Engineering

Before software[1] can be designed, their requirements must be engineered. And before requirements can be expressed, we must understand the domain.

2.1 The Sciences and Engineering of Computer, Computing and Software

By software engineering we understand a confluence of domain engineering (see Sect. 2.4), requirements engineering (see Sect. 2.5), and software design (see Sect. 2.6).

By computer science we understand the study and knowledge of the properties (the what) of entities that may exist inside computers.

By computing science we understand the study and knowledge of how to construct those things that may exist inside computers.

The engineer "walks the bridge" between science and technology: Constructs technology based on scientific insight, and analyses (typically somebody else's) technology in order to ascertain its possible scientific content.

2.2 Software

By software we understand not just so–called *executable code*, but also all the documentation that went into its development: *domain descriptions* (Section 2.4), *requirements definitions* (Section 2.5), and all the stages and steps of *software design* (Section 2.6); and not just that (!), but also all the *manuals* that are necessary for their and its *maintenance*, *installation*, and proper end–user *training* and *use !*

2.3 Domains

By domain we understand the *application area*, the *"actual world"* in which some desired (or actual) *software*[1] is to reside (resp. resides).

We shall illustrate a number of domains: *Railway Systems* (Section 3.1), *Logistics* (Section 3.2), *E–Commerce* (Section 3.3), *Health–care Systems* (Section 3.4), and *Financial Service Industries* (Section 3.5).

2.4 Domain Engineering

By domain engineering we understand a set of science–based engineering activities which results in a set of *models*, ie. a set of *documents informing* about, *describing*, and *analysing* "all" the *phenomena* of the "actual world" relevant to *requirements engineering* — and usually more than that !

Which these *phenomena* are, how to *identify* them, how to *inform* about them, and how to *describe* and *analyse* them, is the subject of a *domain engineering methodology* [4].

A *domain description* is *indicative:* Describes what there is. It makes no reference — whatsoever ! — to any (subsequently) *required software*, let alone to any *design* of such *software*[1].

Domain engineering is both a science and an art ! Whether a *domain description* is adequate ('sufficient'), or not, can usually be somehow ascertained through a consensus of as many relevant domain *stake–holders* as are feasible.

2.5 Requirements Engineering

By requirements engineering we understand a set of science–based engineering activities which results in a set of *models,* ie. a set of *documents informing* about, *describing,* and *analysing* what is expected from *software*[1] that is to support activities of the *domain.*

Which these *requirements* are, how to *identify* them, how to *inform* about them, and how to *describe* and *analyse* them, is the subject of a *requirements engineering methodology* [5].

A *requirements description* is *putative:* Describes what *software* there shall be. It makes no reference — whatsoever ! — to any *design* of such *software*[1].

Requirements engineering is both a science and an art ! Crucial parts of a *formal requirements description,* the *domain requirements description,* must stand in a formal relation to a similarly *formal domain description.* Whether a *requirements description,* additionally, is adequate (is 'sufficient'), or not, can usually be somehow ascertained through a consensus of as many relevant domain *stake–holders* as are feasible.

2.6 Software Design

By software[1] design we understand a set of science–based engineering activities which results in a set of *models,* ie. a set of *documents informing* about, *describing,* and *analysing* the *construction, structure* and *behaviour* of the *software*[1] that supports required activities of the *domain.*

A *software description* is *imperative:* Describes how the *computing & communication system,* when following the prescription of the software design, behaves.

The software design is expected to satisfy, ie. be correct wrt., the *requirements description* in the context of the *domain description.*

2.7 Discussion

The above triptych of software engineering is based on the dogma of applying mathematical abstractions in all phases (domain engineering, requirements engineering and software design), ie. on applying so–called 'formal methods'. Our recent reports cover this dogma [2, 3, 4, 5].

3 Examples of Domain Models

Several domain examples will be given. Each example will feature its own presentation style. We have chosen this approach in order to illustrate as a largest variety of narration, abstraction and modelling principles and techniques.

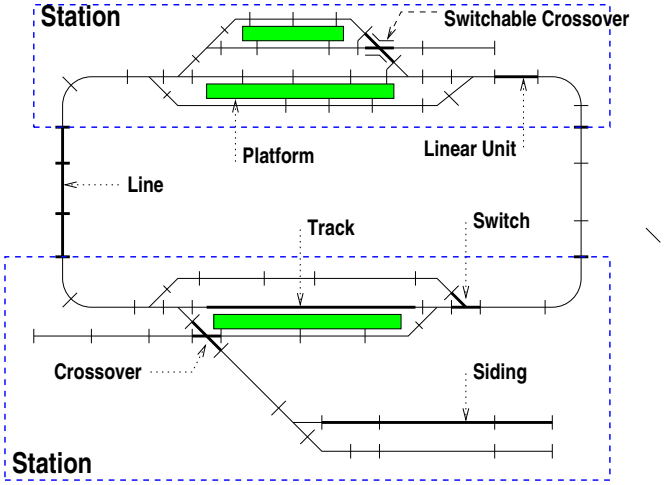
3.1 Railway Systems

We structure this domain specification according to the model–oriented discrete mathematics modelling concepts being deployed: Sets, Cartesians, lists, maps and functions.

3.1.1 Set–oriented Description

Narrative — Static Rail Nets: We refer to Figure 1.

Figure 1: A Sample Rail Net



Railway systems[2] contain rail nets, one per system, n:N. [1] Rail nets contain [1–2] one or more lines (the "things" that connect stations), ℓ:L, [2–3] and two or more stations, s:S. It does not make realistic sense, for other than tourism oriented vintage and toy model railway nets to have just one station ! [1,4] Lines and stations, and hence the net, consists of one or more rail units, u:U. Cf. Figure 1. [1,5] Rail units are delimited by connectors, c:C, such that: [6] *Linear* units have two distinct *connectors*; [7] simple *switch*es, also known, in the trade, as *junctions*, or *turnouts*, in English, resp. American; as *weiche* in German, and in French *aguilette*, have three distinct *connectors*; [8] *switchable crossovers* have four distinct *connectors*; [8] and so do non-switchable, ie. *simple crossovers*.

We refer to Figure 2.

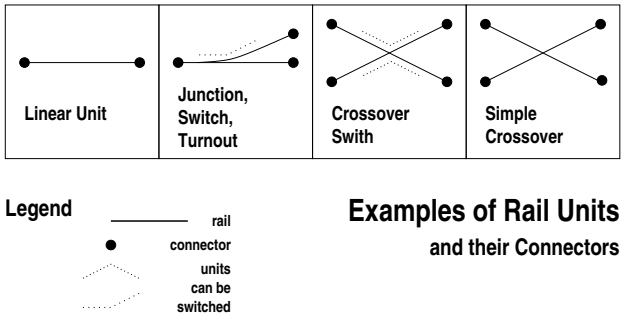
We stop for now and show a formalisation.

Formalisation — Static Rail nets:

type
[1] N, L, S, U, C
value
[2] obs_Ls: N → L-**set**
[3] obs_Ss: N → S-**set**
[4] obs_Us: (N|L|S) → U-**set**
[5] obs_Cs: (N|L|S|U) → C-**set**
[6] is_linear: U→**Bool**

[2]The current author, as child, was one of those *"deprived"* children who never had a toy model railway ! That may be one reason for explaining our fascination with the railway topic. One that seems shared by many. Another is the delight he has in travelling around, in Europe and the Far East, by train.

Figure 2: Four Rail Units



[7] is_switch: U→**Bool**
[8] is_cross: U→**Bool**

axiom
∀ u:U • is_linear(u) ⇒ **card** u=2 ∨ is_switch(u) ⇒ **card** u=3 ∨ is_cross(u) ⇒ **card** u=4

There may be other kinds of units. And there may be such other kinds which also have either two, or three, or four distinct connectors, etc. We decided, [5], that since nets, lines and stations consisted of units, and units had connectors, then it might be foresighted to assume that therefore also nets, lines and stations had connectors. Whether this turns out to be a useful "abstraction" remains to be seen. "Throwing" such observer functions in is, at this stage, "for free". Only if we later, in software designs, make explicit use of the more general observer functions, then it might come at a cost: We might have to implement separate connector query procedures for nets, line, stations and units — instead of perhaps just being able to observe connectors from nets. We shall see.

Narrative — Static Rail net Constraints, Part I: [9] As already said, nets consist of one or more lines, [10] and two or more stations, [11–12] and these of at least one unit each. [13] No two otherwise distinct lines of a net have any units in common. [14] No two otherwise distinct stations of a net have any units in common. [15] No line of a net and no station of that net have any units in common.

Formalisation — Static Rail net Constraints, Part I:

axiom
[9] ∀ n:N • **card** obs_Ls(n)≥1,
[10] ∀ n:N • **card** obs_Ss(n)≥2,
[11] ∀ ℓ:L • **card** obs_Us(ℓ) ≥ 1
[12] ∀ s:S • **card** obs_Us(s) ≥ 1
[13] ∀ n:N, ℓ,ℓ′:L • {ℓ,ℓ′}⊆obs_Ls(n) ∧ ℓ≠ℓ′⇒ obs_Us(ℓ) ∩ obs_Us(ℓ′) = {}
[14] ∀ n:N,s,s′:S • {s,s′}⊆obs_Ss(n) ∧ s≠s′ ⇒ obs_Us(s) ∩ obs_Us(s′) = {}
[15] ∀ n:N,ℓ:L,s:S • ℓ∈ obs_Ls(n) ∧ s ∈ obs_Ss(n) ⇒ obs_Us(ℓ) ∩ obs_Us(s) = {}

Narrative — Static Rail net Constraints, Part II: [16] Stations have *tracks,* sl:SL. [19] Lines and tracks consist of *linear sequences* of linear units.

[20] No line and station track (of a net) have units in common. [21] No two otherwise distinct traks of any station have units in common. [22] For every connector of a net there are at most two units sharing that connector. [23] A line of a net connects exactly two distinct stations of that net. A set of units is said to form a linear sequence, sq, if the set can be rearranged into a non–empty list of exactly the same units such that adjacent units share one connector.

Formalisation — Static Rail net Constraints, Part II:

type
[16] SL
value
[17] obs_SLs: (N|S) → SL-**set**
[18] obs_Us: SL → U-**set**

axiom
[19] ∀ ℓ:L,sl:SL •
∀ u:U • u ∈ obs_Us(ℓ) ∪ obs_U(sl) ⇒ is_linear(u) ∧ sq(obs_Us(ℓ)) ∧ sq(obs_Us(sl))
[20] ∀ n:N,ℓ:L,s:S,sl:SL •
ℓ∈ obs_Ls(n) ∧ s ∈ obs_Ss(n) ∧ sl ∈ obs_SLs(s) ⇒ obs_Us(ℓ) ∩ obs_Us(sl) = {}
[21] ∀ n:N,s,s′:S,sl,sl′:SL • {s,s′}⊆obs_Ss(n) ∧
sl ∈ obs_SLs(s) ∧ sl′ ∈ obs_SLs(s′) ∧ sl≠sl′ ⇒ obs_Us(s) ∩ obs_Us(s′) = {}
[22] ∀ n:N,c:C • c ∈ obs_Cs(n) ⇒ 2 ≥
⇒ **card**{u|u:U•u ∈ obs_Us(n)∧c ∈ obs_Cs(u)}
[23] ∀ n:N, ℓ:L • ℓ∈ obs_Ls(n)
⇒ ∃ ! s,s′:S • {s,s′}⊆obs_Ss(n) ∧ s≠s′ ∧ distinct(ℓ,s,s′)

Auxiliary Functions

value
sq: U-**set** → **Bool**
sq(us) ≡
card us = 1 ∨ ∃ u:U • u ∈ us ⇒ **card**(obs_Cs(u) ∩ cs(us \ {u}))=1 ∧ sq(us \ {u})

distinct: L × S × S → **Bool**
distinct(ℓ,s,s′) ≡
let lcs = cs(obs_Us(ℓ)), scs = cs(obs_Us(s)), scs′ = cs(obs_Us(s′)) **in**
∃ c,c′:C • c ∈ lcs ∧ c ∈ scs ∧ c′ ∈ lcs ∧ c′ ∈ scs′ **end**

cs: U-**set** → C-**set**
cs(us) ≡ **union**{{ c | c:C • c ∈ obs_Cs(u) } | u:U•u ∈ us }

distinct is an auxiliary predicate, and cs is an auxiliary function. They are introduced in order to make the axioms look shorter. distinct determines uniqueness of "end of line" and station

"perimeter" connectors. The predicate expresses that there are unique connectors, one in each of two distinct stations that "connect" to unique line connectors.

3.1.2 Cartesian–oriented Description

Narrative — Dynamic Rail Nets, I: Units are either closed or open for traffic. Linear units can be traversed in either of two directions, but other factors can determine whether they are either closed, or open for traffic in just one, or in just the other, or in both directions. Several factors may determine this "whether–or": Signals, for example, up and/or down line, ie. in one or the other, or both directions "away" from any specific linear unit, may be set so as to "forbid" traversal in some or all directions. The part of the rail net, where the linear unit is located, may be of such physical or other characteristics so as to effectively prevent certain directions of traffic — viz. units in a marshaling yard where special clamps — intended for braking the speed of rail cars — may be so arranged as to make it physically destructive if cars are forced through in "the wrong" direction !

Switch units are set in either of two positions. Verbally, without reference to any figure, let us informally model a switch by the composite character ${}^{c_|}Y_c^{c_/}$. The switch is said to have three connectors: $\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" (to) $c_|$, or "fork" (to) $c_/$.

We refer to Figure 3.

Figure 3: States of Sample Units

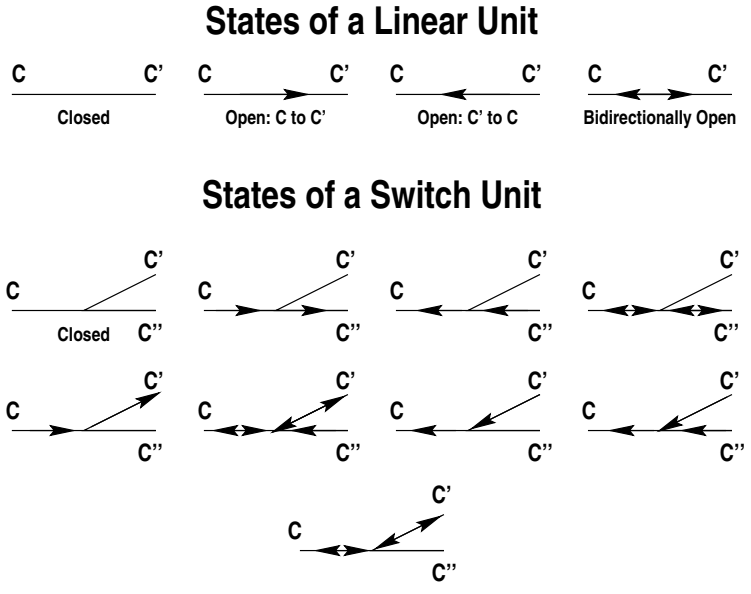


Figure 3 shows four, respectively nine possible sets of directions, zero, one or more, that linear and switch units can be in. If part of a unit has an arrow in some direction then it can be traversed in that direction. Some parts of a switch unit can have arrows in both directions

as determined by the setting of the switch.

Unit States: We therefore introduce *abstract concepts* of unit *paths*, unit *state*, and unit *state space*. A unit path, *syntactically*, is a pair of unit connectors — with the pragmatics that such a unit path designates a direction of unit traversal. A unit state, *semantically*, is a set of unit paths. And a unit state space, again *semantically*, is a set of unit states. Next we show the state space, ω_{g_s}, of all possible states of a switch.

$$\begin{array}{ll}\{ & \{\},\{(c,c_|)\},\{(c_|,c)\},\{(c,c_|),(c_|,c)\},\\ & \{(c,c_/)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c)\},\{(c_/,c),(c_|,c)\},\\ & \{(c,c_/),(c_/,c)\} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}\end{array}$$

ω_{g_s} ideally models a general switch. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$. Nothing is said about how a state is determined: Who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (ie. invisible, intangible) signals up or down the rail away from the switch.

Formalisation — Dynamic Rail Nets, I:

type
[24] P = {| (c,c′):C×C • c≠c′ |}
[25−26] Σ = P-**set**, Ω = Σ-**set**
value
[27−28] obs_Σ: U → Σ, obs_Ω: U → Ω
axiom
[29] ∀ u:U•obs_Σ(u)∈ obs_Ω(u) ∧
[30] ∀ (c,c′):P•(c,c′)∈ obs_Σ(u)⇒{c,c′}⊆obs_Cs(u)

[254 P is the path type. [25–26] Σ is the state that a unit may be in at any on "point in time", and Ω is the space of all those states that a unit may be in "over time" ! [27–28] From any unit one can observe its current state and state space. [29] For all σ states of some u, those σ are in the state space of that u. [30] For all paths of any unit the connectors of that path must be connectors of the unit.

