The United Nations University

UNU/IIST

International Institute for
Software Technology

Prospects for A Viable Software Industry — Enterprise Models, Design Calculi and Reusable Modules

Dines Bjørner

November 7, 1993

UNU/IIST Report No. 12

$\boxed{\mathcal{R}}$

UNU/IIST and UNU/IIST Reports

UNU/IIST is a Research and Training Center of the United Nations University. It was founded in 1992, and is located in Macau. UNU/IIST is jointly funded by the Governor of Macau and the Governments of China and Portugal through contribution to the UNU Endowment Fund.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. advanced development projects in which software techniques supported by tools are applied,
2. research projects in which new techniques for software development are investigated,
3. curriculum development projects in which courses of software technology for universities in developing countries are developed,
4. courses which typically teach advanced software development techniques,
5. events in which conferences and workshops are organised or supported by UNU/IIST, and
6. dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research [R], Technical [T], Compendia [C] or Administrative [A]. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2001

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Prospects for A Viable Software Industry — Enterprise Models, Design Calculi and Reusable Modules

Dines Bjørner

Abstract

This paper is part speculative, part technical, and mostly discursive! The paper is also comprehensive and should be readable by a large fraction of the software community.
The conjectures of the paper are based on more than 30 years of extensive experience in computing and software systems development: in the formal, abstract specification of software system architectures, and in their stepwise implementation. The conjecture of the paper is this: *For a software house, a producer of products, to be able, in future, to compete effectively it must possess: (i) deep and broad (large scale) problem domain knowledge represented in the form of 'mathematical theories of application domains', (ii) ready access to 'software devices and mechanisms', and (iii) otherwise pursue development steadily using a large variety of 'design calculi'. The paper will define the single-quoted concepts, and will illustrate large-scale applications.*

Copyright © **1998** by UNU/IIST, Dines Bjørner

Contents

1 Target Group 1

2 Motivation 1

3 Background Analysis 2

4 Three Pivotal Concepts 3

5 Application Domain Models 3
5.1 General . . . 3
5.2 Section Structure . . . 7
5.3 Railway Systems . . . 8
5.3.1 Delineation and Expectations . . . 8
5.3.2 Railway System Terminology . . . 8
5.3.3 Excerpts of a Narrative . . . 9
5.3.4 Discussion . . . 10
5.4 Small Manufacturing Industries . . . 10
5.5 Health Care Systems . . . 11
5.6 Discussion . . . 11
5.6.1 Computational Models . . . 11
5.6.2 Technological vs. Human Components . . . 12
5.6.3 Enterprise Models . . . 12

6 Design Calculi 12
6.1 Formal Methods . . . 12
6.2 Engineering Design . . . 13
6.3 Software Professionalism . . . 13
6.4 Design Calculi Schools . . . 14
6.5 Discussion . . . 15

7 Devices and Mechanisms 17
7.1 Abstract Data Types and Modules . . . 17
7.1.1 Definition . . . 17
7.1.2 Examples . . . 17
7.2 Objects . . . 18
7.2.1 Definition . . . 18
7.2.2 Examples . . . 18
7.3 Software Device . . . 18
7.3.1 Definition . . . 18
7.3.2 Examples . . . 19
7.3.3 Discussion . . . 19
7.4 Software Mechanism . . . 19
7.4.1 Definition . . . 19
7.4.2 Examples . . . 19

7.4.3 Emulator Systems 19
7.4.4 Discussion 21

8 Conclusion **21**
8.1 Application Domain Modelling 21
8.2 Object-orientedness 22
8.3 Capability Maturity Models 22

9 Acknowledgement **22**

References **22**

A A Railway System Model **26**
A.1 Basic tracks, trains and networks 26
A.1.1 Formulae 26
A.1.2 Informal explanation 30
A.2 Train traffic and scheduling 35
A.2.1 Traffic—Basic definitions 35
A.2.2 Traffic—Auxiliary definitions 36
A.2.3 Traffic—Scheduling 38
A.2.4 Rescheduling 39

1 Target Group

This paper aims[1] at reaching software house professionals engaged in the management and development of software either on behalf of clients (turn-key project houses) or for marketing as general products (product houses).

Thus we are not aiming at for example the software staff of large end-users, engaged primarily in maintaining and slightly augmenting existing software. We see such computer and service centers primarily engaged in the composition of computing systems from ready made software developed by others, namely the ones we are aiming at.

In other words: we are targeting not only the software development management at the likes of:

1. large computer vendors: Apple, Bull, Digital Equipment Corporation, Fujitsu, Hewlett Packard, Hitachi, IBM, Olivetti, Siemens-Nixdorff, Unisys — to alphabetically mention a few,
2. and "pure" software houses: Borland, CAP-Gemini, Computer Associates, CRI Inc., DDC Intl., Lotus, Microsoft, PDC, PPU, SoftPlan, SoftTech, SRA etc.,
3. but also at a new software industry, one engaged in developing end-user application domain specific software for either one of specific such areas as:
   - (a) health care systems, incl. hospitals,
   - (b) mechanical parts, small to medium sized manufacturing industries,
   - (c) railway systems,
   - (d) agriculture (farms),
   - (e) *&c.*

2 Motivation

The motivation for this discursive paper is our concern for the software industry. This concern can be enumerated:

1. We see very few software product houses surviving comfortably.

   Many start, several linger on; some falter, few survive.
2. We see very few software (turn-key) project houses making reasonable profits.

[1]Presented at and published in proceedings of first 'ACM Japan Chapter' Conference, May 7–8, 1994; *Computers as our Better Partners* World Scientific Publisher, Singapore, pp 228–246

3. We see very few software houses, whether (turn-key) project or product houses, which deliver software:
   (a) on time,
   (b) reasonably "debugged",
   (c) meeting customers expectations,
   (d) and otherwise satisfying a number of software qualities.
4. We see no software houses which guarantee the software they deliver, that is: which are willing to cover (incurred) financial losses due to erroneous software.

In general our concern can be summed up:

5. The software industry has yet to mature.

Few will dispute these claims. Many write books about the malaises, some even propose ways out. Here, then, is yet another such "contribution": it proposes, besides other crucial factors, that software development hinge upon three pivotal concepts outlined in section 4.

3 Background Analysis

In order to propose the three pivotal concepts (Sect. 4) we must first, however, analyze:

1. Which qualities do we expect from software and its development?
2. How is software developed?
3. And: What kind of professionals does it take to develop software?

An analysis of these issues was put forth in:

(a) Dines Bjørner. Trustworthy Computing Systems: The ProCoS Experience. Published in: *14th Intl. Conf. on Software Engineering,* Melbourne, Australia, May 11-15, 1992, pp. 15–34.

(b) Dines Bjørner and Jørgen Fischer Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify"? Published in: *Intl. Conf. on Fifth Generation Computer Systems: FGCS'92,* Tokyo, June 1–5, 1992 (separate folder, pp. 191–198).

Sections 5, 6 and 7 then give some details on the three pivotal qualifications that we believe a software house must possess in order to compete successfully.

4 Three Pivotal Concepts

Section 2 drew a rather negative picture of the situation claimed predominant in todays software industry.

What, then, do we offer as our belief of a way out of this less than professional state of affairs?

In addition to, and in a profound way influencing the already outlined techniques and tools of programming, software engineering and management, we claim that three necessary conditions must be met for a software house to be able to meet tomorrows' challenges and stay prosperous in business.

These three conditions are:

1. *Application Domain Models:* that the software house possess proprietary, objectively quantifiable and qualifiable (hence mathematically supported) models of the domain of problems addressed by their software.

   Section 5 provides further details.

2. *Design Calculi:* that application domain modellers, programmers and software engineers apply, to the extent such are available, design calculi to substantiate any claim made with respect to properties of the application or its software.

   Section 6 provides further details.

3. *Software Devices and Mechanisms:* that the software house builds up a set of reusable (and proprietary) software components, here referred to as 'devices' and 'mechanisms', by means of which new applications can be readily developed.

   Section 7 provides further details.

Whereas the first and the last 'remedy' calls for proprietary means, the remaining 'cure' calls for more rapid transfer of a, by now, and in fact since many years, readily available "bouquet" of formal methods: from university research laboratories to industry.

5 Application Domain Models

5.1 General

Other terms are: 'Enterprise, or Problem Domain Modelling'.

The UNU/IIST approach to either is the same and involves a pair of Problem Domain Understanding items [1–2] before we enter into the Requirements Capture item [3] which precedes the intertwined *Software Development* issues of [4] Programming and [5] Software Engineering.

1-2: *Problem Domain Understanding.*

Before any software development is undertaken the application must be thoroughly understood. What this entails is described briefly in items 1–2 below, slightly more detailed in items 2.1 through 2.9 subsequently.

A software house which somehow cannot document its thorough command of the application domain stands at great risk to loose its share of the market. We believe, as outlined in items 1–2 below, that such an understanding can best be demonstrated through a coherent combination of both informal textual, incl. terminological descriptions and formal, mathematical descriptions. These are contained in proprietary company (reference handbook) documents. All new professional employees are first trained in understanding what these documents express before they are "let loose".

The purpose of achieving, maintaining and applying this domain knowledge is to secure the 'fit-for-purpose' product quality issue, and to serve as a basis for requirements capture and the subsequent (formal) software development, thus indirectly helping to secure further product quality issues.

1: *Informal Problem Domain Description:*

Systems development is also a linguistic "exercise". Command of one's national (i.e. natural) language is a rather definite prerequisite for trustworthy development. To serve as a means for documenting such a command as well as to serve as the ultimate interface between client and producer, informal descriptions of the application domain must be established. See steps 1.1–1.3 below.

2: *Formal Problem Domain Modelling:*

By 'models' we not only mean models (i.e. model-oriented descriptions), but also 'theory' (logical, property, law-oriented) descriptions, all expressed in a basically mathematically fashion.

3: *Informal and Formal Requirements Capture:*

This point varies from project to project within the same application domain. It expresses, based on the narrative and the models, the specific requirements that are expected from a given, i.e. desired piece of software. The Requirements Capture is expressed both formally and informally, using only the terminology established.

1.1: a **Synopsis Step** which frames the domain, brief and usually esoteric, with such (esoteric) terms being explained in parts 1.2–1.3.

1.2: a **Narrative Step** which painstakingly and thoroughly explains what the domain is "all about".

For real-time, embedded application systems, it seems that a Narrative can be advantageously decomposed into the following sub-parts:

1. **Components**: descriptions of the parts which make up the system state, naming these parts, equipping them with attributes (types) and initial values, and expressing context constraints within and among such parts.

   Examples will be given in section 5.

2. **Functions**: descriptions of the non-temporal operations which involve components and may amount to functions which change their state (i.e. value) in response to external inputs (i.e. events, se next).

3. **Behaviour** — Events and Processes: descriptions of the temporal, incl. real-time responses of the system to external input and output as well as to internal communication between processes.

4. *&c.* as outlined in items 2.4 to 2.9 below.

Examples will be given in section 5.

1.3: and a **Terminology**. In any software project it is wise to settle, from the very beginning what the meaning of the terms that are (to be) used; first those of the application domain, as here, and subsequently those of the computation field: software etc.

As the Synopsis and the Narrative are developed, terms are identified for which concise definitions are given.

The terminology developed in the initial phases of any project is then to be carefully adhered to and continuously validated and updated.

2.: Problem Domain Modelling

— oftentimes consists of several, possibly concurrently "performed" issues:

For embedded, real-time computing systems these may typically include items 2.1–2.4 below:

2.1: *Component (i.e. Data) Models.*

We remind the reader that what we are after here are descriptions of intrinsic facets of the problem domain, not of the software to be developed. Usual, so-called 'data models' seem to get entangled, right away, in (all be they, abstract) models of data models (incl. data base schemas) of the final software implementation.

What comes first: the chicken or the egg syndrome, plays a full hand here: whether to start with component models, functional laws or with behavioral laws is a question that can only be partially answered given the specific application domain, and which is finally anyway a matter of style.