Discussion: The notion of unit is a powerful one, one that may not be that easy to fully grasp: Think of a physical piece of rail that someone, an official from the owner of a rail net, has designated to be a real, an actual unit. There is it: Typically lying on the ground, perhaps it is just a linear unit, but curved in the terrain, both horizontally and vertically; with sleepers and many other "adornments" about which we have so far not stated anything. But any additional attributes, with their current values, that you may think of, are subsumed by the model being developed: One can simply add more observer functions, etc.

But that same unit, over time, with its changing states, how can we express that it is indeed the same unit ? So far we have not really had to say that. But we will soon need. So let us define a "sameness", an equality predicate.

Formalisation — Dynamic Rail Nets, I, Continued:

value
  equal: U × U → **Bool**
axiom
  ∀ u,u′:U • obs_Ω(u) = obs_Ω(u′) ⇒ equal(u,u′)

3.1.3 List–oriented Description

Preamble Analysis: We refer to Figure 4.

Figure 4: Route of a Rail Net

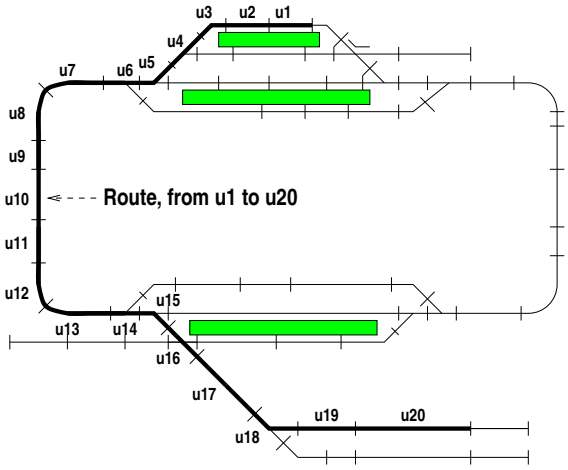


Figure 4 attempts to illustrate the notion of a route as a sequence of unit paths. The units of a path may be open or closed, and if open they may be open in the "right" or in a "wrong" direction.

Narrative — Open and Closed Routes: A *route* is a sequence of paths through units such that adjacent unit paths "connect" ! A route may be well–formed. A route is *planned* either if all its units are closed, or all its units are open and in the "direction" of the unit path. In the first case the *route is* said to be *closed*. In the second case the *route is* said to be *open*.

Formalisation — Open and Closed Routes

type
  R′ = (P × U)*
  R = {| r:R′ • wf_R(r) |}
value
  wf_R: R′ → **Bool**
  wf_R(r) ≡ ∀ i:**Nat** • i ∈ **inds** r ⇒ {i,i+1}⊆**inds** r ⇒

let ((c,c′),u)=r(i),((c″,c‴),u′)=r(i+1) **in**
(c,c′)∈ u ∧ (c″,c‴)∈ u′ ∧ c′=c″ **end**

open, close: R → **Bool**
open(r) ≡ ∀ (p,u):(P×U) • (p,u) ∈ **elems** r ∧ (p,u) ∈ obs_Σ(u)
close(r) ≡ ∀ (p,u):(P×U) • (p,u) ∈ **elems** r ∧ obs_Σ(u)={}

Narrative — Route Planning: Given a net one can generate the possibly infinite set of all finite and indefinite length routes of the net. Some may revisit units "as the route 'meanders' on" ! Such routes are said to be crossing. If crossing routes "enters" the "same"[3] path repeatedly then it is said to be cyclic.

Given a net one can, more specifically, can define the set of all non–crossing routes between any two units of the net. Non–crossing route sets between some pairs of units may be empty.

We define the above route generation functions and testing predicates by first defining the set of all routes of a net.

Basis Clause: For every unit, u, of a net, n, let p be a path of that unit, then ⟨(p,u)⟩ is a route of the net.

Induction Clause: Let r be any route of a net, n. Let r be expressible as a concatenation of a possibly empty subroute r′ and a singleton route ⟨((c,c′),u)⟩ either as ⟨((c,c′),u)⟩^r′ or r′^⟨((c,c′),u)⟩. If there exists a singleton path either ⟨((c″,c),u′)⟩, or ⟨((c′,c″),u′)⟩, or both, then either ⟨((c″,c),u′)⟩^⟨((c,c′),u)⟩^r′ or r′^⟨((c,c′),u)⟩^ ⟨((c′,c″),u′)⟩ or both are routes of the net.

Formalisation — Route Planning, I:

value
all_rs: N → R-**infset**
all_rs(n) ≡
let us = obs_Us(n) **in**
let srs = s_rs(us) **in**
let ars = srs ∪
{ r^⟨((c″,c),u′)⟩ | r:R • r ∈ ars ∧ ∃ c,c′,c″:C, u,u′:U,r′:R •
r = r′^⟨((c,c′),u)⟩ ∧ ⟨((c″,c),u′)⟩∈ srs }
∪
{ ⟨((c′,c″),u′)⟩^r | r:R • r ∈ ars ∧ ∃ c,c′,c″:C, u,u′:U,r′:R •
r = ⟨((c,c′),u)⟩^r′ ∧ ⟨((c′,c″),u′)⟩∈ srs }
in ars **end end end**

s_rs: U-**set** → R-**set**
s_rs(us) ≡
{⟨((c,c′),u)⟩ | u:U, c,c′:C • u ∈ us ∧ (c,c′) ∈ obs_Σ(u) }
assert: ∀ r:R • r ∈ s_rs(us) ⇒ **len** r = 1

spec_rs: U × U → N ~→ R-**infset**

[3] "Sameness" of rail units, and hence of paths, was dealt with in Section 3.1.2.

```
spec_rs(fu,tu)(n) ≡
  { r | r:R • r ∈ all_rs(n) ∧ let (p,u)=r(1), (p′,u′)=r(len r) in u=fu ∧ u′=tu end }
  pre {fu,tu}⊆obs_Us(n)
```

Discussion of Routes: The route generator function all_rs is a fix point generating function. The fix point generated is the solution to the recursive equation is ars (for all routes). The solution may be an infinite set — but all its elements will be of finite length. The infiniteness is mathematically acceptable. But we can do with finite sets, and we can do with non-crossing routes.

The function spec_rs generates the possibly infinite, possibly empty set of routes between two given units, even if they are the same !

We have not bothered to constrain the route sets only to plannable routes. That can be expressed later. For now we define functions which generate all non-crossing routes, and all acyclic routes.

Formalisation — Route Planning, II:

```
value
  nc_rs: N → R-set
  nc_rs(n) ≡
    { r | r:R • r ∈ all_rs(n) ∧ ∼∃ i,j:Nat • i<j ∧ {i,j}⊆inds r ∧
          let (p,u) = r(i), (p′,u′) = r(j) in u = u′ end }

  ac_rs: N → R-set
  ac_rs(n) ≡
    let ars = all_rs(n) in
    { r | r:R • r ∈ all_rs(n) ∧ ∼∃ i,j:Nat • i<j ∧ {i,j}⊆inds r ∧
          let ((c,c′),u) = r(i), ((c″,c‴),u′) = r(j) in
          c = c‴ assert: u=u′ end } end
```

3.1.4 Map–oriented Description

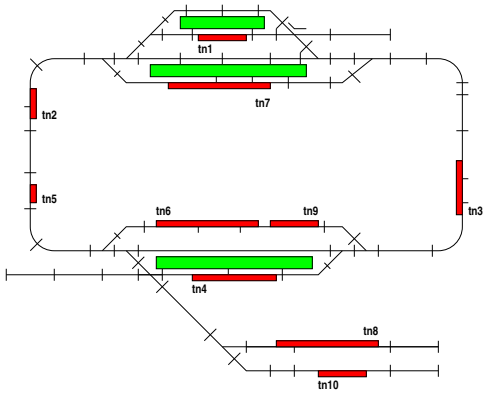
Narrative — Train Positions: We refer to Figure 5.

Figure 5 intends to illustrate ten, uniquely identified, trains, tn1, ..., tn10.

Each train occupies one or more units. Trains, identified by tn5, tn9, and tn10, occupy just one rail unit. The other trains straddle two or more units.

Analysis — Train Positions: We have not said anything about the state of the units: Whether open or closed, and, if open, whether in commensurate directions ! We have shown intended direction of train traffic by placing respective, unique train identifiers at the head of the trains.

Figure 5: Train Positions



Formalisation — Train Positions:

type
[1] Tn, TR
[2] TP = Tn $\overrightarrow{m}$ R
value
[3] obs_Tn: TR → Tn
[4] obs_R: TR → R
axiom
∀

[1] Tn and TR are the sorts of train indentifiers and trains. [2] TP is a model of train positions: each unique train occupies a route. [3] From a train we can observe, obs_Tn, its train identifier, and [4] obs_R its route.

Since we are modelling the domain there is no need to restrict routes of distinct trains to not overlap: Accidents do occur !

Narrative — Train Movement: We refer to Figure 6.

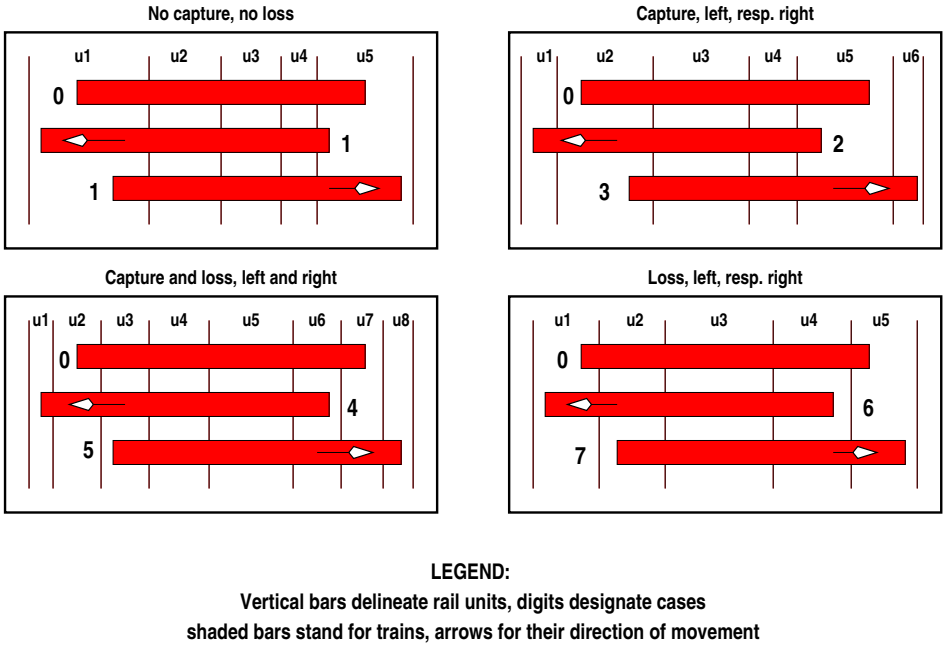
Figure 6 intends to illustrate the movement, in either direction, of a train.

We explain the eight move transitions (two are numbered 1., otherwise 0.–7.) shown in Figure 6.

[0→1] The train has moved, but it has stayed within the same route, represented here just by its units: u1,u2,u3,u4,u5.[4] [0→2] The train has moved: "capturing" a unit, u1, to occupy u1,u2,u3,u4,u5. [0→3] The train has moved: "releasing" a unit, u2, to occupy u2,u3,u4,u5,u6. [0→4] The train has moved: both "capturing" a unit, u9, and "releasing" —

[4]The granularity of a route: It being composed from units, some of which may be a few meters long, others of which may be, say, hundreds of meters long, that "coarse" granularity — with no change to the model — currently prevents us from a smoother, more continuous model of movement.

Figure 6: Train Movement



"instantaneously" — a unit, u4, to occupy u1,u2,u3,u4,u5,u6. Etcetera. Notice: Three "real" movements in either direction makes six, and one indiscernable "move".

These are, "ideally" speaking, the only logical movements that trains can make. In addition trains may derail: "Fall" off the rail, "straddle" across "parallel" units, etc.

Formalisation — Train Movement:

value
wf_Move: R × R → **Bool**
wf_Move(r,r′) ≡
equal(r,r′) /∗ 0→1 ∗/
∨ ∃ r″:R, u:U, (c,c′):(C×C) • (c,c′) ∈ obs_Σ(u) ∧
r′=⟨((c,c′),u)⟩^r″ ⇒ equal(r,lst(r′)) /∗ 0→2 ∗/
∨ r′=r″^⟨((c,c′),u)⟩ ⇒ equal(r,fst(r′)) /∗ 0→3 ∗/
∨ r′=⟨((c,c′),u)⟩^fst(r)⇒equal(fst(r),lst(r′)) /∗ 0→4 ∗/
∨ r′=lst(r)^⟨((c,c′),u)⟩⇒equal(lst(r),fst(r′)) /∗ 0→5 ∗/
∨ r′=⟨((c,c′),u)⟩^r ⇒ equal(fst(r),r′) /∗ 0→6 ∗/
∨ r′=r^⟨((c,c′),u)⟩ ⇒ equal(r,lst(r′)) /∗ 0→7 ∗/

fst: R → R, fst(r) ≡ ⟨r[i]|q≤i<**len** r⟩, lst : R → R, lst(r) ≡ ⟨r[i]|q<i≤**len** r⟩

equal: R × R → **Bool**
equal(r,r′) ≡ **len** r = **len** r′ ∧

∀ i:**Nat** • i ∈ **inds** r ⇒
  let ((c,c′),u) = r[i], ((c″,c‴),u′) = r′[i] **in** (c,c′) = (c″,c‴) ∧ equal(u,u′) **end**

Narrative — Time Tables: Also time–tables can be used to show model–oriented map abstractions. Several variants can be shown, and 'transfer' functions (isomorphisms) expressed between them. We select an arbitrary model: To every train, known by its name (or number), there is associated, in the chosen time–table, a train journey. A train journey associates stations with train arrival and depature times.

Formalisation — Time Tables:

type
  Tn, Sn, T
  TT′ = Tn $\overrightarrow{m}$ J
  TT = {| tt:TT′ • ∀ j:J • j ∈ **rng** tt ⇒ wf_J(j) |}
  J′ = Sn $\overrightarrow{m}$ (T × T)
  J = {| j:J′ • wf_J(j) |}
  SV′ = Sn × (T × T)
  SV = {| (sn,(t,t′)):SV′ • t < t′ |}
  SVl′ = SV*
  SVl = {| svl:SVl • wf_SVl(svl) |}
value
  wf_J: J′ → **Bool**
  wf_J(j) ≡ **card dom** j ≥ 2 ∧ ∃ svl:SVl • svl = ⟨(s,j(s))|s:Sn•s ∈ **dom** j⟩ ⇒ wf_SVl(svl)

  wf_SVl: SVl′ → **Bool**
  wf_SVl(svl) ≡
    ∀ i:**Nat**•{i,i+1}∈ **inds** svl ⇒
      let ((,(t1,t2)),(,(t2,t4)))=(v[i],v[i+1]) **in** t1<t2<t3<t4 ∧... **end**

We could, of course, instead have chosen:

type
  J′ = SV*
  J = {| j:J′ • wf_SVl(j) |}

Etcetera.

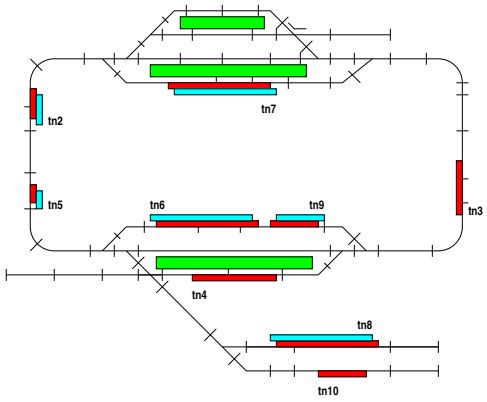
3.1.5 Function–oriented Description

Narrative — Train Traffic: We refer to Figure 7.

Figure 7 intends to illustrate the dynamics of train traffic. The two, "thin", adjacent but "skewed" (i. "slightly offset") rectangles, designate the movement of certain trains.

By *train traffic* we understand a function, continuous over a closed time interval, from time to pairs of *rail nets* and *train positions*. By *train positions* we mean a discrete map from

Figure 7: Train Positions



trains to *open routes*. The *rail net* thus changes over time, to reflect that rail units change state, ie. to open, to reverse and to close units.

Formalisation — Train Traffic:

type
TN, Vel, AccDec /∗ trains, velocity, acceleration/deceleration ∗/
TF′ = T → (N × (TN $\xrightarrow[m]{}$ R))
TF = {| tf:TF′ • wf_TF(tf) |}
value
wf_TF: TF′ → **Bool**
obs_Tn: TN → Tn, obs_Vel: TN → Vel, obs_AccDec: TN → AccDec

Narrative — Train Traffic Well—formedness: The train routes must be open at the time observed, and must all be routes of the network at the time observed. Between any two "sufficiently close" time points trains must only make well–formed moves. Etcetera.