What we are trying to model here could be termed the state components: their attributes (types and values), etc. But we warn the practitioner: one person's data model is another person's functional and behavioural laws. The classical duality principle of physics reappears in the computation sciences.

Object-orientedness is of concern here, not as a software concept, but as a problem domain notion. By an object we here understand something that entails notions similar to those of programming: abstract data type with multiple inheritance facets, class encapsulation,

process with input/output communication (messages, methods), etc. We are not specifically advocating object-oriented design — we always have, but under different names and with different techniques. But we are saying that only extensive experimentation iterating across the three items: components, functions and behaviour, will reveal which stylistic choices one has with respect to 'data' versus functions versus behaviours.

So: by components we may therefore understand the "things" to which we apply functions and which are yielded by such applications. That is: the "things" that we normally model in the form of data, variables, etc.

Our component models are of those of the application domain. Examples could be taken from Railway Systems: tracks, trains, time tables, staff, passengers, freight, tickets, etc. We model their individuality and their composition, incl. compositional constraints: no trains on the tracks, at certain positions, unless in the time table at approximate locations, etc.

As it turns out: many laws determine properties of composite components.

Section 5 will give an extensive example of components (incl. their "internal" laws), functions and behaviours.

2.2: *Functional Laws.*

As in the natural sciences the applications to which software is applied, be they within banking (B), insurance (I), manufacturing (M), railway (R) systems, etc., follow certain laws. Results of application of functions to components shall obey these laws.

Functional laws deal with non-temporal facets. Example laws are: (B) Amount of monies withdrawn from an account is less than or equal to deposit plus interests plus credit; (R) no two trains in the same block at any time; etc.

2.3: *Behavioural Laws.*

Behavioural laws deal with temporal (incl. Real-time) aspects. (Kirchoff law) examples are: (M) number of parts flowing into a machine (manufacturing) tool (for example: a lathe) equals the numbr of parts flowing from that machine; (R) number of trains leaving a station — over given intervals — equals the number of trains entering minus the number of trains taken out of service plus the number of the trains put into service at that station; etc.

The above reflects the *Conceptual Architecture*, while the next items are concerned with the *Physical Architecture:* the contemplated ways of implementing the entire system under design.

2.4: *System Architecture.*

In realizing a physical system, such as a banking, or a railway system, components that reflect current technological capabilities are usually inserted into the "pure" system. Examples of such "pure" components were given above for railway systems. Example of inserted components for railway systems are: switch points [which serve to share track segments], signals (or semaphores) [which serve to space trains apart], optical gates [which

serve to identify passing trains], etc. With the intrinsic components these inserted, technological components, together make up the physical architecture, and bring with them possibly both changed and additional laws.

The above items (2.1–2.4) may additionally encompass considerations of:

2.5: *Reliability, Fault Tolerance and Safety Criticality.*

Components, whether intrinsic or technological, may fail. To the extent one is able to enumerate and describe (informally and formally) failures, to that extent one may be able to express which laws must not be violated despite failures. Thus identifiable failures induce new laws.

2.6: *Dependability.*

Component failure rates allow the expression of system dependability [49]. Later: given a proposed software implementation, where the software itself is safety critical, one is then able to compute whether the total system meets expected dependability.

2.7: *Performance.*

Similarly: component, incl. software performance figures together determine system performance — a number which can then be compared to expectations, and, if not satisfactory, may then lead to design changes.

2.8: CHC/CHI *(computer human communicationinterfaces)*

Given that application domain modelling has exposed and clarified a sufficient number of system component interfaces, including those between man and machine (computer and other technological interfaces) one can then systematically tackle "all" the interfaces that require human intervention. Tackling means: propose and evaluate interface (interaction) dialogues. Dialogue design may change system architecture facets enough to warrant further system design iterations.

The individuality or [partial] totality of the above items may — during the course of modelling be subject to:

2.9: *model execution, ie. simulation.*

Thus 'concerns' (2.5–2.8) apply to items (2.1–.2.4), while simulation (2.9) applies generally.

5.2 Section Structure

In this section we wish to illustrate what we mean by *Understanding the Application Domain.*

We will give a number of illustrations.

1. Railway Systems — Subsect. 5.3

2. Small Manufacturing Industries — Subsect. 5.4
3. Health Care Systems — Subsect. 5.5

We will briefly exemplify each of these in turn. Our first example is detailed. Subsequent examples are increasingly less detailed. The idea is that we first attempt to secure the readers' understanding of what we mean by a informal and a formal description of an application domain — such as we claim to present for railway systems. Then we indicate where we would start and on what we would focus when establishing similar informal and formal models for other application domains. Throughout it is important to note that we are **not** mentioning computing at all. That is: our application domain models must, we take as a dogma, be void of any reference to the technology that help achieve some of the functions and behaviours otherwise described.

5.3 Railway Systems

5.3.1 Delineation and Expectations

For us to develop software to even the tiniest corner of railway specific systems, i.e. software that is indigenous to railway problematics, we must first understand: "what is a railway system?". The "size" of the system delineated and subject to our scrutiny depends, obviously, on our trust in the completeness and consistency of the interfaces to the other railway system parts outside the boundary of our study and the requirements at hand.

Suppose, therefore, that we were to develop *software for the dispatch of trains along specific tracks* (i.e. the base requirements), what would we first application domain study before tackling the software development problem proper? We will propose an answer to that in the next subsection.

5.3.2 Railway System Terminology

Three terms: *dispatch*, *trains* and *tracks*, formed the core of our expectations. Derived from them we then find such terms as: *station (identifier)*, *time table*, *train identifier*, *arrival time*, *departure time*, *punctual*, *delay*, *marshalling yard*, *shunting*, *by-pass*, *block*, *point*, *crossing*, etcetera. Further terms (while not high-lighting these by slanted text) are: seasonal, 12 and 3 hour time tables, (current) traffic, monitoring, controlling, setting signals, station master, cabin man, engine man, messages (incl. engine man's clipboard message), etc. These and many other terms constitute the daily operational vocabulary of railway system staff. Any software system that purports to address any subset of the operations covered by these terms had better have its developers clearly understand these terms and "all" their implications, and had better itself embody this knowledge in clearly identifiable form. That is our claim.

5.3.3 Excerpts of a Narrative

The terms form the basis for constructing a Synopsis, the Narrative and the Terminology, mentioned earlier.

Let us give tiny excerpts from the Narrative:

1. *The railway track sub-system is built up from units. Units are identifiable. Units have identifiable connectors. Examples of units are: (i) a piece of two-rail track (one in-connector and one out-connector); (ii) a point: a piece of track leading in, and two pieces of track leading out (and vice-versa), that is: one in-connector, two out-connectors; (iii) a crossing: two pieces leading in (two in-connectors), two out (two out-connectors), etc. Units compose into tracks. Connectors are means for composition. Tracks provide for paths — which we could, for example represent in terms of sequences of connectors. Units provide simple paths, with points providing two possibilities one of which is current. Crossings provide two simple paths, one current. A block is a connected set of units, and are uniquely identified. Two or more blocks may share units. Tracks and the possible setting of points, i.e. the allowable paths provided by points, together with the intended direction of traffic along units (one or bi-directional), define layouts. The railway track sub-system is a layout. A station is an identifiable layout which is an embedded part of a larger layout. No two stations overlap, i.e. have any units in common, and a railway track system has as least two stations.*

2. *A train is an identifiable entity. A train positioned on a layout occupies a connected set of units, i.e. a path. Two or more trains may not occupy overlapping blocks. A train has a velocity (including direction) (which may be zero), and possibly a non-zero acceleration. A train can be associated with a front and a rear unit. A next stopping unit is a unit which may become the next front unit. A train has a "clip-board" message which lists a sequence of next stopping units. Train on tracks amount to traffic. Traffic can be understood as set of (time-stamped) histories of where trains were, incl. are now. A time table is a specification of traffic. Actual traffic may be more-or-less punctual, more-or-less delayed (advanced) with respect to its specification. Thus a time-table denotes a possibly infinite set of traffics. Railway system staff include operational staff: engine men, cabin men, station master, &c.; service staff: ticket agents and collectors, &c.; etcetera.* It is extremely important to model their interactions with the system, already from an early stage.

3. Railways laws may involve: *A train cannot depart before it has arrived. A train cannot arrive at second station before some time after it has left first station. Assuming a "non-stack track topology" of marshalling yards, then two cars in a same train into a marshalling yeard cannot be order-reversed if they occur in a same train leaving the yard; and no car can be direction reversed in a marshalling yard. If trains cannot change moving direction over a given rail segment then two trains cannot move in opposite directions within such segments. No two trains within the same block.*

5.3.4 Discussion

Please observe that in the above narrative there was absolutely no reference to computing, hence neither to software!

5.4 Small Manufacturing Industries

In preparation for making software for small (and medium-sized) manufacturing industries we must first understand a totality of what it means to be one such industry in a context of its external facets: the components, functions, behaviour, etc. of:

1. suppliers,
2. customers,
3. legal regulations concerning:
    (a) production
    (b) environment
    (c) financing
    (d) accounting
    (e) personnel
    (f) *&c.*
4. industry standards,
5. environmental facilities,
6. transportation services,
7. financing services,
8. job market: potential (future), present and retired employees,
9. *&c.*

For any one industry one can then proceed, after understanding its context, to understand its internal aspects: the components, functions, behaviour, etc. of:

1. product research and development — the kind of things that may eventually end up in CAD/CAE subsystems,
2. product manufacturing (i.e. the production line facets) — the kind of things that may eventually end up in CAM/CIM/FMS subsystems,

   and all the usual facets of any small (etc.) business:

3. wharehousing and inventory control,
4. marketing and sales support,
5. order processing,
6. accounting, payroll, personnel, etc.
7. *&c.*

The focal point, for the domain to be qualified as 'small manufacturing industries', must be that of product development and production size and that it is being just established.

Thus our emphasis is that this application domain is much wider than when one just emphasizes for example that which is to be monitored and controlled by a CIM: computer integrated manufacturing system, or by an FMS: flexible manufacturing system. The objects of CIM's and FMS's are but a small part of what makes a manufacturing industry an interesting system.

5.5 Health Care Systems

A health care system has a major components: healthy and sick people (some of the latter also identified as patients), and health care facilities. The composition of more-or-less healthy and more-or-less sick people form the basis for defining such concepts as epidemics, etc. Health care facilities come in many forms: from rural nurses and clinics to city physicians, clinics and hospitals. Definitions of clinics and hospitals are such as to include its human and material resources: medical doctors, nurses, laboratory technicians, administrators, etc., resp. hospital beds, operating theatres, pharmacies, diagnostic equipment, analysis laboratories, etc.

Examples of functions over health care systems are: commitment, laboratory tests, patient diagnosis, etc.

Examples of behaviours of health care systems are: monitoring of patients in critical ward, surgery, hospitalization, etc.

5.6 Discussion

5.6.1 Computational Models

We have briefly touched upon the kind of things subject to our understanding of an application domain. We have "boldly" claimed that we wish to understand "all" facets. That, of course, must be taken with a "grain of salt"! What we wish to describe informally is — of course —

that which we also wish to formally model. And the reason for that is that we can only put into the computer computable subsets of that which we can model.

Put more generally: just as the natural sciences have progressed, over hundreds of years, to establish formal models of, say, physics, so we believe that computing shall eventually require similar models also of "all" man-made "systems"!

The task ahead is tremendous. It will not be done in 2–3 years per domain. It will take for as long as we wish: we will continuously wish to sharpen our modelling techniques, to refine and extend our models, and to encompass ever widening circle of (previously) contextual parts.

5.6.2 Technological vs. Human Components

The examples chosen above were selected in such a way that we could (afford to) emphasize the (albeit man-made) technological components with their seemingly strict adherence to reasonably fixed functional and behavioural laws.

Increasingly, however, human components, functions and behaviour from the point of view of psychology and cognition are, or become, part of our systems, including their perception. To claim that we can model also these facets would be unscientific. We, for one, do not believe that we can establish any definitive model with the techniques and tools today available. We are, obviously, well aware that such models are being attempted by cognition psychologists and cognition "engineers", but we are not at all sure that theirs is a right direction of work.