Formalisation — Train Traffic Well—formedness: Left as an exercise !

Narrative — Traffic Scheduling: Given a time table that is assumed well–formed wrt. a rail net, and given such a net, one can define the possibly, usually, infinite set of all traffics that satisfy the net and time table. From such a set one can select a schedule that is optimal wrt. a number of criteria we shall omit.

Formalisation — Traffic Scheduling:

value
schedules: TT × N → TF-**infset**

schedules(tt,n) **as** tfs
  pre wf_TT_N(tt,n)
  post ∀ tf:TF • tf ∈ tfs ⇒ satisfy(tf)(tt,n)

schedule: TF-**infset** → TF
schedule(tfs) **as** tf
  pre tfs ≠ {}
  post tf ∈ tfs ∧ ...

satisfy: TF → (TT × N) → **Bool**
satisfy(tf)(tt,n) ≡ ...

3.1.6 Discussion

We have developed core aspects of a model of railways: Nets, time tables, traffic and scheduling. The present model is based on many previous models co–developed with Søren Prehn, Chris George, Li Xiaoshan, a group of five railway professionals (Jin Danhua, Dong YuLin, Liu Xin, Sun Guoqing, and Ma Chao), S. Parthasarathy — as part of UNU/IIST's[5] R&D for and with the Chinese Ministry of Railways [25, 26, 27, 28, 29].

3.2 Logistics

We present this example in a rough sketching and formal model style; different from the previous and following examples. The reason is that it might perhaps be useful to explore the logistics concepts a bit more informally before showing a terse narrative and its formalisation. As a consequence we keep the narrative to a minimum.

3.2.1 Rough Domain Sketches

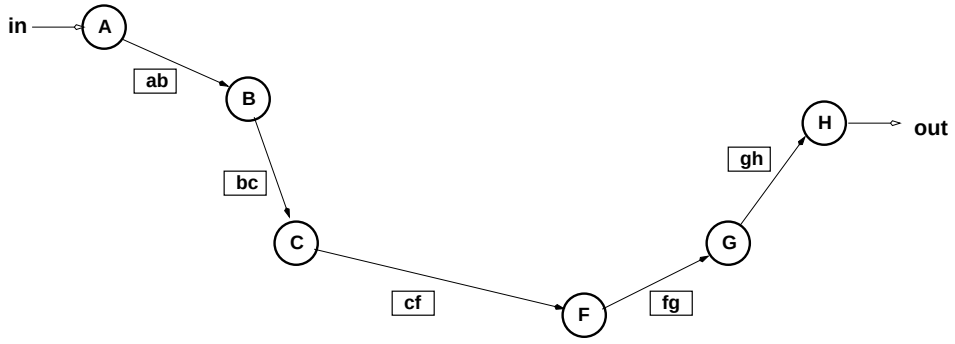
We encircle the problem by informally, and not necessarily systematically, (hence the term 'rough') sketching some situations centering on freight clients and logistics firms.

Scenario 1 — A Freight Transport: Let ϕ designate some piece of cargo, γ, some freight — for example a box of some volume, of some weight, of some value, of some fragility (fragile, or robust, etc.), of some security risk (flammable, poisonous or non-poisonous, etc.), sent by sender α in city A to some receiver ω in city H. Let ϕ be transported as indicated in Figure 8:

Freight ϕ follows transport route $A{\rightarrow}B{\rightarrow}C{\rightarrow}F{\rightarrow}G{\rightarrow}H$. From $A{\rightarrow}B$ freight ϕ is transported as described by the bill-of-lading part ab. From $B{\rightarrow}C$ freight ϕ is transported as described by the bill-of-lading part bc. From $C{\rightarrow}F$ freight ϕ is transported as described by the bill--of-lading part cf. From $F{\rightarrow}G$ freight ϕ is transported as described by the bill-of-lading part fg. From $G{\rightarrow}H$ freight ϕ is transported as described by the bill-of-lading part gh.

[5] UNU/IIST UN University's International Institute for Software Technology, located in Macau SAR, China. The current author was the first and founding UN Director of this research and post–graduate cum post–doctorial training centre. Please refer to `www.iist.unu.edu`.

Figure 8: A Freight Transport



These bill-of-lading parts are part of the total, or overall bill-of-lading, $\beta o \lambda_\phi$. An overall bill-of-lading registers: sender α, receiver ω, the logistics firm $\mathcal{L}$ which has planned and is now overseeing (monitoring) the "execution" of $\beta o \lambda_\phi$, the particulars of the freight γ — as indicated above, including the total cost to clients α and ω, pick-up and delivery conditions, insurance conditions, *&c.*

Each bill-of-lading part xy describes relevant particulars of the specific transport company $\mathcal{F}$ which carries freight ϕ from X to Y, date and times of transport (departure from X and arrival at Y), cost to logistics firm $\mathcal{L}$ of the transport of freight ϕ from X to Y on that particular date, etc. *&c.*

Logistics firm $\mathcal{L}$ now sees to it that the freight ϕ is delivered to relevant transport companies $\mathcal{F}_{xy}$ — initially from customer α to $\mathcal{F}_{ab}$, and, along the route $A \rightarrow B \rightarrow C \rightarrow F \rightarrow G \rightarrow H$ from $\mathcal{F}_{bc}$ to $\mathcal{F}_{cf}$, from $\mathcal{F}_{cf}$ to $\mathcal{F}_{fg}$, from $\mathcal{F}_{fg}$ to $\mathcal{F}_{gh}$, and from $\mathcal{F}_{gh}$ to receiver ω. How logistics firm $\mathcal{L}$ does this we presently leave undefined. In other words, logistics firm $\mathcal{L}$ traces the progress of freight ϕ along the route and regularly informs sender (α) and receiver (ω) about the progress, any delays or advances, etc.

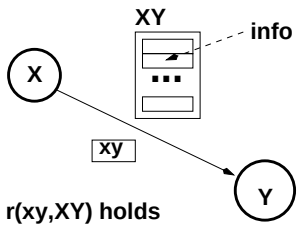
Etcetera.

Scenario 2 — Freight Planning, I: In order to plan — "to do the 'logistics' of" — a freight transport from location X to location Y logistics firms $\mathcal{L}$ must possess information, let us call it (them) point-to-point transport tables, XY.

These point-to-point transport tables contain information (info) about which transport companies provides transport between these locations (points), and, for each of these, their transport schedules, fees, costs, conditions, etc. *&c.* We refer to this kind of information as XY. That part of a total bill-of-lading, xy which concerns the path from X to Y is derived from XY: That is: there is a "correspondence" predicate r such that $r(xy, XY)$ holds.

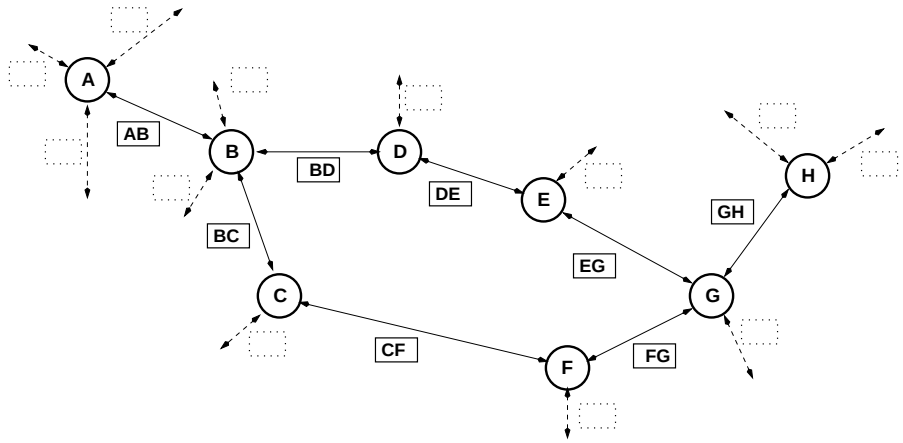
Scenario 3 — Freight Planning, II: In order to service a sender, α, and a receiver, ω, wrt. to a transport from α to ω, ie. from point (location) A, to point H, which, as shown in Figure 10, are connected by several routes, the logistics firm refers to a logistics network

Figure 9: Path Transport Facilities



(See Figure 10), and inspects all relevant point-to-point tables — based on finding all routes between A and H, here just two.

Figure 10: Part of a Logistics Net



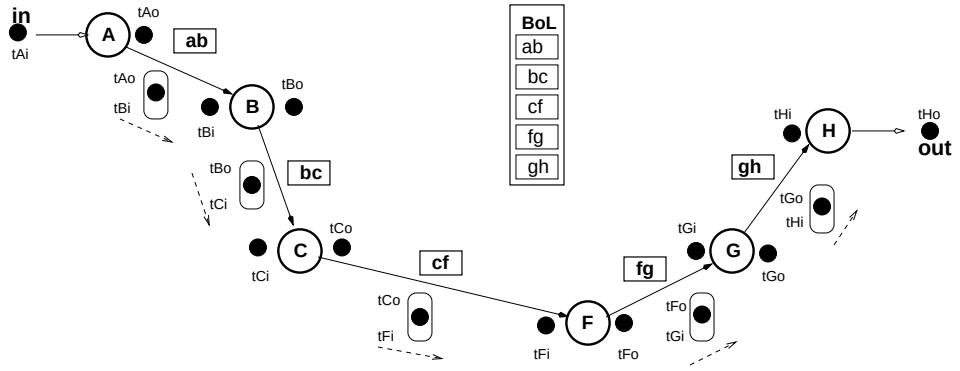
The logistics firm, based on client preferences for example selects the route of Figure 8. More specifically: Amongst several alternative route choices of transport, as described in the logistics network, the logistics firm first selects all relevant point-to-point tables, and from these find or negotiates the, or an, optimal combination of point-to-point transport firms and their scheduled (or perhaps even negotiates unscheduled) transport times, fees, conditions. The result is a transport plan, which when approved by the clients (α and ω), is made into a Bill-of-Lading $\beta o \lambda_\phi$ for the particular freight ϕ.

Now the actual transport can start at some time, tAi, after client approval of the transport plan.

Figure 11 intends to illustrate the trace (ie. the "history"), or a simulation, of the transport, resp. a transport, from A to H.

Initially: Freight $\phi_{\alpha\omega}$ (shown as a black bullet •) is delivered by, or informally (ie. not shown)

Figure 11: The Trace of a Specific Transport



fetched from client α at time tAi.[6] The freight is passed on to the first transport at time tAo, commensurate with Bill-of-Lading part **ab**.

Generally: The freight leaves point X at time tXo, and is transported from X to Y commensurate with Bill-of-Lading part **xy** between times tXo and tYi. The transport is shown as a rounded box within which we illustrate the freight bullet (•). These transport are shown "as moving" in the appropriate direction — by the dashed arrow. The freight (ie. the transport) arrives at point Y at time tYi where the freight (not necessarily the transport) resides till time tYo when the freight is either passed on to next transport, or, in this case for $Y = H$ fetched by receiver ω.

Scenario 4 — The State of a Logistics Network: We wish to analyse a notion of state: The state of all "executions" of a specific logistic firm's, or all logistic firms', Bill-of-Ladings.

Seen from the point of view of one logistic firm, Figure 12 is intended to be a snapshot at some (arbitrary) time t, of where 28 different Bill-of-Lading freights along the routes between points on the paths between A and H are.

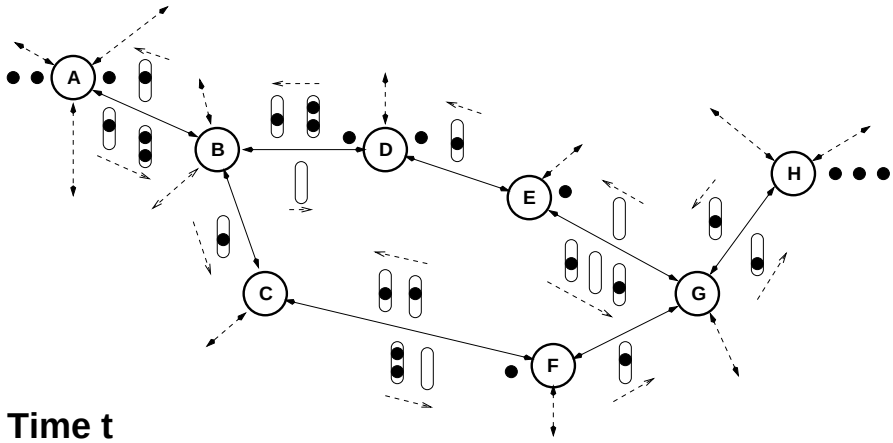
Some are moving in one direction, "to the right", ie. towards H, others are moving "to the left". Some transports carry one, some two, and some no freights — understood to be with respect to a specific logistics firm and only on the stretches shown.[7] The transports shown not carrying freight (for a, or the, particular logistics firm) probably, so we intended it, were supposed to indeed carry some freight — but that freight probably was mishandled or arrived too late for loading onto the shown transports.

Scenario 5 — Transactions: Clients α and ω, senders and receivers, often express queries concerning possible or actual freights, and state orders leading to changes in the state of a

[6] tXi stands for **t**ime **i**nput (delivered) at **X**. tXo stands for **t**ime **o**utput (delivered) from **X**.

[7] That is: there could be other executions of Bill-of-Ladings between other points not shown. Also note that we allow that freights shown — either on transports or entering or leaving the logistics network — are intended to be transported only at some proper sub-part of the routes from A to H, resp. from H to A.

Figure 12: A State of a Logistics Net



transportation. Logistics firms need regularly review the performance of past transportations. Etcetera. We illustrate possible such transactions (in *slanted* text):

Client to Logistics Firm: (1) *Can, and how would you transport this* $1\frac{1}{2}$ *ton 4–by–6—by–8 foot box from San Luis, Argentina to Novosibirsk, Siberia, Russia ?.* (2) "The above" + *the fastest way as from tomorrow ?* (date given). (3) "The above" (first item) + *the cheapest way ?* (4) "The above" (first item) + *with the fewest number of transfers [priority 1], the fastest way [priority 2], and cheapest possible [priority 3] ?* Logistics Firm to Client: (5) *Here is our answer to your query of such-and-such-data !* Logistics Firm to Transport Firm: (6) *Pls. send us your transport schedules for the period of (two dates are given).* Transport Firm to Logistics Firm: (7) *Pls. be informed about our new fee schedules.*

Queries are supposed to be presented in the form of a Logistics Plan, ie. a hypothetical Bill-of-Lading.

(8) Client to Logistics Firm: *Please accept this (ie. "such-and-such") freight and start its transport as per your proposed Logistics Plan ?.* (9) *Where, along the route, is my (ie. "such--and-such") freight now ?.* (10) *Please abort the (ie. "such-and-such") transportation and destroy it ?* Logistics Firm to Transport Firm: (11) *Please [re–]schedule the following freight items as indicated.* Transport Firm to Logistics Firm: (12) *We are sorry to inform you that our transport (such–and–such) sank (if boat), crashed (if flight), or (whatever).* Clients to Logistics Firm: (13) *Why was my freight late, or damaged, or not delivered to receiver ?* Logistics Firm to itself (!): (14) *What is percentage of "on-time" (eg. less than 2% late), of between 2% and 10% late, freight during the last 12 months ?* (15) *What is profit of our business: The difference between actual costs and invoiced fees in the last three, six and twelve months ?*

We observe that these "commands" introduce a number of new concepts, some explicitly, some rather implicitly.

3.2.2 A Brief Domain Requirements Narrative

We have rough sketched a larger domain than will be formalised. We will now provide a narrative for the formalised part of a well–behaving domain such as a desired, a required, software system shall ensure it.

A logistics system consists of the following reactive behaviours: customers (senders, receivers) of freight transport, logistics firms arranging such transports, transport companies which operate conveyors (and hence of these, ie. of transport vehicles [trucks, trains], boats and aircrafts) that convey freight between hubs (and hence of these), along the routes (and hence of these) as they form a route network. That is, of c customers, ℓ logistics firms, f transport companies, k conveyors, h hubs and the route network

Important components of these behaviours include freight items, bill–of–ladings, and conveyor time and fee tables.

The route network can be viewed as a graph whose nodes are the hubs and whose directed edges are the routes. Hubs have a *capability* for *receiving*, from logistics firms freight items for *loading*, and a *capacity* for temporarily *storing freight items unloaded* from conveyors for further *transfer* to next conveyors, or for final *delivery* to logistics firms.

A bill–of–lading is a description, a document. It "defines" the route: The load (origin), transfer and final (destination) hubs; the load and unload times (ie. schedule); the sending and receiving logistics firms, etc.

From hence we focus only on the operational meaning of a bill–of–lading: the traces of temporary storages at hubs, loadings and unloading onto, respectively from conveyors. That is, we shall describe the actions implied by a bill–of–lading. But we will do so by going straight to a formalisation of that relevant part of the logistics system.

3.2.3 Domain Requirements Formalisation

In this section we formalise the domain of well–behaving hubs and conveyors such as a required software system shall help ensure.

Route Network: The network is modelled as follows: Hubs form graph nodes labelled with hub names (HIdx) and immediate hub–to–next–hub routes form edges. There can be many such between any pair of hubs. These can be therefore be distinctly labelled and otherwise provided with route information.

type
  HIDx, Rn, RInfo
  $\mathrm{RN}' = \mathrm{HIDx} \xrightarrow{m} (\mathrm{HIdx} \xrightarrow{m} (\mathrm{Rn} \xrightarrow{m} \mathrm{RInfo}))$
  $\mathrm{RN} = \{|\ \mathrm{rn}{:}\mathrm{RN}' \bullet \mathrm{wf\_RN(rn)}\ |\}$
  $\mathrm{R} = \mathrm{HIDx}^*$
value
  wf_RN: $\mathrm{RN}' \xrightarrow{m}$ **Bool**
  wf_RN(rn) $\equiv \cup \{$**dom** rs$|$rs:(HIdx $\xrightarrow{m}$ (Rn $\xrightarrow{m}$ RInfo))$\bullet$rs $\in$ **rng** rn$\}\subseteq$**dom** rn
  routes: RN $\to$ R**-infset**
  routes(rn) $\equiv$
    let rs$=\{\langle$hj,hj$'\rangle|$hj,hj$'$:HIdx$\bullet$hj $\in$ **dom** rn$\wedge$hj$'$ $\in$ **dom** rn(hj)$\}$ **in**

let rs′ = rs ∪{r^⟨hj⟩|hj:HIdx,r:R•r ∈ rs′∧**let** hj′=r[**len** r] **in** hj ∈ **dom** rn(hj′) **end**}
rs′ **end end**

A route network defines an infinite set of all finite length routes between any pair of connected hubs.

Bill of Ladings: An example bill of lading:

value
bol:BoL = (fn,(h_0,to,ko,⟨(t1a,h1,t1d,k1),(t2a,h2,t2d,k2)⟩,td,h_d))

informs about the name, fn, of the freight item for which it is the bill of lading, the hub origin ho, the hub destination h_0, the initial departure time to, the initial conveyor name ko, and if the route involves intermediare hubs, say two, then arrival time, t1a, at first hub h1, departure time, t1d, from that hub on next conveyor k1, and then arrival time, t2a, at next hub h2, departure time, t2d, from that hub on final conveyor k2. In other words:

value
rn:RN
type
BoL′ = Fn × (HIdx × T × Kn × (T×sj:HIdx×T×Kn)* × T × HIdx)
BoL = {| bol:BoL′ • wf_BoL(bol)(rn) |}
value
wf_BoL: BoL′ → RN → ... → **Bool**
wf_BoL(fn,(h_0,to,ko,lst,td,h_d))(rn)(...) ≡ **let** r = route(bol) **in** r ∈ routes(rn) ∧ ... **end**
route: BoL → R
route(fn,(h_0,to,ko,lst,td,h_d)) ≡ ⟨ho⟩^⟨sj(lst[i])|1≤i≤**len** lst⟩^⟨h_d⟩
hubs: BoL → HIdx-**set**, hubs(bol) ≡ ...
convs: BoL → Kn-**set**, convs(bol) ≡ ...
nexth: HIdx × Bol → HIdx
nextk: HIdx × Bol → KIdx

The ... are meant to express the wellformedness of bill of ladings wrt. departure and arrival times (t) for named conveyors (k) according to transport company and logistics firm transport schedules.

Freight: From a transported freight item one can observe its freight name, its bill of lading, and the position of the freight: Either at a hub or on a conveyor from a hub.

type
F, W == F_at_H(HIdx) | F_on_K(HIdx,KIdx)
value
obs_Fn: F → Fn, obs_BoL: F → BoL, obs_W: F → W
axiom
∀ f:F •
let fn = obs_Fn(f), bol = obs_BoL(f), w = obs_W(f) **in**

```
    let (fn′,(h0,to,ko,lst,td,hd)) = bol in fn′=fn ∧
    cases w of
      F_at_H(hj) → hj ∈ hubs(bol),
      F_on_K(hj,ki) → (hj,ki)=(h0,ko) ∨
        ∃ hj′:HIdx • hj′ ∈ hubs(bol)∧(hj,ki)=(nextH(hj′,bol),nextK(hj′,bol))
    end end end
```

Hubs: From a hub state one can observe the freight temporarily stored at that hub, the freight to be loaded on a next, named conveyor, and the freight to be delivered to final receivers.

```
type
  HΣ
value
  obs_Fs: HΣ → F-set
  obs_Fs: KIdx → HΣ → F-set
  obs_final_Fs: HΣ → F-set
axiom
  ∀ hσ:HΣ •
    let nfs = union{ obs_Fs(kn)(hσ) | ki:KIdx } in
    nfs ∩ obs_final_Fs(hσ) = {} ∧ obs_Fs(hσ) = nfs ∪ obs_final_Fs(hσ) end
```

Conveyors: From a conveyor state one can observe its route, its current load of freights, and its current position: At a hub or en route, between hubs.

```
type
  KΣ, KIdx
  WK == K_at_H(HIdx) | K_en_R(HIdx,HIdx)
value
  obs_KIdx: KΣ → KIdx, obs_R: KΣ → R,
  obs_Fs: KΣ → F-set, obs_WK: KΣ → WK
  next_Travel: KΣ → R
axiom
  ∀ kσ:KΣ •
    let ki = obs_KIdx(kσ), r = obs_R(kσ), fs = obs_Fs(kσ), wk = obs_WK(kσ) in
    ∀ f:F • f ∈ fs ⇒
      let bol = obs_BoL(f), w = obs_W(f) in
      ∃ j,j′:Nat•j ∈ inds route(bol)∧j′ ∈ inds r⇒
        route(bol)[j]=r[j′]∧∃ j″:Nat•j″ ∈ inds r∧j″>j′∧(route(bol))[j+1]=r[j″] ⇒
        cases (wk,w) of
          (K_at_H(hj),F_on_K(hj′,ki′))→ r[j′]=hj ∧ ki=ki′,
          (K_en_R(hj,hj′),F_on_K(hj″,ki′))→ r[j′]=hj ∧ ki=ki′
    end end end
```

```
value
  next_H: HIdx → KΣ → HIdx
  next_H(hj)(kσ) ≡
    let r = obs_R(kσ), wk = obs_WK(kσ) in
    cases wk of
      K_en_R(,hj′) → hj′,
      K_at_H(hj′) → assert: hj=hj′ ;
        if hj = r[ len r ] then h_d next_Travel(kσ)
        else let i:Nat • i ∈ inds r ⇒ r[i]=hj in r[i+1]
    end end end end
```

Conveyor routes will here be considered recurrent. An observed route:

$\langle hj_1,hj_2,...,hj_n\rangle$

is made recurrent as follows. The route, the conveyor *travel*, starts at hub hj_1, next stop is at hub hj_2, and so forth. "Last" stop, on a specific *travel*, is at hub hj_n. Then a next *travel*, starts. obs_R applied to the conveyor, once it has reached hub hj_n, that is: Is in that "final" state, now yields the next *travel*, for example, some variant of a "reverse" *travel:*

$\langle hj_n,hj'_{n-1},...,hj'_1\rangle$

By suitably decorating the hub names, except the first, viz.: hj'_{n-1}, ..., hj'_1, a next, a "reverse" travel does not have to "mirror" the preceding travel.

System: We focus on just the hub and conveyor behaviours. Between hub and conveyors sets of freight items are transferred (either direction) when conveyors are at hubs. We model this transfer ability by synchronisation and communication over channels. Hubs are stationary, never move. (Conveyors are mobile, always, almost always move.) When a conveyor is at a hub it synchronises and communicates with that hub, first providing the hub with the freight to be unloaded and then waiting for freight to be loaded.

```
channel
  { khc[ki,hj] | ki:KIdx, hj:HIdx } F-set
value
  hub: hj:HIdx → HΣ → in,out{khc[ki,hj]|ki:KIdx} Unit
  hub(hj)(hσ) ≡
    [] { let ufs = khc[ki,hj]? in
      let lfs = to_be_loaded(ki)(hσ) in
      khc[ki,hj]!lfs ; hub(hj)(update_state(ufs,lfs)(hσ)) end end
      | ki:KIdx }

  to_be_loaded: KIdx → HΣ → F-set

  update_state: (F-set × F-set) → HΣ → HΣ
  update_state(ufs,lfs)(hσ) as hσ′
```

```
    pre ufs ∩ obs_Fs(hσ)={}∧lfs⊆obs_Fs(hσ)∧...
    post lfs ∩ obs_Fs(hσ′)={}∧ufs⊆obs_Fs(hσ′)∧...

con: ki:KIdx → KΣ → in,out{khc[ki,hj]|hj:HIdx} Unit
con(ki)(kσ) ≡
    let w = obs_WK(kσ), r = obs_R(kσ) in
    cases w of
        K_at_H(hj) → con(ki)(at_hub(ki,hj,r)(kσ)),
        K_en_R(hj,) → con(i)(move(ki,hj,r)(kσ))
    end end

at_hub: KIdx × HIdx × R → KΣ → KΣ
at_hub(ki,hj,r)(kσ) ≡
    let (ufs,kσ′) = unload_freight(hj)(kσ) in
    khc[ki,hj]!;ufs;
    let lfs = khc[ki,hj]? in
    load_freight(lfs)(kσ) end end

unload_freight: HIdx → KΣ → F-set × KΣ
unload_freight(hj)(kσ) as kσ′
    post ∼∃ f ∈ obs_Fs(kσ′) • to_be_unloaded_at(f,hj) ∧...

load_freight: F-set → KΣ → KΣ
load_freight(lfs)(kσ) as kσ′
    pre lfs ∩ obs_Fs(kσ) = {} ∧...
    post lfs ⊆ obs_Fs(kσ′) ∧...

move: KIdx × HIdx × R → KΣ → KΣ
move(ki,hj,r)(kσ) as kσ′
    pre obs_WK(kσ) = K_at_H(hj) ∧...
    post let wk = obs_(kσ′), hj′ = nextH(kσ) in
            wk = K_en_R(hj,hj′) end
```

Discussion: We ask the reader to compare the two behaviours: hub and conveyor. Their definition only focuses on well–behaviours: Only freight to be unloaded at a given hub is so unloaded, and all such freight. Only freight to be loaded onto a given conveyor is so loaded, and all such freight. The movement of conveyors is hinted at by the move function: It abstracts movement as discretionary steps, from hub to hub via being 'en route'.

3.2.4 Discussion

We have unravelled a description of first a domain of logistics, and finally of requirements to fragments of a software system for the support of loading and unloading of freight to, respectively from conveyors while at hubs. That is: The informal domain description of

Section 3.2.1 portrayed a domain in which things could and will go wrong: Freight is being lost, loaded onto wrong conveyors, unloaded from conveyors at wrong hubs, etc. That rough sketch domain description was followed up, ever so briefly, in Section 3.2.2, by a brief narrative of a logistics system that was assumed well–behaved: No misroutings, etc. Finally, in Section 3.2.3, we modelled the domain requirements for such a well–behaving logistics system.

3.3 E–Business

To understand e–business we must understand the "space" — "the market" — in which business takes place today. From a domain analysis of the market we may then be able, more believably, to arrive at what e–business could be.

3.3.1 Domain: "The Market"

Government, Business and Citizen Transactions: In the case of the government, business and citizen infrastructure interactions we can postulate the following *domain* (ie. not necessarily computer & data communication supported), transactions: $\mathcal{G}2\mathcal{G}$: Government institutions, $\mathcal{G}$, buy services from other $\mathcal{G}$, and $\mathcal{G}$ sell services to other $\mathcal{G}$. The $\mathcal{G}$ pay with monies (obtained through taxes, etc.), respectively offer free services, in return. $\mathcal{G}2\mathcal{B}$: $\mathcal{G}$ buy services, or request taxes, from businesses ($\mathcal{B}$), and pay, respectively offer free services, in return. $\mathcal{G}2\mathcal{C}$: $\mathcal{G}$ buy services (hire), or request taxes, from citizens ($\mathcal{C}$), and pay, respectively offer free services, in return. $\mathcal{B}2\mathcal{G}$: Businesses ($\mathcal{B}$) buy services from $\mathcal{G}$, and pay $\mathcal{B}$ for these either through having already paid taxes or by paying special fees. $\mathcal{B}2\mathcal{B}$: $\mathcal{B}$ buy merchandise or services from other $\mathcal{B}$, and $\mathcal{B}$ offer merchandise or services to other $\mathcal{B}$. $\mathcal{B}$ usually pay for these outright. $\mathcal{B}2\mathcal{C}$: $\mathcal{B}$ buy services from citizens: ie. hire temporary or permanent staff (employment), and $\mathcal{B}$ pay for these through salaries. $\mathcal{C}2\mathcal{G}$: Citizens ($\mathcal{C}$) obtain services from $\mathcal{G}$ (passport, drivers licence, etc., health–care, education, safety and security, etc.) and $\mathcal{C}$ pay for these either by paying special fees or through having already paid taxes. $\mathcal{C}2\mathcal{B}$: $\mathcal{C}$ buy merchandise from $\mathcal{B}$, and $\mathcal{C}$ pay for this. $\mathcal{C}2\mathcal{C}$: Two or more $\mathcal{C}$ together enter into political "grass–root" organisations, or leisure–time hobby club activities, or just plainly arrange meetings (incl. BBQ parties); and the two or more $\mathcal{C}$ "pay" for this by being "together".

Traders — Buyers and Sellers: Above we have stressed that also government (institutions) are part of the more general concept of `E-Business,` some aspects of contractual obligations, and a seeming "symmetry" between partners to any such contract (ie. buy, sell, etc.). As such we have stressed that "The Market" consists of buyers and sellers, whom we, as one, refer to as traders.

Traders — Agents and Brokers: An *agent,* to us, while we are still only discussing the *domain,* is a trader that acts (in a biased manner) on behalf of usually one other trader (either a buyer, or a seller), vis—a–vis a number of other traders (sellers, respectively buyers), in order to secure a "best deal". A *broker,* to us, while we are still only discussing the *domain,* is a trader that acts (in a neutral manner) on behalf one or more buyers and one or more sellers in order to help them negotiate a "deal."

Schematic Transactions Sequences of contractual transactions can be understood in terms of "primitive" transactions:

A buyer `inquire`s as to some merchandise or service. A seller may respond with a `quote.` A buyer may `order` some merchandise or service. A seller may `confirm` an order. A seller may `deliver` an order. A buyer may `accept` a delivery. A seller may send an `invoice.` A buyer may `pay` according to the invoice. A buyer may `return,` within warranty period, a delivery. And a seller may `refund` such a return.

We have, deliberately, used the "hedge" 'may':

A trader may choose an action of `no response,` or a trader may inform that a transaction was `misdirected,` or a trader may `decline` to `quote, order, confirm, deliver, accept, pay` or `refund !`

Formalisation of Syntax

type
Trans == Inq|Ord|Acc|Pay|Rej|Qou|Con|Del|Acc|Inv|Ref|NoR|Dec|Mis

Annotations: The first five, resp. the next six items above, list the 'buyer', respectively the 'seller' initiated transaction types. The last three items above lists common transaction types.

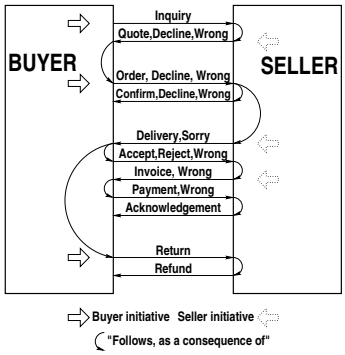
U below stand for unique identifications, including time stamps (T), Sui for surrogate information, and MQP alludes to merchandise identification, quantity, price.

U, T, Su1, Su2, MQP
Inq = MQP × U
Qou = (Inq|Su1) × Inf × U
Ord = Qou|Su2 × U
Con = Ord × U
Del = Ord × U
Acc = Del × U
Inv = Del × U
Pay = Inv × U
Rej = Del × U
Ref = Pay × U
NoR = Trans × U
Dec = Trans × U
Mis = Trans × U
value
obs_T: U → T

Annotations: In general we model, in the *domain,* a "subsequent" transaction by referring to a complete trace of unique, time stamped transactions. Thus, in general, a transaction "embodies" the transaction it is a manifest response to, and time of response.

Figure 13 attempts to illustrate possible transaction transitions between buyers and sellers.