Thus it is that we leave it open whether any of our models, incl. the ones indicated in this paper, will ever be 'complete'!

5.6.3 Enterprise Models

What we are aiming at in application domain modelling, is software implementation un-biased *Enterprise Models* — and we refer to [22, 23, 24].

6 Design Calculi

6.1 Formal Methods

1. By a *method* is meant a set of *procedures* for *selecting* and *applying* a number of *techniques* according to some *principles* and using some *notations* and *tools* in order efficiently to construct some efficient artifact.

2. By a *formal method* is here understood a way of formulating requirements and software development documents such that calculations can be made, based on the text of the documents, whereby assertions about properties of that which the documents express can be objectively validated.
3. By a *design calculus* is understood a logical system with syntax and with a semantics which expresses rules of calculation over syntactic structures.

6.2 Engineering Design

Classical engineering deals with construction of large artifacts and uses formal designs to help establish interfaces that can help decompose the task at hand into such parts for which stable interfaces between parts can be established such that their individual development as well as the overall development can be pursued in a trustworthy manner. This trust includes the ability to manage oftentimes hundreds of workers without confusion and thereby inherent mistakes.

Typically aeronautical, chemical, civil, electrical and mechanical engineering builds on the natural sciences and relies on applied mathematics to express numerous forms of calculations. In all these disciplines engineers perform many calculations based on draft designs. Results of such calculations often necessitate redesigns. In cases where the calculi are applied to such, "novel" designs for which the validity of the underlying mathematical model may be questioned, experiments are used to confirm or reject designs.

Design calculi oriented formal methods are the only methods known to all engineering professions which significantly help increase our trust in our designs.

6.3 Software Professionalism

Software development enjoys a rather doubtful position today in being perhaps the only 'engineering' profession where is it not taken for absolutely granted that in order to be an accredited (chartered) engineers one, of course uses design calculi whereever applicable and relevant.

To us a software professional is a person who, amongst other:

1. is well-versed in a non-trivial number of design calculi,
2. applies these judiciously, i.e. whereever reasonable and with critical review of their effect,
3. keeps abreast of the literature of matured formal techniques and tools,
4. and who is genuinely interested, perhaps more in the conceptual (incl. theoretical) progress of the field of computation sciences, than in the specifics of the commercial products to which the professional applies his or her skills.

6.4 Design Calculi Schools

Alphabetically listed examples of design calculi ('formal methods') are:

1. Abel is a comprehensive method (language, proof system, design techniques) for implementing non-parallel software.

   [11] covers Abel.

2. B is a comprehensive method (language, proof system, design techniques) for implementing non-parallel software.

   [50] covers B

3. BM — for Boyer-Moore — is a proof tool for the formal verification of software properties.

   [5] covers BM.

4. ccs is a calculus for describing and analyzing parallel and distributed systems.

   [39] covers ccs

5. Cold is a family of comprehensive methods (languages, proof systems, design techniques) for implementing software.

   [17, 30] cover Cold

6. Estelle — a comprehensive method for communication systems design.

   [32] covers Estelle.

7. Eves is a comprehensive method (language, proof system, design techniques) for implementing non-parallel software.

   [14] covers Eves

8. Gypsy is a proof tool for the formal verification of software properties.

   [20, 21] cover Gypsy.

9. HOL — for Higher-Order Logic — is a proof tool for the formal verification of software properties.

   [45, 25] cover HOL.

10. Larch is a comprehensive method (set of languages, proof system (LP), design techniques) for implementing software. Larch has well established interfaces to the Modula-3, C, C++ coding languages.

    [27, 28] cover Larch

11. Lotos is a comprehensive method (language, proof system, design techniques) for implementing communications (protocol) software.

    [48, 6] cover Lotos

12. OBJ is a comprehensive method (language, proof system, design techniques) for implementing non-parallel software.

    [18, 19] cover OBJ-2

13. PROSPECTRA — for Programming through Specification Transformation — is a comprehensive method (language, proof system, design techniques) for implementing software.

    [13] covers PROSPECTRA.

14. PVS — for Predicate Verification System — is a proof tool for the formal verification of software properties.

    [37] covers PVS.

15. RAISE — Rigorous Approach to Industrial Software Engineering. — is a comprehensive method (language, proof system, design techniques) for implementing software.

    [47, 42, 3, 38, 7, 26] cover RAISE,

16. The Refinement Calculus is a comprehensive technique for the stepwise refinement of correct, sequential software.

    [40, 41] cover The Refinement calculus.

17. VDM — the Vienna Development Method. — is a comprehensive method (language, proof system, design techniques) for implementing non-parallel software.

    [33, 1, 2] cover VDM,

18. Z — for Zermelo-Frankel Set Theory. — is a comprehensive method (language, proof system, design techniques) for implementing non-parallel software.

    [29, 55, 51, 52, 54, 12, 46, 36] cover Z.

Most, if not all of the above have tool sets; some, notably RAISE, but also, to a large extent Estelle, Larch, Lotos, VDM and Z, are industrial strength.

It is sobering to observe, that as HOL is being fairly well established in the hardware community of the electronic industry for verifying chip designs, so LOTOS is being well accepted in the telecommunications industry ([10]) — unlike that of general acceptance in the more general software industry of these and other formal methods!

6.5 Discussion

The main thrust of this subsection builds on the strange fact that as a professional of the field one still has to argue that it is necessary to apply formal methods.

It is still widely held that one can do without even trying to use one or more design calculi in understanding and validating what one is actually developing.

Needless we find this situation rather disturbing. Since we are working professionally in the field we tend to surround ourselves with professionals and we tend to gravitate, when among 'strangers' of the software field towards other professionals. So we could rest calm and satisfied. The numbers are growing: most leading northern European universities is graduating large numbers of candidates who will quickly become software professionals. But our responsibility as educators and scientists require us to continue making those who have yet to "catch on" understand.