Figure 13: Buyer / Seller Protocol



Transaction Sequences Figure 14 attempts to show the possible sequences of transactions as related to one "trade": From inquiry through refunding, that is: For one pair of buyer/seller.

"The Market" Figure 15 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.

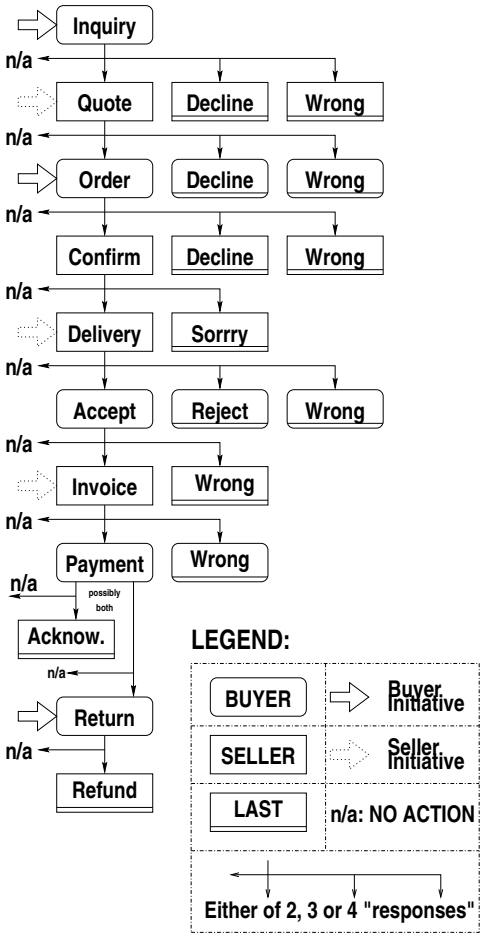
Figure 16 attempts to show "an arbitrary" constellation of buyer and seller traders. It highlights three supply chains. Each chain consists, in this example, of a "consumer", a retailer, a wholesaler, and a producer.

Formalisation of Process Protocols "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 treatment of how traders come to know of one another. We focus only on the internal and external non–determinism which is always there, in the *domain*, when transactions are selected, sent and received.

Our model is in a variant of `CSP`, but expressed "within" `RSL` [12].

type
  Θ
  Idx = {| 1..n |}
channel
  { tc[i,j] | i,j:Idx • i≠j } Trans
value
  sys: (Idx $\overrightarrow{m}$ Θ) × n:**Nat** → **Unit**
  sys(mθ,n) ≡ ‖ { tra(i)(mθ(i)) | i:Idx }

Figure 14: Transaction Sequence "Chart"



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

nxt: i:Idx → Θ → **in** {tc[j,i]|j:Idx•i≠j} **out** {tc[i,j]|j:Idx•i≠j} Θ
nxt(i)(θ) ≡
  let choice = rcv ⌈⌉ snd **in**
  cases choice **of** rcv→receive(i)(θ), snd→send(i)(θ) **end end**

Annotations: The *system* is the parallel combination of *n traders*. *Traders* communicate over channels: *tc[i,j]* — from *trader i* to *trader j*. Each *trader* is modelled as a process which "goes on forever", but in steps of *next* state transitions. The *next* state transition non—deterministically (internal choice, ⌈⌉) "alternates" between expressing willingness to

Figure 15: Trader = Buyer + Seller

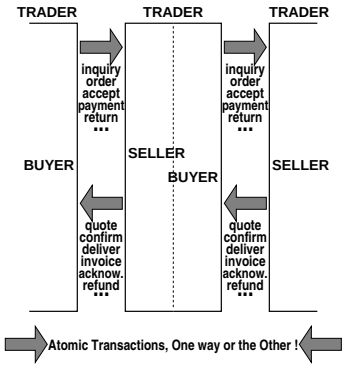
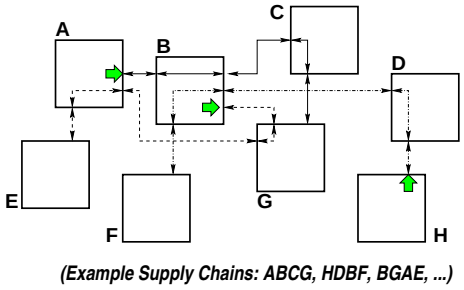


Figure 16: A Network of Traders and Supply Chains



receive, respectively desire to *send*. In "real life", ie. in the *domain*, the choice as to which transactions are taken is *non–deterministic*. And it is an *internal choice*. That is: The *choice* is not influenced by the *environment*.

receive: i:Idx $\to \Theta \to$ **in** {tc[j,i]|j:Idx•i≠j} Θ
receive(i)(θ) $\equiv$ [] {**let** msg=tc[j,i]? **in** update_rcv_state(msg,j)(θ) **end**|j:Idx}

update_rcv_state: i:Idx $\times$ Trabs $\to \Theta \to \Theta$

Annotations: *update_rcv_state* is not a protocol function. *update_rcv_state* describes the deposit of *msg* in a repository of received messages. If *msg* is a response to an earlier sent transaction, *msg_o*, then *update_rcv_state* describes the removal of *msg_o* from a repository of sent messages. *remove_set_msg* models the situation where no response (*nor*) is (ever) made to an earlier sent message. Once the internal non–deterministic choice ($\sqcap$) has been made: 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.

```
send: i:Idx → Θ → in,out {tc[i,j]|j:Idx•i≠j} Θ
send(i)(θ) ≡
  let choice = ini ⌈⌉ res ⌈⌉ nor in
  cases choice of
    ini → send_initial(i)(θ),
    res → send_response(i)(θ),
    nor → remove_received_msg(θ) end end
```

Annotations: Either a *trader*, when communicating a transaction chooses an *initial (ini)* one, or chooses one which is in *response (res)* to a message received earlier, or chooses to *not respond (nor)* to such an earlier message The choice is again non–deterministic internal. In the last case the state is updated by a *non–deterministical internal choice* (not shown) *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 more the particular *domain* at hand: Here "The Market".

```
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
```

Fourth Protocol Observation: 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**

Annotations: *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 (θ',msg)=sel_update_sell_state(θ), j=obs_trader(msg) **in**
  let (θ'',msg$'$) = response_sell_msg(msg)(θ') **in**
  tc[i,j]!msg$'$; θ'' **end end**

Annotations: *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_sell_msg* and *response_buy_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.

Discussion In this example the protocol aspect was quite "pronounced". Again we remind the reader that we have, so far only described aspects of the domain. Next we shall deal with requirements.

3.3.2 Requirements: An E–Market

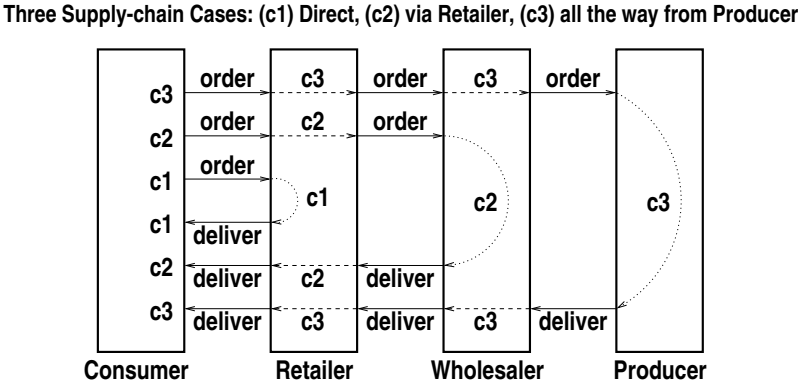
Domain Requirements

Projection Synopsis: We omit consideration of all operations and queries, ie. of any specific o, io, q and iq in respectively O, IO, Q and IQ, and focus only on the communication

between traders. We basically ignore the "content" of any transaction, and shall instead focus on automating certain sequences of transactions.

Instantiation Synopsis: Whereas the domain model of traders was a model, essentially, intrinsically, of human operations, we now try to automate as much as possible the response to received transactions. Thus, as an example: (1) If a consumer order can be delivered by the retailer — without human (retailer staff) intervention — it will be done so. (2) If a consumer order cannot be delivered by the retailer, but that retailer can re-order from a wholesaler, who can deliver — both atomic transactions without human (retailer and wholesaler staff) intervention — it will be done so. (3) And if a consumer order cannot be delivered by the retailer, but that retailer can re-order from a wholesaler, who then cannot deliver, but must re–order from producer, who can deliver — all atomic transactions without human (retailer, wholesaler and producer staff) intervention — it will be done so. Figure 17 attempts to show the three cases listed above. There might be delays, waiting times, between order receipt and delivery and/or re–ordering, across the supply–chain.

Figure 17: E–Market Supply Chain



Extension Synopsis We introduce electronic traders, agents and brokers. They permit arbitrarily wide inquiries: Potentially to all providers (retailers, wholesalers, or producers) of specified merchandise (or services), offers ("confirmations") of merchandise (or services) to all "takers" (consumers, retailers, or wholesalers), first–come–first serve ("auction"-like) orders, *&c.* These roughly sketched domain requirements are considered extensions as they might not be humanly feasible in actual domains.

Initialisation Synopsis Due to our projection we need only consider how traders, agents and brokers initially, and in an ongoing way, come to know of one another. We omit details — *"left as an exercise"*.

Interface Requirements

We omit detailed consideration of trader interfaces to the electronic market support. This is "classic" web–design !

Machine Requirements

The *availability*, *accessibility*, and *security* requirements are assumed "taken care" of by an "underlying" Internet system. We focus just on the *fault tolerance* issue: If a trader "goes out of business", or electronically "breaks down", while many transactions, from and to many other traders, are pending, then what ? Here, for example, the possible simplicity of a supply chain protocol, as indicated earlier, is at stake. A proper protocol for handling this requires back–ups, duplication, "proxies" and the like. In other words: Protocol engineering "takes over" !

3.3.3 Discussion

3.4 Health–care Systems

3.4.1 Narrative: Flow of People, Material and Information

The health–care sector consists of at least the follow stake–holders: (1) citizens (healthy or sick) C, (2) family physicians (ie. privately practising medical doctors) M, (3) pharmacies (drug stores) P, (4) community nurses N, (5) clinical test laboratories L, (6) hospitals H, (7) revalidation centres (physio–therapeutical, chiropractical, etc.) and re-convalescence homes R, (8) health insurance companies I, (9) medico–technical companies T, (10) pharmaceuticals Π, ..., (11) the National Board of Health B, and the (12) Government Ministry of Health G, etcetera.

Except for the last two, there are many instances within each of these groups of stake–holders (c, m, p, n, ℓ, h, r, i, t, and π respectively).

We can consider each player within these categories as a having a behaviour: citizen, meddoc, pharmacy, nurse, lab, hospital, centre, insurance, medico–techn, pharmaceutical, board, and government. Each behaviour evolves around a changing state: cσ:CΣ, mσ : $M\Sigma$, pσ:PΣ, nσ:NΣ, $\ell\sigma$:LΣ, hσ:HΣ, rσ:RΩ, iσ:IΩ, tσ:TΩ, $\pi\sigma$: $\Pi\Sigma$, bσ:BΣ, and gσ:GΣ

Any one of these behaviours can communicate with any other: A citizen can visit the family doctor, wait in line, be serviced (be interviewed by the doctor, analysed (tested), diagnosed, treated and observed), or give up waiting and go elsewhere (for example home, or to another doctor); or a citizen can go to a pharmacy, wait in line, be serviced (buy some over-the-counter or prescription medicine), or give up waiting and go elsewhere (for example home, or to another pharmacy); or a citizen may go hospital (either to the emergency ward, or being referred there by a physician), be registered, admitted (to a ward, being allocated a bed, etc.), serviced (the above plus: operated upon [surgery], etc.), and discharged; or a citizen may visited by a community nurse (interviewed, analysed, treated, and observed); etc. A practising physician may deposit a prescription with a pharmacy; or a practising physician may inform a hospital that a patient need be called in; etc. Pharmacies receive supplies from pharmaceuticals. Medical doctors, community nurses, pharmacies, etc., may receive instructions (rules & regulations) from the National Board of Health; and they, in

turn, regularly, or upon request, deliver statistics to that board — which in turn interacts with the ministry of health. Etcetera. There are many interaction possibilities. And with each of these there are many functionalities and hence sub–behaviours.

With a citizen one can, in the domain, associate a "virtual" patient medical journal PMJ. You should here think of a PMJ as consisting of all that which is conceivably knowable and known about that citizen's health: From cradle till current time, whether recorded (say on paper, by X–rays, ECGs, MRs, CTRs, etc.) or just remembered, whether recorded centrally or geographically widely dispersed, whether remembered by the citizen, or family of that citizen, by medical professionals, or other. A PMJ typically can be composed from patient medical records, PMR, and other. A PMR usually is related to a particular case of illness or its treatment (with the family physician, at a hospital or other). With a citizen one can also, in the domain, associate a medicine cache: All those pills and pill boxes floating around, at home, on shelves, in pockets, etc. Similar for medico–technical gadgets, other equipment and disposable: Blood sugar meter, crutches, band aid, creams, etc.

Whenever a citizen visits any of the "players" described above, the PMJ is inspected and augmented. So are the medicine (etc.) caches. In-between the citizen consumes parts or all of these caches.

3.4.2 Formalisation: Flow of People, Material and Information

We limit the presentation to basically only cover the intra and inter–behaviours of citizens (CΣ), medical doctors (private physicians, MΣ), and pharmacies (PΣ) — and then primarily wrt. PMJs and medicine (MED) caches. The Σ suffixed type names denote respective state spaces.

```
value
  c:Nat, m:Nat, p:Nat
type
  CIdx = {| 1..c |}, MIdx = {| 1..m |}, PIdx = {| 1..p |}
  CΣ, MΣ, PΣ
  CΩ = Cidx ~m~> CΣ, MΩ = Midx ~m~> MΣ, PΩ = Pidx ~m~> PΣ,
  PMJ, MED
value
  obs_PMJ: CΣ → PMJ, MΣ → CIdx → PMJ, PΣ → CIdx → PMJ
  obs_MED: CΣ → MED, MΣ → MED, PΣ → MED
channel
  { cm[i,j] | i:CIdx, j:MIdx }:(CM|MC),
  { cp[i,j] | i:CIdx, j:PIdx }:(CP|PC),
  { mp[i,j] | i:MIdx, j:PIdx }:(MP|PM), ...
value
  system: CΩ × MΩ × PΩ → Unit
  system(cω,mω,pω) ≡
    ‖ { citizen(i,cω(i)) | i:CIdx } ‖
    ‖ { meddoc(i,mω(i)) | i:MIdx } ‖
    ‖ { pharmacy(i,pω(i)) | i:PIdx } ‖ ...
```

citizen: i:CIdx × CΣ → **in**,**out** {cm[i,j]|j:MIdx},{cp[i,j]|j:PIdx},... **Unit**
citizen(i,cσ) ≡ **let** cσ′ = cime(i,cσ)⌈⌉ciph(i,cσ)⌈⌉... **in** citizen(i,cσ′) **end**

meddoc: i:MIdx × MΣ → **in**,**out** {cm[j,i]|j:CIdx},{mp[i,j]|j:PIdx},... **Unit**
meddoc(i,mσ) ≡ **let** mσ′ = meci(i,mσ)⌈⌉meph(i,mσ)⌈⌉... **in** meddoc(i,cσ′) **end**

pharmacy: i:PIdx × PΣ → **in**,**out** {cp[j,i]|j:CIdx},{mp[j,i]|j:MIdx},... **Unit**
pharmacy(i,pσ) ≡ **let** pσ′ = phme(i,pσ)⌈⌉phci(i,pσ)⌈⌉... **in** pharmacy(i,cσ′) **end**

There are c, m and p citizens, medical doctors and pharmacies. CIdx, MIdx, and PIdx are the index set over citizens, medical doctors and pharmacies — viz.: Their names and addresses. CΩ, MΩ, and PΩ are the indexed sets of states spaces for all citizens, medical doctors and pharmacies. CΩ, MΩ, and PΩ are the individual state spaces of citizens, medical doctors and pharmacies. From a citizen one can observe that citizen's PMJ. From a medical doctor and a pharmacy, given a citizen index, one can observe that citizen's PMJ as known to them. Similar for medicine caches. cm, cp, and mp abstracts the ability of citizens to communicate with medical doctors and vice versa, for citizens to communicate with pharmacies and vice versa, and for medical doctors to communicate with pharmacies and vice versa. These means of physical ("meeting up in person" or electronic) communication are abstracted in terms of indexed sets of channels where indices range over the "parties" to the communication. The system that we shall consider consists, therefore, of the parallel composition of citizen, medical doctor and pharmacy behaviours.

We model only that part of the citizen behaviour which involves medical doctors and pharmacies.

A citizen non–deterministically either interacts with a medical doctor (cime) or a pharmacy (ciph). The choice is here modelled as being made by the citizen, that is *internal* ⌈⌉.[8] Similarly for medical doctor and pharmacy behaviours.