We are not saying, of course not, that design calculi are a panaceae, a "magic wand", as it were, which when applied to software development will yield just the right thing, painlessly etc. What we are saying is: that without formal methods all these 'wonderful' other things that people so dearly cling to are just not raising anyone's confidence level in the software developed in any objective way.

Object-oriented Design

It seems, however, that so-called Object-oriented Methods (OOM's) ([16, 15, 56], OMT, Objectory, HOOD), do indeed improve one or more of the software qualities — but that can only be claimed. The claims cannot be substantiated by calculated evidence.

The strength of the OOM's, as was the case with all similar informal predecessors: SADT, SA/SD, Yourdon, ..., is their reliance on graphics tools — something that most formal design calculi do not (yet) have.

It seems therefore that the essential distinction between informal, graphics-based tool methods and formal design calculi is:

- The informal, graphics-based tool methods emphasize models that are only slight abstractions of the software to be built and do not allow logic reasoning, whereas
- the formal design caculi emphasize equationally (textually) expressed, and usually very high level abstract properties of that software and thus allows for extensive logic reasoning.

We are saddened to observe that also perhaps the best of these Object-Oriented Software Engineering methods, the one based on Objectory, as presented in [15], contains few if any means for formal abstraction, refinement, and reasoning about designs. I say this with some personal regret since I do otherwise believe Objectory to be a most carefully and conscienciously designed method.

Such calculi as OBJ, RAISE and Z, in contrast, allows for both object-orientedness, a fact overlooked by most, and calculation. There are now attempts to combine the good facets of visualizing designs, such as contained in HOOD, Objectory and OMT, with the calculus aspects of OBJ, RAISE or Z.

7 Devices and Mechanisms

We introduce the notions of:

1. Abstract Data Types and Modules,
2. Objects,
3. Software Devices and:
4. Software Mechanisms.

The first two (Abstract Data Types and Modules, resp. Objects) are reasonably technical and refer primarily to programming notions. The last two (Software Devices and Software Mechanisms), the subject of this section, are more pragmatic, and primarily refer to a suitability of their applicability to specific problem domains, respectively their support.

7.1 Abstract Data Types and Modules

7.1.1 Definition

An abstract data type is usually expressed in the form of some kind of module. 'Abstract data types' is a semantic notion: a possibly infinite set of data values structured according to some definitions and a usually finite set of operations over these structures. 'Modules' is a syntactic notion: something that helps secure expressiveness while helping to avoid name conflicts and other syntactic issues. Usually a simple abstract data type is conveniently expressed by one module relying otherwise on the built-in data types of the programming (i.e. the specification, design or coding) language at hand (Booleans, integers, reals, etc.). Instantiations of an abstract data type usually leads to the creation of some value (and for imperative languages, some assignable data structure) to which the operations are valid. 'Invocation' then means to apply an operation to such data.

7.1.2 Examples

Classical examples are: Stacks, queues, binary trees, graphs.

7.2 Objects

7.2.1 Definition

An object, as in 'object-orientedness', is an abstract data type with some suitable combination of most, if not all of the following properties:

1. a type facility which allows for inheritance, preferably multiple inheritance, that is: a facility that allows for one type to be a proper subtype of another type, and possibly, as for multiple inheritance, for one type to be a subtype of two other types where the latter may define disjoint, or at least only partial properly overlapping, subtypes, and themselves most likely be subtypes of some other (super) type.[2]
2. a notion of state, i.e. of assignable variables whose values are "carried over", that is: persistent between invocations of the object.
3. and possibly, most preferably, a notion of process behaviour. Instantiation of an object thus leads to the creation of a process which while executing, i.e. "running", that is exhibiting an own, non-invocated (internal) behaviour, is able to be otherwise externally invocated — through some event or message mechanism.

7.2.2 Examples

Examples are: (1) Each bank customer may be represented inside the computer in terms of a number of multiple inheritance structured objects some of which then represent various (credit and debit, loan or deposit) accounts, etc. (2) Similarly: each telephone subscriber may likewise be represented by objects represented different services: a conventional home phone, a fax, a mobile phone, etc. (3) Finally: train engines, cars and whole trains, as well as connectors, units, tracks, layouts, etc. of a railway computing system may likewise be represented by respective multiple inheritance object structures.

7.3 Software Device

7.3.1 Definition

A software device is a usually object-oriented abstract data type which represents a sub-theory (components, functions, behaviours) of an application domain.

[2] We refer to [9, 8] for proper expositions of a notion of subtypes.

7.3.2 Examples

We refer to the examples given above, in subsection 7.2.2.

7.3.3 Discussion

Thus we closely relate the application domain modelling efforts to that of their counterpart in software devices, that is: to those facets of the problem domain which finds its way into computing.

7.4 Software Mechanism

7.4.1 Definition

A software mechanism is an abstract data type that provide for computing resource-oriented co-ordination among software devices.

7.4.2 Examples

Examples are: A Data Dictionary; a Window System; a Real-time Run-time System for specific language, for example Ada, programs and typically compiled from some given compiler: a Spread-sheet System: etc.

7.4.3 Emulator Systems

To us, the most universal form of software mechanism is an Emulator System.

To explain what an Emulator System is we "tell the following story":

First we observe that knowledge about the various general 'arithmetics': that is, operations over integers, rationals, "reals", Booleans, characters, character-strings, etc., and arrays and records, etc., over these can be encoded as logical data. So can rules governing physical, chemical, biological etc. laws. Let us call these encoded laws and rules: the meta-meta data.

Then we note that the specific 'algorithms', i.e. the particular programs we may wish to write using the base arithmetics as applied to for example certain facets of railway computing can likewise be logically data encoded: the meta-data.

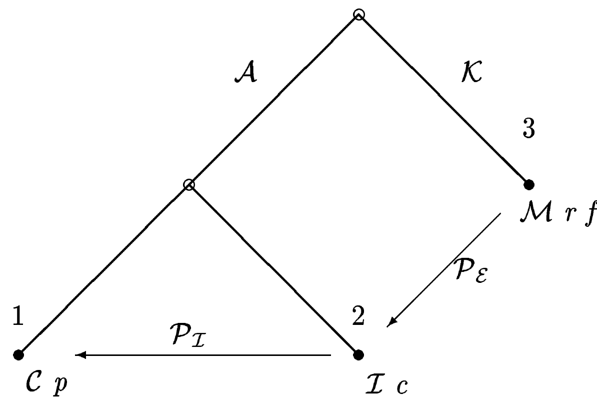
Finally we realize that also all the data: simple and structured representing, for example some of the components of our application domain, that these can also be represented, as (component) data 'facts': the data.