The citizen's cime (citizen to medical doctor) behaviour either is willing to engage in any one (non–deterministically, external choice [], chosen) interaction (msg=cm[i,j]?) with a medical doctor behaviour. This models contacts made by a medical doctor (j:MIdx) to citizen i:CIdx. The medical doctor request (msg) causes the citizen to respond (cm[i,j]!rep) to the medical doctor contact. The response, rep, is a function of the contact message.

Or the citizen's cime (citizen to medical doctor) behaviour, by an internal non–deterministic choice, ⌈⌉, instead selects to contact medical doctor j:MIdx. In that contact the citizen "passes on" (cm[i,j]!cms) a projection, cms, or all of its state, cσ, to the selected medical doctor j:MIdx.

Similar for citizen to pharmacy and medical doctor to pharmacy interactions.

value
cime: i:CIdx × CΣ → **in**,**out** {cm[i,j]|j:MIdx} CΣ
cime(i,cσ) ≡
[] { **let** msg = cm[i,j] ? **in let** (rep,cσ′) = c_res_mc(msg,j,cσ) **in**
cm[i,j] ! rep ; c_upd_mc(rep,j,cσ′) **end end** | j:MIdx }
∗ ⌈⌉ **let** (j,cms) = c_res_cm(cσ) **in** cm[i,j] ! cms ; c_upd_cm(cm[i,j] ?,cσ′) **end**

[8] Technically the citizen state (cσ) is given as a parameter to (either of) the two functions, cime or a ciph. They, in turn, yield citizen state (cσ′) which are "passed" on to the continued citizen behaviour.

```
ciph: i:CIdx × CΣ → in,out {cp[i,j]|j:PIdx} CΣ
ciph(i,cσ) ≡
    [] { let msg = cp[i,j] ? in let (rep,cσ′) = c_res_pc(msg,j,cσ) in
         cp[i,j] ! rep ; c_upd_pc(rep,j,cσ′) end end | j:PIdx }
    ⌈⌉ let (j,cps) = c_res_cp(cσ) in cp[i,j] ! cps ; c_upd_cp(cp[i,j] ?,cσ) end

meci: i:MIdx × MΣ → in,out {cm[j,i]|j:CIdx} CΣ
meci(j,mσ) ≡
*   [] { let msg = cm[i,j] ? in let (rep,mσ′) = m_res_cm(msg,i,mσ) in
*        cm[i,j] ! rep ; m_upd_cm(rep,i,mσ′) end end | i:CIdx }
    ⌈⌉ let (i,mcs) = m_res_mc(mσ) in cm[i,j] ! mcs ; m_upd_mc(cm[i,j] ?,mσ) end

meph: i:MIdx × MΣ → in,out {mp[i,j]|j:PIdx} CΣ
meph(i,mσ) ≡
    [] { let msg = mp[i,j] ? in let (rep,mσ′) = m_res_pm(msg,j,mσ) in
         mp[i,j] ! rep ; m_upd_pm(rep,j,mσ′) end end | j:PIdx }
    ⌈⌉ let (j,mps) = m_res_mp(mσ) in mp[i,j] ! mps ; m_upd_mp(mp[i,j] ?,mσ) end

phci: j:PIdx × PΣ → in,out {cp[j,i]|j:CIdx} CΣ
phci(j,pσ) ≡
    [] { let msg = cp[i,j] ? in let (rep,pσ′) = p_res_cp(msg,j,pσ) in
         cp[i,j] ! rep ; p_upd_cp(rep,j,pσ′) end end | i:CIdx }
    ⌈⌉ let (i,pcs) = p_res_pc(pσ) in cm[i,j] ! pcs ; p_upd_pc(cp[i,j] ?,pσ) end

phme: i:PIdx × PΣ → in,out {mp[j,i]|j:MIdx} CΣ
phme(i,pσ) ≡
    [] { let msg = mp[i,j] ? in let (rep,pσ′) = p_res_mp(msg,j,pσ) in
         cm[i,j] ! rep ; p_upd_mp(rep,j,pσ′) end end | j:MIdx }
    ⌈⌉ let (j,pms) = p_res_pm(pσ) in mp[i,j] ! pms ; p_upd_pm(mp[i,j] ?,pσ) end
```

Each of the above three kinds of behaviours: citizens, medical doctors and pharmacies consist of a pair of behaviours. The pair "matches" in that a "player" α to player β behaviour is "mirrored" by a "player" β to player α behaviour.

The "story" told above, in detail for the citizen to medical doctor behaviour is thus archetypical of all these pairs of players (stake–holders) in the health–care sector. We listed 12 players and need thus define twelve pairs. We show only three pairs, ie. six behaviours. Instead of defining these 24 behaviours we need define one higher–order, ie. parameterised functional ("behavioural !").

We have assumed a set of functions:

α_res_$\beta\alpha$, α_upd_$\beta\alpha$, α_res_$\alpha\beta$, and α_upd_$\alpha\beta$, where the pairs (α,β) range over (c,m), (m,c), (c,p), (p,c), (m,p), and (p,m).

We will only sketch some of the functions: The four that relates to a citizen calling on the medical doctor, and the medical doctor's response — the lines marked with * above:

```
value
```

```
  c_res_cm: CΣ → j:MIdx × CM, c_res_cm(cσ) ≡ ... (j,cσ)

type
  MC = CΣ
value
  m_res_cm: MC × i:CIdx × MΣ → MC × MΣ
  m_res_mc(cσ,i,mσ) ≡ consultation(cσ,mσ)(i)

  consultation: CΣ × MΣ → i:CIdx → CΣ × MΣ
  consultation(cσ,mσ)(i) ≡
    (cσ,mσ)
    ⌈⌉
    consultation(
      let action = intv⌈⌉anal⌈⌉diag⌈⌉trea⌈⌉obse⌈⌉ ... in
      cases action of
        intv → intvw(cσ,mσ)(i), anal → analy(cσ,mσ)(i),
        diag → diagn(cσ,mσ)(i), trea → treat(cσ,mσ)(i),
        obse → obser(cσ,mσ)(i), ...  → ... end end)(i)

  intvw,analy,diagn,treat,obser,...: CΣ × MΣ → i:CIdx → CΣ × MΣ

  m_upd_cm: CM × i:CIdx × MΣ → MΣ, m_upd_cm(rep,i,mσ) ≡ ...
  c_upd_cm: MC × CΣ → CΣ, c_upd_cm(cσ′,cσ) ≡ cσ′
```

c_res_cm: The citizen chooses a medical doctor and presents an aspect (cms) of its own state (cσ).

m_res_cm: The medical doctor either ingnores the presence of the citizen (incl. the citizen giving up waiting at the medical doctor), or subjects the citizen to some variant of service: Some sequence of interviews, analyses, diagnostics, treatments and observations. During this the medical doctor m_upd_cm updates an own patient medical journal for that citizen as well as changing the citizen state. The either/or are internal non–deterministic choices (⌈⌉). The model does not express the pragmatics of who makes this choice: The medical doctor or the cisitizen, or both !

c_upd_cm: The next state of the citizen is the result, rep = cσ′, of having visited the medical doctor — replacing the state of the citizen before going to the medical doctor (ie. no projection).

3.4.3 Discussion

We have sketched — but a fragment of — a large, seemingly complex system of interacting stake–holders. We have focused, narrowly, on the interaction between a citizen (a medical doctor's patient) and a medical doctor. Other pairwise interactions follow similar "patterns". The synchronisations between these stake–holders are modelled by CSPs channel output/input mechanism: !/? [14, 15, 16]. The flow of people, material and information is modelled by the communication along the channels. The establishment and augmentation of (possibly

already established) patient medical records (and/or journals), as well as the dispensing and consumption of medicine (etc.), is modelled by changes to the citizen and medical doctor states.

We claim that, using the above "templates" for behaviour and function definitions, we can model "all" flow aspects of a health–care sector. Of course, the interesting functionalities are not definable, viz.: interview, analyse, diagnose, treatment, and observer, other than through suitable signatures and perhaps a few (axiomatic) parts of function pre/post conditions.

Hospitalisation of a citizen thus can be modelled very much like a consultation: The interviews, analyses (tests), diagnostics, treatments and observations now take place in a "larger setting" where each of these functions may be modelled as behaviours that model the physical visits, by the patient, to various wards, clinical test laboratories, operating theatres, revalidation centres (within the hospital), etcetera.

3.5 Financial Service Industry

We model only two of the players in the financial services market: Banks and securities (typically stock and bond) exchanges. Also: We do not model their interaction, that is, transfers of securities between banks and stock exchanges[9] — but the models presented lend themselves to such extensions rather easily.

3.5.1 Banking

Domain Analysis

We start out with a major analysis cum domain narrative!

Account Analysis: We choose a simple, ordinary person oriented banking domain.

(This is in contrast to for example an import/export, or an investment, or a portfolio bank domain. And it is in contrast to the many other perspectives that one could model: securities and portfolio management, foreign currency trading, customer development, etc.)

On one hand there are the clients, k:K, and on the other hand there is the bank. (We initially assume that the bank is perceived, by the clients, as a single, "monolithic thing" — although it may have a geographically widely distributed net of branch offices.) Each person or other legal entity, who is a client, may have several accounts. Each account has an identity, c:C, and is an otherwise complex quantity, a:A, whose properties will be unfolded slowly. A client, k:K, may have more than one account, but has at least one — otherwise there would be no need to talk about "a client" (but perhaps about a prospective client). (So the banking domain includes all the client accounts and the bank.) Two or more clients may share accounts.

Account Types: Accounts have types: Some are savings & loan accounts; and some are demand/deposit accounts; yet other accounts are credit (or mortgage) accounts; salary/earnings accounts, etc. With each account we associate a contract which is set up when the account is first established.

[9] Such as was done in Sections 3.2, 3.3 and 3.4.

Contract Rules & Regulations: The contract establishes rules & regulations that determine several account properties.

Example rules & regulations are *The demand/deposit account* (in question)(i) yields $y\%$ interest, and (ii) has a credit limit of ℓ currency units. (iii) When the account balance is between 0 and the (negative) credit limit, then the credit interest owed the bank is $j\%$; (iv) deposits carry interest from the day after deposit; (v) interest on a demand/deposit account is otherwise calculated as follows: ...,[10] (vi) the client is sent a statement of transactions every d days (typically *every month, or every quarter*, or for every d transactions, or some such arrangement), ... *the statement lists, in chronological order, all client as well as bank initiated transactions involving this account and as from (ie. since) the last time a statement was issued.* (vii) Fees for handling certain (or any) transactions could be as follows: account establishment fee e, statement fee s, loan repayment fee i, exceeding credit (overdraw) limit fee o_ℓ, account closure fee t, etc. The rules & regulations, also called the conditions (of the account contract), for any specific type of account, may differ from client to client, and may change over time.

The rules & regulations are set up when the account is established. Some may be changed by the client, and some by the bank — giving notice to the client. Establishing an account, changing its conditions and closing an account are examples of joint client/bank or just bank transactions.

Transactions: Depending on the account type a number of different kinds of transactions can be issued "against", ie. concerning (primarily) a specifically named, c:C, account, a:A.

- Client transactions:

  Clients can (i) deposit monies into and (ii) withdraw monies from a demand/deposit account (rather freely — and the contract may stipulate so); (iii) clients can save money in a savings & loan account (and the contract may stipulate minimum monthly savings); (iv) clients can borrow money from their savings & loan account (and the contract will undoubtedly state frequency and size limits on such loans). (v) Clients may obtain a large mortgage loan whereafter one regularly, as stipulated in the contract, (vi) repays the loan by installing — for example — three kinds of monies: (vi.1) interest on the loan (these are monies that go to a account of the bank), (vi.2) annuity on the loan (this is a quantity which is deducted from the clients' mortgage loan balance) and (vi.3) fees (again monies that go to some [other] bank account). (vii) And a transaction may produce a statement (balance) of a client account.

  A statement is a list of summaries of transactions. The listed transactions give the date and hour of the transactions, its nature[11], the amounts involved (and, in cases according to which rules & regulations they were calculated), the resulting (current) account balance, etc., etc. ! A statement also lists the transactions "executed against" the account but by the bank. See next.

- Bank Transactions:

[10] ...: here follows a detailed (pseudo-algorithmic) explanation on how interest is calculated.

[11] deposit, withdrawal, loan (withdrawal from a loan account), installation of interest, annuity and fees on a loan (repaymentment), transfer between client accounts, including salary and other payment deposits as well as payments on for example loans of other client accounts, on credit cards, etc.

The bank regularly performs transactions "against" several accounts: (viii) calculation of interests due the clients (say on demand/deposit and savings & loan accounts), and (ix) calculation of interests due the bank (say on overdrawn demand/deposit and on loan accounts). The bank may regularly inform clients as to the status of their account: (x) regular statements, (xi) reminders of loan payments (interests, annuity, fees), (xii) warnings on overdue payments, information on irregular or regular payments (say of salary) into (salary) accounts, etc. (xiii) Finally the bank may change the rules & regulations of contracts, and (xiv) may suspend client transactions on (ie. freeze) an account.

Immediate & Deferred Transaction Handling: When a transaction is issued, say at time t, some of its implications are transacted "immediately", some are deferred. Examples are: installation of interest, annuity and fees on a loan is expected to immediately lead to the update on client and bank accounts, while a transaction, to be issued by the bank, namely for a reminder to be issued, say, some period prior to a quarter later, to that client (concerning amounts of next loan repayments), is deferred. Other transactions are also deferred in relation to this example. A deferred warning transaction will be invoked if the client has not responded — as assumed — to a reminder by providing a repayment. That deferred warning transaction will be annulled if a proper repayment takes place. The warning transaction, if eventually invoked, as its time "comes up", will lead to further warning reminders as well as invocation of mora interest rates, etc. Rules & regulationss concerning these reminders and warnings, etc., are also contained in the contract.

Thus we see, on one hand, that the contract is a serious and complex document. In effect its rule & regulation conditions define a number of named routines that are applied when relevant transactions are handled (executed). These routines, in the domain, are handled either manually, semi-automatically or (almost fully) automated. The bank staff (or, in cases, perhaps even clients) who handle the manual parts of these transactions may and will make mistakes. And the semi or fully automated routines may be incorrect !

Requirements

We can summarise the analysis as follows:

- Transactions are initiated by:
  - Clients:
    * Establishment and closing of accounts
    * demand (withdrawal) and deposits of monies
    * borrowing and repayment of loans
    * transfer of monies into or out of accounts
    * request for (instantaneous or regular) statements
    * *&c.*
  - and the bank:
    * Regular calculation of yield and interest
    * regular payment of bills

    * regular issue of statements
    * reminder of loan repayments
    * warning on overdue payments
    * annual account reports
    * change in (and advice about) account conditions
    * *&c.*

- Transactions are handled by the bank:
  - immediately: certain parts of f.ex. withdrawals, deposits, repayments, etc.
  - overnight:[12] remaining parts of f.ex. above
  - deferred: issue of reminders and preparation for warnings, calculation of interests, yields, mora and fees. etc.
  - conditionally:[13] issue of warnings, etc.
- In the domain this handling may be by any combination of human and machine (incl. computer) labour.
- **Support technology** is here seen as the various means whereby transactions are processed and their effect recorded.
- Examples of **support technology** are: The paper forms, including (paper) books, used during transaction and kept as records; mechanical, electro-mechanical and electronic, hand-operated calculators; chops (used in authentication on paper forms); typewriters; computers (and hence data communication equipment).

Abstraction of Immediate and Deferred Transaction Processing: We proceed by first giving — again — a rather lengthy analysis, cum narrative, of transaction processing related concepts of a bank.

We have a situation where transactions are either "immediately" handled, or are deferred. For the domain we choose to model this seeming "distinction" by obliterating it ! Each transaction is instead deferred and affixed the time interval when it should be invoked. If a transaction is issued at time t and if parts or all of it is to be handled "immediately" then it is deferred to the time interval (t, t). There is therefore, as part of the bank, a repository of *time interval marked transaction requests*. The bank (staff, computers, etc.) now is expected to repeatedly, ie. at any time t', inspect the repository. Any transactions that remain in the repository such that t' falls in the interval of transaction requests are then to be handled "immediately". In the model we assume that the handling time is 0, but that transaction requests that are eligible for "immediate" handling are chosen non-deterministically. This models the reality of a domain, but perhaps not a desirable one!

[12] We will treat overnight transactions as deferred transactions.
[13] We will treat conditional transactions as deferred transactions.

Account Temporality: Time is a crucial concept in banking: interests are calculated over time during which the balance changes and so do the interest rates — with no synchronisation between for example these two. Because of that temporality, we shall — in the domain model — "stack" all changes (initialisations and updates) to the contractual conditions (rules & regulations) such that all such changes are remembered and with a time-stamp of their occurrence.

Likewise most other account components will be time-stamped and past component values kept, likewise time-stamped.

A Summary: We shall subsequently repeat and expand on the above while making it more precise and while also providing an emerging formal specification of a domain model.

Before we do so we will, however, summarise the above:

- There are clients, k:K, and clients may have more than one account, and accounts are identified, c:C.
- With each accountthere is a contract. The contract lists the conditions, including all the rules & regulations that shall govern the routine handling of any transaction "against" the account.
- Transactions are either client initiated such as deposit, withdraw, borrow, repayment, transfers, etc., or are bank initiated such as interest calculations, reminders, warnings, issuance of requested regular statements, etc.
- Transactions are expected handled within a certain time-interval — which may be "now" or later. For simplicity we treat all transactions as deferred (till now or later!).
- So there are transaction requests and transaction processing. The latter corresponds to the actual, possibly piecemeal, handling of transaction requests.
- And there are statements. This term — which is also a computing science and software engineering term — has here a purely banking connotation.
- And there are commands. The actual routine handling of a transaction is decribed by means of a program in a hypothetical Banking Programming Language, BaPL. Programs in BaPL are commands, and commands may be composite and consist of other commands !
- So please keep the five concepts separate: Transaction requests, transaction processing, statements, routines and commands. Their relations are simple: Transaction requests lead to the eventual execution of one or more routines, each as described by means of commands. The excution of transaction request related routines constitute the transaction (ie. the transaction processing). One kind of transaction request may be that of "printing" a client account statement.

We have given a normative overview of the structure and the logic of some base operations of typical banks.

That is: We have mentioned a number of important bank state components and hinted at their inter-relation. But we have not detailed what actions actually occur when a transaction

is "executed": what specific arithmetic is performed on account balances, what specific logic applies to conditional actions on account components, etc.

We shy away from this as it is normally not a normative property, but highly specialised: differs from bank to bank, from account to account, etc. These arithmetics and logics are properties of instantiated banks and accounts. With repect to the latter the arithmetic and logic transpire from the bank rules & regulations. The essence of the above analysis is the notion of deferred action. The consequence of this modelling decision is twofold: (i) First we are able to separate the possibly human (inter)action between clients and tellers, or between clients and 'automatic teller machines' (ATMs) from the actual "backroom" (action) processing; (ii) and then we are able to abstract this latter considerably wrt. for example the not so abstract model we shall later give of bank accounts.

There are client, k:K, account identifiers, c:C, accounts a:A, and transactions, tr:Trans. And there is the bank repository r:R. The repository contains for different time intervals (t,t′) [where t may be equal to t′] and for different client account identifiers zero, one or more "deferred" transactions (to be executed).

Each transaction is modelled as a pair: a transaction routine name, rn:Rn, and a list of arguments (values) to be processed by the routine.

We assume that (for example) client accounts, a:A, contain routine descriptions (scripts).

type
  K, C, A
  B = ({clients} $\overrightarrow{m}$ (K $\overrightarrow{m}$ C-**set**))
    $\bigcup$ ({accounts} $\overrightarrow{m}$ (C $\overrightarrow{m}$ A))
    $\bigcup$ ({bank} $\overrightarrow{m}$ R)
    $\bigcup$ ({conditions} $\overrightarrow{m}$ (C $\overrightarrow{m}$ (Rn $\overrightarrow{m}$ Routine-**set**)))
  R = (T × T) $\overrightarrow{m}$ Jobs
  Jobs = C $\overrightarrow{m}$ Trans-**set**
  Trans == mk_Trans(rn:Rn,vl:VAL*)
  Routine = /* **BaPL** Program */

Client Transactions: A client may issue a transaction, tr:Trans, w.r.t. to an account, c:C, and at time t:T. Honouring that request for a transaction the banking system defers the transaction by repositing it for execution in the (instantaneous) time interval (t,t). The client may already, for some reason or another, have a set of such reposited transactions.

Insert One Transaction:

value
  client: C × Trans → T → B → B
  client(c,trans)(t)(b) ≡ insert([(t,t) ↦ [c ↦ {trans}]])(b)

We can safely assume that no two identical:

  [(t,t) ↦ [c ↦ tsk]]

can be submitted to the bank since time passes for every one client or bank transaction.

Insertion of Arbitrary Number of Transactions: You may wish to skip the next two function definitions. They show that one can indeed express the insertion and merge of deferred transactions into the bank repository.

```
value
  insert: R ~→ B ~→ B
  insert(r)(β) ≡
    if r = [ ]
      then beta
      else
        let r′ = β(bank), (t,t′):(T×T) • (t,t′) ∈ dom r in
        let r″ =
          if (t,t′) ∈ dom r′
            then
              let bjobs = r′(t,t′), cjobs = r(t,t′) in
              r′ † [ (t,t′) ↦ merge(bjobs,cjobs) ] end
            else
              r′ ∪ [ (t,t′) ↦ cjobs ] end
        insert(r \ {(t,t′)})(β † [ bank↦ r′ ])
    end end end
```

Merge of Jobs: Client Transactions:

```
value
  merge: Jobs × Jobs ~→ Jobs
  merge(bjobs,cjobs) ≡
    if cjobs=[ ]
      then bjobs
      else
        let c:C • c ∈ dom cjobs in
        let jobs =
          if c ∈ dom bjobs
            then [ c ↦ cjobs(c) ∪ bjobs(c) ]
            else [ c ↦ cjobs(c) ] end in
        merge(bjobs † jobs,cjobs \ {c}) end end
    end
```

The Banking Cycle: The bank at any time t:T investigates whether a transaction is ("defer") scheduled [ie. "deferred" for handling] at, or around, that time. If not, nothing happens — and the bank is expected to repeat this investigation at the next time click ! If there is a transaction, tr:Trans, then it is fetched from the repository together with the time interval (t',t'') for which it was scheduled and the identity, c:C, of the client account. (c may be the identity of an account of the bank itself!)

```
value
  bank: B → T ~→ B
  bank(β)(t) ≡
    if β(bank) = [ ] then β else
    if is_ready_Task(β)(t)
      then
        let (((t′,t″),c,mk_Task(rn,al)),β′) = sel_rmv_Task(β)(t) in
        let rout:Routine • rout ∈ ((β′(conditions))(c))(rn) in
        let (β′,r) = E(c,rout)(al)(t,t′,t″)(β′) in
        bank(insert(r)(β″))(t) end end end
      else
        let t‴:T • t‴ = t + Δτ in bank(β)(t‴) end
    end end

  E: C × Routine ~→ VAL* ~→ (T×T×T) ~→ B ~→ B × R
```

The expression $\Delta\,\tau$ yields a minimal time step value.

Auxiliary Repository Inspection Functions:

```
value
  is_ready_Task: B → T ~→ Bool
  is_ready_Task(β)(t) ≡ ∃ (t′,t″):T×T • (t′,t″) ∈ dom β(bank) ∧ t′ ≤ t ∧ t ≤ t″

  sel_rmv_Task: B → T ~→ (((T×T) × C × Task) × B)
  sel_rmv_Task(β)(t) ≡
    let r = β(bank) in
    let (t′,t″):T×T • (t′,t″) ∈ dom r ∧ t′ ≤ t ∧ t ≤ t″ in
    let jobs = r(t′,t″) in
    let c:C • c ∈ dom jobs in
    let tasks = jobs(c) in
    let task:Task • task ∈ tasks in
    let jobs′ = if tasks\{task} = {} then jobs\{c} else jobs † [ c ↦ tasks\{task} ] end in
    let r′ = if jobs′ = [ ] then r\{(t′,t″)} else r † [ (t′,t″) ↦ jobs′ ] end in
    (((t′,t″),c,task),β † [ bank↦ r′ ])
    end end end end end end end end
```

Performing the execution as prescribed by the transaction, tr:Trans, besides a changed bank — except for "new" deferred transactions, result in zero, one or more new deferred transactions, trs. These are inserted in the bank repository. And the bank is expected to "re-cycle": ie. to search for, ie. select new, pending transactions "at that time"! That is: the bank is expected to handle, ie. execute all its deferred transactions before advancing the clock!

Merging the Client and the Bank Cycles: On one hand clients keep coming and going: submitting transactions at irregular, unpredictable times.

On the other hand the bank keeps inspecting its repository for "outstanding" tasks.

These two "processes" intertwine. The client_step function extends the client function. The bank_step function "rewrites" the (former) bank function:

```
value
  cycle: B ~→ B
  cycle(β) ≡ let β′ = client_step(β) ⫿ bank_step(β) in cycle(β′) end

  client_step: B ~→ B
  client_step(β) ≡ let (c,tr) = client_ch?, t = clock_ch? in client(c,tr)(t)(β)

  bank_step: B ~→ B
  bank(β) ≡
    if β(bank) = [ ]
      then β
      else
        let t = clock_ch? in
        if is_ready_Task(β)(t)
          then
            let (((t′,t″),c,mk_Task(rn,al)),β′) = sel_rmv_Task(β)(t) in
            let rout:Routine • rout ∈ ((β′(conditions))(c))(rn) in
            let (β′,r) = E(c,rout)(al)(t,t′,t″)(β′) in
            insert(r)(β″) end end end
          else β end
    end end end
```

The cycle function (internal choice) non–deterministically chooses between either a client step or a bank step.

The client step **input**s a transaction at time t from some client. This is modelled by a channel communication. Both the client and the bank steps "gets to know what time it is" from the system clock.

3.5.2 Securities Trading

"What is a Securities Industry ?": In line with our approach, we again ask a question — see the section title line just above! And we give a synopsis answer.

Synopsis: The securities industry consists of:

- the following components:
  - one or more stock exchanges,
  - one or more commodities exchanges,
  - *&c.*
  - one or more brokers,

  - one or more traders,
  - *&c.*
  - and associated regulatory agencies,

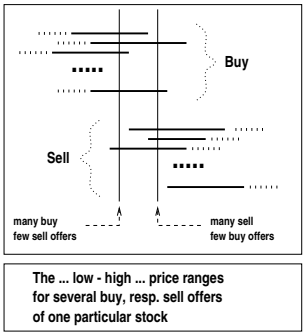
- together with all their:
  - stake-holders,
  - states,
  - events that may and do occur,
  - actions (operations) that change or predicates that inspect these states,
  - intra and inter behaviours and
  - properties of the above!

A Stock Exchange "Grand" State: Domain-wise we will just model a simple stock exchange — and from that model "derive" domain models of simple brokers and traders. Technically we model the "grand" state space as a sort, and name a few additional sorts whose values are observable in states. To help your intuition we "suggest" some concrete types for all sorts, but they are only suggestions.

type
  S, O, T, Q, P, R
  SE = (Buy × Sell) × ClRm
  Buy, Sell = S $\overrightarrow{m}$ Ofrs
  Ofrs = O $\overrightarrow{m}$ Ofr ！
  Ofr = (T×T) $\overrightarrow{m}$ (Q × (lo:P×hi:P) × ...)
  ClRm = O $\overrightarrow{m}$ Clrd | Rmvd
  Clrd = S × P × T × Ofrs × Ofrs
  Rmvd = S × T × O × Ofr
  Market = T → SE

The main (state) components of a stock exchange — reflecting, as it were, 'the market' — are the current state of stocks offered (ie. placed) for buying Buy, respectively selling Sell, and a summary of those cleared (that is bought & sold) and those removed (because the broker who placed them withdrew the offer or because the time interval of the validity of their offer elapsed). The placement of an offer of a stock, s:S, results, r:R, in the offer being marked by a unique offer identification, o:O. The offer otherwise is associated with information about the time interval, (bt,et):T×T, during which the offer is valid — an offer that has not been cleared during that time interval is to be removed from buy or sell status, or it can be withdrawn by the placing broker — the quantity offered and the low to high price range of the offer. (There may be other information (...).)

Figure 18: A "Snapshot" Stock Exchange View of Current Offers of a Single Stock



Observers — State Structure: Having defined abstract types (ie. sorts) we must now define a number of observers. Which one we define we find out, successively, as we later sketch signatures of functions as well as sketching their definition. As we do the latter we discover that it would "come in handy" if one had "such and such an observer"! Given the suggested concrete types for the correspondingly named abstract ones we can also postulate any larger number of observers — most of which it turns out we will (rather: up to this moment has) not had a need for!

value
  obs_Buy: SE → Buy, obs_Sell: SE → Sell,
  obs_ClRm: SE → ClRm
  obs_Ss: (Buy|Sell) → S-**set**
  obs_Ofrs: S × (Buy|Sell) $\xrightarrow{\sim}$ Ofrs
  obs_Q: Ofr → Q
  obs_Qs: Ofrs → Q
  obs_lohi: Ofr → P×P
  obs_TT: Ofr → T×T
  obs_O: R → O
  obs_OK: R → {ok|nok}

Main State Generator Signatures: The following three generators seems to be the major ones:

- place: expresses the placement of either a buy or a sell offer, by a broker for a quantity of stocks to be bought or sold at some price suggested by some guiding price interval (lo,hi), such that the offer is valid in some time (bt,et) interval.[14]

[14] We shall [probably] understand the buy (lo,hi) interval as indicating: buy as low as possible, do not buy at a pricer higher than hi, but you may buy when it is lo or as soon after it goes below lo. Similarly for sell (lo,hi): sell as high as possible, do not sell at a pricer lower than lo, but you may sell when it is hi or as soon after it goes above hi; the place action is expected to return a response which includes giving a unique offer identification o:O.

value
place: {buy|sell}×B×Q×S×(lo:P×hi:P)×(bt:T×et:T)×... → SE $\xrightarrow{\sim}$ SE × R

- wthdrw: expresses the withdrawal of an offer o:O (by a broker who has the offer identification).
- next: expresses a state transition — afforded just by inspecting the state and effecting either of two kinds of state changes or none!

value
wthdrw: O × T → SE $\xrightarrow{\sim}$ SE × R
next: T × SE → SE

A Next State Function: At any time, but time is a "hidden state" component, the stock exchange either clears (fclr) a batch of stocks — if some can be cleared (pclr) — or removes (frmv) elapsed (prmv) offers, or does nothing!

value
next: T × SE → SE
next(t,se) ≡
if pclr(t,se) **then** fclr(t,se) **else if** prmv(t,se) **then** frmv(t,se) **else** se **end end**

pclr: T × SE → **Bool**, fclr: T × SE → SE
prm: T × SE → **Bool**, frm: T × SE → SE

Next State Auxiliary Predicates: A batch (bs,ss) of (buy, sell) offered stocks of one specific kind(s) can be cleared if a price (p) can be arrived at, one that satisfies the low to high interval buy, respectively sell criterion — and such that the batch quantities of buy, resp. sell offers either are equal or their difference is such that the stock exchange is itself willing to place a buy, respectively a sell offer for the difference (in order to finally clear the offers).

value
pclr(t,se) ≡ ∃ s:S,ss:Ofrs,bs:Ofrs,p:P • aplcr(s,ss,bs,p)(t,se)

apclr: S×Ofrs×Ofrs×P → T×SE → **Bool**
apclr(s,bs,ss,p)(t,se) ≡
let buy = obs_Buy(se), sell = obs_Sell(se) **in**
s ∈ obs_Ss(buy) ∩ obs_Ss(sell)
∧ bs ⊆ obs_Ofrs(s,buy) ∧ ss ⊆ obs_Ofrs(s,sell)
∧ buysell(p,bs,ss)(t)
∧ **let** (bq,sq) = (obs_Qs(bs),obs_Qs(ss)) **in** acceptable_difference(bq,sq,s,se) **end end**

buysell: P×Ofrs×Ofrs → T → **Bool**

```
buysell(p,bs,ss)(t) ≡
  ∀ ofr:Ofr • ofr ∈ bs ⇒
    let (lo,hi) = obs_lohi(ofr) in p ≤ hi end
    let (bt,et) = obs_TT(ofr) in bt ≤ t ≤ et end
∧ ∀ ofr:Ofr • ofr ∈ ss ⇒
    let (lo,hi) = obs_lohi(ofr) in p ≥ lo end
    let (bt,et) = obs_TT(ofr) in bt ≤ t ≤ et end
```

Next State Auxiliary Function: We describe the result of a clearing of buy, respectively sell offered stocks by the properties of the stock exchange before and after the clearing.

Before the clearing the stock exchange must have suitable batches of buy (bs), respectively sell (ss) offered stocks (of identity s) for which a common price (p) can be negotiated (apclr).