So everything is in the form of data (meta- or even meta-meta data).

As for Logic Programming (Prolog, CLP, CHIP, etc.) these data are now being subject to some inference machine computations.

A Partial Evaluation ([34]) applying the meta-meta data to the meta-data (call the application function $\mathcal{M}$) gives us an Emulator System which is capable of handling all proper instantiations of the application domain embodied in the meta-meta and meta-data.

So: an Emulator System is a higher-order function which when (partially evaluated, i.e.:) applied (call the application function $\mathcal{P}_{\mathcal{E}}$) to component data gives us an Interpreter for a specific instantiation of an application domain. The Interpreter can now be used to experiment with, or actually 'run' the desired end-user applications.



$\mathcal{A}$ Algorithmic, Model-oriented Spec. — (railway system) laws r

$\mathcal{K}$ Law/Knowledge-based Spec. — (railway system) components f

$\mathcal{M}$ Meta-interpreter/Inference Machine — (railway system) data structures c

$\mathcal{I}$ Interpreter — (railway system) program p

$\mathcal{C}$ Compiler — *Partial Evaluator* $\mathcal{P}$

Functionals must satisfy the laws:

- $[\![\mathcal{M}]\!]_L\ r\ f = [\![\mathcal{I}]\!]_L\ c = [\![p]\!]_L = [\![\,[\![\mathcal{C}]\!]_{L'}\ p\,]\!]_M$

- $(\llbracket \mathcal{P}_\mathcal{I} \rrbracket_\mathcal{L}\ \mathcal{M}\ r, f) \approx (\mathcal{I}, c)$
- $\llbracket \mathcal{P}_\mathcal{E} \rrbracket_\mathcal{L}\ \mathcal{I}\ d \approx p$

where L and L' are coding languages, with the meaning functions ($\llbracket \cdots \rrbracket$) taken with respect to such languages.

The $\mathcal{M}$, $\mathcal{P}_\mathcal{E}$ and $\mathcal{P}_\mathcal{I}$ are "most general" software mechanisms!

Properly 'adorned', Emulator Systems should be tailored to the specific application domains by providing for multi-media, incl. "virtual reality" functions and behaviour.

Thus a software house, specializing in specific application domains are strongly advised to research and advance develop proprietary Emulator Systems.

7.4.4 Discussion

The concept of Reusable Software has been around for some time. We believe that the 'dream' of reusability can only be attained if one properly relates reusable components to either devices or mechanisms.

8 Conclusion

Three aspects will be mentioned in this conclusion.

8.1 Application Domain Modelling

Foremost on our list is a comparison to the work at Oxford University. We refer to the Centre for Requirements and Foundations directed by Prof. Joseph A. Goguen — the leading proponent also of OBJ [18, 19].

In a number of internal notes and publications Prof. Joseph A. Goguen espouses the principles according to which that centre pursues very similar goals to those of our *Understanding the Application Domain:*

1. Research on Requirements Engineering, Joseph A. Goguen, April 1993, 8 pages.
2. Towards a Social Theory of Information, Joseph A. Goguen, (no date) 15 pages

3. The Dry and the Wet, Joseph A. Goguen; in: Eckhard Falkenberg et al. (eds.) *Information System Concepts*, pp 1-7, Elsevier, 1992 (IFIP WG 8.1 Conf., Alexandria, Egypt).
4. Requirements Engineering as the Reconciliation of Social and Technical Issues, Joseph A. Goguen and Charlotte Linde; in: Marina Jirotka et al. (eds.) *Social and Technological Issues in Requirements Engineeing*, Academic Press, 1994
5. Techniques for Requirements Elicitation, Joseph A. Goguen and Charlotte Linde; in: Stephen Fickas and Anthony Finkelstein (eds.) *Requirements Engineering '93* pages 152–164, IEEE, 1993

We find this work fascinating and hope to include its findings in our own work.

8.2 Object-orientedness

We have mentioned this topic at various point in the text. We obviously have a problem in that we cannot really reconcile the informal, almost "anti-calculi" appproaches taken by most highly verbal and publication productive advocates of Object-Oriented Software Engineering, and the similarly almost "anti-graphics" approach taken by the 'calculationists'!

We find it an interesting challenge to try reconcile these "schools" — if they can be reconciled? Maybe it is not worth many academics' time and effort to do this while they are at the same producing many design-calculi oriented candidates?

8.3 Capability Maturity Models

There is a whole dimension and approach that has not been mentioned in this discursive paper. It is that of the maturity of the software houses: are they ready for the "high-tech" technology transfer needed in order to cope with the three pivotal facets as proposed here?

We find the work of the group at the CMU operated Software Engineering Institute, SEI, led by Watts Humphrey, on CMM: Capability Maturity Models likewise fascinating — and would recommend readers to study the sobering observations made and the techniques propagated in [31, 53, 35, 43, 44].

9 Acknowledgement

The first author is grateful to Mr. Søren Prehn and Mr. Dong Yulin for their kind permission to "borrow" their work, shown in the appendix. We also gratefully acknowledge our discussions with Dr. Jan Goossenaerts, on Enterprise Models, and otherwise our colleagues at UNU/IIST.

References

[1] D. Bjørner. *Software Development. Volume I: Specification Principles — the VDM Approach.* Lecture Notes, Department of Computer Science, Technical University of Denmark, 710 pages, 1992.

[2] D. Bjørner. *Software Development. Volume II: Design Principles — the VDM Approach.* Lecture Notes, Department of Computer Science, Technical University of Denmark, 599 pages, Incomplete, 1992.