After the clearing the stock exchange will "be in a different state". We choose to characterise here this "different state" buy first expressing that the cleared stocks must be removed as offers (rm_Ofrs). If the buy batch contained more stocks for offer than the sell batch then the stock exchange becomes a trader and places a new buy offer in order to make up for the difference. Similarly if there were more sell stocks than buy stocks. At the same time the clearing is recorded (updClRm).

```
fclr(t,se) as se′
  pre pclr(t,se)
  post
    let s:S,bs:Ofrs,ss:Ofrs,p:P•apclr(s,ss,bs,p)(t,se) in
    let (bq,sq) = (obs_Qs(bs),obs_Qs(ss)), buy = obs_Buy(se), sell = obs_Sell(se) in
    let buy′ = rm_Ofrs(s,bs,buy), sell′ = rm_Ofrs(s,ss,sell) in
    obs_Buy(se′) = if bq > sq
        then updbs(buy′,s,bq−sq,tt_buy(s,bq−sq)(t,se)) else buy′ end ∧
    obs_Sell(se′) = if bq < sq
        then updss(sell′,s,sq−bq,tt_sell(s,bq−sq)(t,se)) else sell′ end ∧
    let clrm = obs_ClRm(se) in obs_ClRm(se′) = updClRm(s,p,t,bs,ss,clrm) end
    end end end
```

Many comments can be attached to the above predicate for clearability, respectively the clearing function:

- First we must recall that we are trying to model the domain. That is: we can not present too concrete a model of stock exchanges, neither what concerns its components, nor what concerns its actions.

  The condition, ie. the predicate for clearable batches of buy and sell stocks must necessarily be loosely defined — as many such batches can be found, and as the "final clinch", ie. the selection of exactly which batches are cleared and their (common) prices is a matter for "negotiation on the floor". We express this looseness in several ways: the batches are any subsets of those which could be cleared such that any possible difference in their two batch quantites is acceptable for the stock exchange itself to take the risk

of obtaining a now guaranteed price (and if not, to take the loss — or profit!); the batch price should satisfy the lower/upper bound (buysell) criterion, and it is again loosely specified; and finally: Which stock (s) is selected, and that only exactly one stock is selected, again expresses some looseness, but does not prevent another stock (s≠s′) from being selected in a next "transition".

- There is no guarantee that the stock s buy and sell batches bs and ss and at the price p for which the clearable condition pclr holds, is also exactly the ones chosen — by apclr — for clearing (fclr), but that only could be said to reflect the "fickleness" of the "market"!

- Time was not a parameter in the clearing part of the next function. It is assumed that whatever the time is all stocks offered have valid time intervals that "surround" this time, ie. the current time is in their intervals. We shall have more to say about time later.

- Then we must recall that we are modelling a number of stake-holder perspectives: buyers and sellers of stocks, their brokers and traders, the stock exchange and the securities commission. In the present model there is no clear expression, for example in the form of distinct formulas (distinct functions or lines) that reflect the concerns of precisely one subset of these stake-holders as contrasted with other formulas which then reflect the concerns of a therefrom distinct other subset of stake-holders.

  Now we have, at least, some overall "feel" for the domain of a stock exchange. We can now rewrite the formulas so as to reflect distinct sets of stake-holder concerns. We presently leave that as an exercise!

Auxiliary Generator Functions:

value
  rm_Ofrs: S × Ofrs × (Buy|Sell) $\xrightarrow{\sim}$ (Buy|Sell)
  rm_Ofrs(s,os,busl) **as** busl′
    pre s ∈ obs_Ss(busl) ∧ subseteq(os,obs_Ofrs(s,busl))
    post if s ∈ obs_Ss(busl) **then** ∼∃ ... **else** ... **end**

3.5.3 Discussion

We have detailed two "narrow" aspects of a financial industry: How banks may choose to process client (and own) transactions, and how securities are traded. The former model is chosen so as to reflect all possibilities as they may occur in the domain, ie. in actual situations. The latter model is sufficiently "loose" to allow a widest range of interpretations, yet it is also sufficiently precise in that it casts light on key aspects of securities trading.

In this section we have not shown, as we did in several other sections, how the two infrastructure stake–holders: Banks and securities traders interact. We believe that earlier models (Sections 3.2, 3.3 and 3.4) show the reader ways of how to model such interactions.

3.6 Discussion

We have shown five examples. We used different modelling styles. We emphasised different aspects. Together these five examples illustrate that we can indeed tackle very large scale domains, yet contain their models to within human scales wrt. reading, and hence comprehension and acceptance.

4 Conclusion

We have brought some sketches of domain models of railways, logistics, E–commerce, health–care and financial services. We have not dealt with principles and techniques of abstract modelling, but see [3], nor of domain modelling, but see [4]. In proper development from reasonably comprehensive domain models one, "next", develops requirements, see [5], and then designs the software, see [1].

We have taken the liberty of referring primarily to our own recent summary work. The reason is simple: The referenced collection of lecture notes (a planned book) and papers (being submitted, one by one, for publication) constitutes a consistent and reasonably complete whole. No other, scientifically more collegial, referencing would, we seriously think, comprehend the entire field as does our own documents !

4.1 Informatics Collaboration

One reason for bringing the detailed models of Section 3 is now manifested. Many of the things that were modelled could be, and was (sketch) modelled in the, by now well–established computing science and software engineering style of formal specification. But here and there certain things were left unexplained. We now take up some of these "dangling" explanations.

4.1.1 Possibilities

- Topology:
  - Railway Net Topologies:

    To calculate train speed along the rail, when making detailed running map schedules, it is necessary to know the specific topology of each individual rail unit: Its slope (height difference, if any, between connectors), its curvature (expressed, for example by some Bezier curve parameters, and, when curved rails, the inclination [the height difference between the two rails]), as well as the actual lengths of rail units, whether they pass over bridges, through tunnels, along station platforms, etc.
  - Air–dome and Air–Corridor Topologies:

    We have not given, but will do so here, a model of an air traffic related airspace [30]:

    type
    An, A, D, Cn, C
    $AS' = ias{:}An\text{-}\mathbf{set} \times oas{:}An\text{-}\mathbf{set} \times w{:}(An \xrightarrow[m]{} (A \times D \times (An \xrightarrow[m]{} (Cn \xrightarrow[m]{} C))))$

```
  AS = {| as:AS′ • wf_AS(as) |}
value
  wf_AS: AS → Bool
```

An, A, D, Cn, and C stand for airport names, airports (including their topologies), air–domes (emphasising their topologies), air–corridor names, respectively air–corridors (emphasising their topologies). The airspace wellformedness criterion includes such considerations as:

* Air–corridors are smooth point sets, sufficiently wide and high to contain flying aircrafts,
* they are nicely glued onto air–domes,
* and these fit nicely onto airports.
* Intersecting air–corridors must satisfy a number of relations.
* *&c.*

Modelling these topological issues brings computing science "back into the realm" of classical mathematics: Here illustrated by simple uses of simple topology.

- Scheduling & Allocation:
  - Train Scheduling: Section 3.1.5, towards the end, characterised a pair of scheduling functions. We left out details on optimality — for good reasons. Such is work, not so much by software engineers, as it is of operations research analysts.
  - Logistics Route Planning: Section 3.2 did not detail transport companies' conveyor nets and time tables, and hence did not explicitly go into optimal bill–of–lading scheduling. We, as software engineers can characterise what we mean by schedules and optimality once we are told how transport companies and logistics firms wishes their optimal schedules, but we think it better as work for operations research analysts to come up with sensible criteria and perhaps clever algorithms for their subsequent implementation by software engineers.

  One can easily identify several other areas where co–design between software engineers and operations research analysts ought become a professional "standard".

- Chaos Theory and Fuzzy Systems Theory:
  - Securities Buying/Selling Strategies: Section 3.5.2 characterised possible spans of bid/ask prices that may lead to securities deals. But really the choice is "magic !" — or is there some clever use of chaos or fuzzy systems theory or other ? We believe so.
  - E–Market Auction Strategies: Section 3.3 did not venture into agents and brokers and their functions in the E–business market. But one area is that of auctions, and there are many exciting auction schemes being in place in the market today as well as being proposed [31, 32].

All this is to say that we again see an area for co–design, an area to be shared between software engineers and such applied mathematicians who study and apply theories of chaos, fuzzy sets, or rough sets.

- Time–Series Analysis *&c.*:
  - Health–care Image Analysis: Section 3.4 only mentioned the concept briefly: That of patient journals containing MR and CTR scans, X–Ray images, and ECGs. Or of medical doctors analysing such images. For a proper health–care computing and communication system to smoothly provide such images on–line, on demand, and pre–process these in support of the diagnostics work of physicians, we can already today see much co–design between medical doctors, image processing mathematicians and electronics people. But to secure the smooth on–line provision of data and analyses we suggest that software engineers be included as they, and only they, it seems, understand how to formulate the "larger picture": The "entire" health–care sector.
- Statistics *&c.*:
  - Epidemiological Statistics: We again refer to Section 3.4 — although little was intimated, it can now be done: Once we develop software and communications support for wide–area integration of "all" of a health–care sectors' "zillions" of functions, we can also, as a side–benefit monitor far more stringently illness statistics, to closely watch for possible virus disease outbreaks, hopefully long before they reach epidemic scale. Again we see an area ripe for co–development between software engineers, medical doctors, and mathematical epidemologists.

4.1.2 Discussion

This survey of relations between computer & computing science, on one hand, and mathematical modelling, on the other hand, is necessarily cursory. We believe the reader can now easily add own, relevant observations.

4.2 Continuity and Monotonicity

The well–formedness predicates on train movements, train traffic, and air space (and its traffic) were either sketched, or hinted at in Sections 3.1.4 (towards the end of that section), and 4.1 (above). They really reflect the fact that computing science and the formal specifications of software engineering "move" in a discrete world — in contrast to mathematical modelling's Newton/Leibniz world of continuous, integrable and differentiable functions. In other words: We, computing science cum software engineering, need develop calculi of continuity and monotonicity over discrete types — such a discrete maps over discrete sets and lists etc.

Take yet an example: The railway nets of Section 3.1.4 (early in that section) are later made dynamic: See the definition of train traffic (end of that section). It includes, for each "continuous time", a railway net. That is: We must make sure that rail units can be opened and closed, hence must change the state of the net, hence must make it a function of time.

But other factors also require time dependency. From time to time the railway system owner wish to further develop the net: Add new, or remove old lines, stations, tracks within these, even just rail units. Such changes must satisfy some kind of monotonicity criterion: *"Changes to the railway net must be incremental or decremental — exemplified as follows: Stations cannot be removed without first having removed the lines to and from the stations; lines can probably not be removed in their entirety (for "quick fix" repairs) when carrying traffic, hence virtual stations must be inserted around the stretch of "fix", etcetera."*

We look forward to future work on the topic of 'Continuity and Monotonicity' over discretely specified types.

4.3 Future Work

A triangle of previously separate "actors" are here being asked to collaborate: Mathematicians, computer & computing scientists and engineers. Previously engineers were never really "invited" in to join the scientists in their studies. They now need be. They are the first, and normally, unfortunately, the only ones, who "meet" the challenges of domains. They are, or become, familiar with domains. But they are not always themselves alon capable of establishing proper domain theories. For that to happen they turn to the computer & computing scientists. And these again turn to mathematicians for reasons sketched in the previous section.

- **A Challenge:** *We seriously believe that software engineers, computer & computing scientists as well as applied mathematics mathematicians need interact far more tightly: First for the purpose of carrying through actual, engineering scale developments cum experiments. Then for the purpose of conducting first graduate courses in the subject area. And, finally, to identify and carry out serious research studies in the area.*

We may have a handle on what wave propagation is, in physics. But do we really, seriously and mathematically, know what a work flow system is ? Economists may think they know main characteristics of macro and micro economics — that is: Mathematically. But have we tried to broach in a like manner the issue of understanding, mathematically, and as computer scientists, what an infrastructure, let alone an infrastructure component, is ?

This is, indeed, a fascinating world !

4.4 Acknowledgements

The author wishes to acknowledge (i) many years of collaboration with colleagues Søren Prehn, Chris W. George, and Bo Stig Hansen; (ii) recent work with students, amongst others, Hans Madsen Petersen and Li Zhu; and (iii) the deep influence of Michael Jackson's work.

4.5 Bibliographical Notes

We refer to a major source of inspiration: [33, 34, 35, 36, 37, 38, 39, 40].

Otherwise, as in the introduction, we refer to own, recent work: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

References

[1] Dines Bjørner. *Software Engineering: Theory & Practice.* (Publisher is being contacted), 2002. These Lecture Notes represent the author's *Chef d'Œvre* — the summary of more than 25 years of research, development and teaching.

[2] Dines Bjørner. Models, Semiotics, Documents and Descriptions — Towards Software Engineering Literacy. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [3, 4, 5, 6, 7, 8, 9, 10, 11].

[3] Dines Bjørner. Principles and Techniques of Abstract Modelling — Some Basic Classifications. — Towards a Methodology of Software Engineering. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 4, 5, 6, 7, 8, 9, 10, 11].

[4] Dines Bjørner. Domain Engineering — A Prerequisite for Requirements Engineering — Principles and Techniques. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 5, 6, 7, 8, 9, 10, 11].

[5] Dines Bjørner. Requirements Engineering — Some Principles and Techniques — Bridging Domain Engineering and Software Design. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 6, 7, 8, 9, 10, 11].

[6] Dines Bjørner. Healthcare Systems. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 5, 7, 8, 9, 10, 11].

[7] Dines Bjørner. E–Business. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 5, 6, 8, 9, 10, 11].

[8] Dines Bjørner. Logistics. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 5, 6, 7, 9, 10, 11].

[9] Dines Bjørner. Projects & Production: Planning, Plans & Execution. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 5, 6, 7, 8, 10, 11].

[10] Dines Bjørner. Railways Systems: Towards a Domain Theory. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 5, 6, 7, 8, 9, 11].

[11] Dines Bjørner. Financial Service Institutions: Banks, Securities Trading, Insurance, *&c.* Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [2, 3, 4, 5, 6, 7, 8, 10, 9].

[12] Chris George, Peter Haff, Klaus Havelund, Anne 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.

[13] Chris George, Anne Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[14] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), Aug. 1978.

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[16] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice–Hall, 1997.

[17] D. Bjørner and C.B. Jones. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[18] D. Bjørner and C.B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[19] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[20] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997. ISBN 0521 62348 0.

[21] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[22] I. J. Hayes. *Specification Case Studies*. Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[23] J. B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley Publishing Company, 1993.

[24] Jean-Raymond Abrial. *The B Book*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1996.

[25] Dines Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. Technical Report 12, UNU/IIST, P.O.Box 3058, Macau, 7 November 1993. Appendix — on a railway domain model — by Søren Prehn and Dong Yulin,
Published in *Proceedings from first ACM Japan Chapter Conference*, March 7–9, 1994: World Scientific Publ., Singapore, 1994.

[26] Dines Bjørner, Dong Yu Lin, and S. Prehn. Domain Analyses: A Case Study of Station Management. Research Report 23, UNU/IIST, P.O.Box 3058, Macau, 9 November 1994. Presented at the *1994 Kunming International CASE Symposium*: KICS'94, Yunnan Province, P.R.of China, 16–20 November 1994.

[27] Dines Bjørner, C.W. George, and S. Prehn. *Scheduling and Rescheduling of Trains*, chapter ???, pages ???–??? *Industrial Strength Formal Methods*, Eds.: M. Hinchey and J.P. Bowen. FACIT, Springer–Verlag, London, England, 1999.

[28] Dines Bjørner, Søren Prehn, et al. 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.

[29] Dines Bjørner, Søren Prehn, et al. 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.

[30] Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture [— an Air Traffic Control Example]. Technical Report 48, UNU/IIST, P.O.Box 3058, Macau, November 1995. Keynote paper for the *Asia Pacific Software Engineering Conference*, APSEC'95, Brisbane, Australia, 6–9 December 1995.

[31] He Minghua and Leung Ho-fung. An Agent Bidding Strategy based on Fuzzy Logic in a Continuous Double Auction. In *Systems, Man and Cybernetics Conference*, page 26 pages, 2001. Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong, P.R. China.

[32] He Minghua and Leung Ho-fung. Agents in E–Commerce: State of the Art. Technical report, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong SAR, P.R. China, 2001.

[33] Michael A. Jackson. Description is Our Business. In *VDM '91: Formal Software Development Methods*, pages 1–8. Springer-Verlag, October 1991.

[34] Michael A. Jackson. *Software Development Method*, chapter 13, pages 215–234. Prentice Hall Intl., 1994. Festschrift for C. A. R. Hoare: *A Classical Mind,* Ed. W. Roscoe.

[35] Michael A. Jackson. Problems, Methods and Specialisation. *Software Engineering Journal*, pages 249–255, November 1994.

[36] Michael A. Jackson. Problems and requirements (software development). In *Second IEEE International Symposium on Requirements Engineering (Cat. No.95TH8040)*, pages 2–8. IEEE Comput. Soc. Press, 1995.

[37] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices.* ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.

[38] Pamela Zave and Michael A. Jackson. Four dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.

[39] Michael A. Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997.

[40] Michael A. Jackson. *Problem Frames — Analysing and structuring software development problems.* ACM Press, Pearson Education. Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England, 2001.