[3] D. Bjørner, A.E. Haxthausen, and K. Havelund. Formal Software Development Methods: From VDM to RAISE, and from ProCoS to LaCoS. In *InfoJapan'90, Proceedings of IPSJ 30th Anniv. Conf.* IPSJ, 2–5 October 1990.

[4] D. Bjørner, M. Mac an Airchinnigh, E.J. Neuhold, and C.B. Jones, editors. *VDM – A Formal Method at Work, Proc. of VDM-Europe Symposium '87.* Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 1987.

[5] R.S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[6] H. Brinksma. On the Design of extended LOTOS. Technical report, Ph.D. Thesis, Twente University, Enschede, The Netherlands, 1990.

[7] S. Brock and C.W. George. The RAISE Method Manual. Technical Report LACOS/CRI/DOC/3, CRI: Computer Resources International, 1990.

[8] L. Cardelli. Basic polymorphic type-checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[9] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 19.

[10] C.A.Vissers, J.van de Lagemaat, and L.Ferreira Pires. Formal Description Technques for Distributed Computing Systems — the Challenges for the 1990's. Techn.Rept. 90-31, University of Twente, Enschede, The Netherlands, June 1990.

[11] Ole-Johan Dahl. *Verifiable Programming.* C.A.R.Hoare series in Computer Science. Prentice Hall International, 1992.

[12] Antoni Diller. *Z: An Introduction to Formal Methods.* Wiley, Chichester, UK, June 1990.

[13] Bernd Krieg-Brückner et al. The PROSPECTRA Methodology and System: Uniform Transformational (Meta-) Transformation Development. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods, vol.II*, pages 363–397. Springer Lecture Notes in Computer Science, vol. 551, 1991.

[14] Dan Craigen et al. Eves: An Overview. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods, vol.I*, pages 389–405. Springer Lecture Notes in Computer Science, vol. 551, 1991.

[15] Ivar Jacobson et al. *Object-Oriented Software Engineering — A Use Case Approach.* Addison-Wesley, ACM Press, 1992.

[16] J. Rumbaugh et al. *Object-Oriented Modelling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[17] L.M.G. Feijs and H.B.M. Jonkers. *First course on COLD-K.* PhD thesis, Eindhoven Techn. Univ., The Netherlands, Nat.Lab. course, march-April, 1988.

[18] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of obj-2. In *12th Ann. Symp. on Principles of Programming*, pages 52–66. ACM, 1985.

[19] Jospeh A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. CSL Technical Report Draft, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, March 1992.

[20] Don I. Good, R.L̃. Akers, and L.M̃. Smith. Report on Gypsy 2.5. Techn.Rept. CLI-1, CLInc., Austin, Texas, Oct. 1986.

[21] Don I. Good, B.L̃. Divito, and M.K̃. Smith. Using the Gypsy Methodology. Techn.Rept. Draft CLI-2, CLInc., Austin, Texas, Jan. 1988.

[22] Jan Goossenaerts. *TIE: A Formal Language for Organisation Design.* PhD thesis, Catholic Univ. of Leuven, Leuven, Belgium, 1991.

[23] Jan Goossenaerts. The Step-wise Refinement of Enterprise Formulas. In *Intl. Conf. on Industrial Engineering and Production Management*, Mons, Belgium, 2–4 June 1993.

[24] Jan Goossenaerts and H. Yoshikawa. Enterprise Formulas, Information Infrastructures and Manufacturing Systems. In *MAPLE'93: Symposium on Manufacturing Automation Programming Language Environemtns*, Ottawa, 4–5 Oct. 1993.

[25] Michael Gordon. HOL: A Proof Generating System for Higher-Order Logic. Techn. Rept. 77, University of Cambridge, Computer Lab., 1985.

[26] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioners Series. Prentice-Hall International, 1992.

[27] J. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[28] John Guttag, James Horning, and Jeanette Wing. *Larch.* Prentice Hall International, 1992.

[29] Ian J. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1987.

[30] H.B.M.Jonkers. Introduction to COLD-K. In M. Wirsing and J.A. Bergstra, editors, *Alberaic Methods, Theory, Tools and Applications*, pages 139–2205. Springer Verlag, LNCS 394, 1989.

[31] Watts S. Humphrey. Characterizing the Software Process: A Maturity Framework. Techn. Rept. CMU/SEI-87-TR-11, Software Engineering Institute, Pittsburgh, PA, USA, June 1987.

[32] ISO. Estelle: A Formal Description Technique based on an extended state transition model. ISO Standard IS9074, ISO: Intl. Standards Org., July 1989.

[33] C.B. Jones. *Systematic Software Development — Using VDM, 2nd Edition.* Prentice-Hall International, 1989.

[34] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* C.A.R.Hoare Series in Computer Science. Prentice Hall International, 1993.

[35] Tim C. Kasses, David H. Kitson, and Watts S. Humphrey. The State of Software Engineering Practice: A Preliminary Report. Techn. Rept. CMU/SEI-89-TR-1, Software Engineering Institute, Pittsburgh, PA, USA, Feb. 1989.

[36] David Lightfoot. *Formal specification using Z.* Macmillan, 1991.

[37] Patrick Lincoln, Sam Owre, Natarajan Shankar, John Rushby, and Friedrich von Henke. Eight Papers on Formal Verification. CSL Technical Report SRI-CSL-93-04, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, May 1993. Papers 1, 3, 5 and 8 deal directly with PVS.

[38] R.E. Milne. The RSL Proof Rules. Technical Report LACOS/CRI/DOC/5, CRI: Computer Resources International, 1990.

[39] A.R.J. Milner. *Calculus of Communicating Systems.* C.A.R.Hoare series in Computer Science. Prentice Hall International, 1992.

[40] C.C. Morgan. *Programming from Specifications.* C.A.R.Hoare series in Computer Science. Prentice Hall International, 1992.

[41] C.C. Morgan, K.A. Robinson, and P.H.B. Gardiner. *On the refinement calculus.* BCS FACS Series. Springer, UK, 1993.

[42] M. Nielsen, K. Havelund, K. Ritter Wagner, and C.W. George. The RAISE Language, Method and Tools. *Formal Aspects of Computing*, 1:85–114, 1989.

[43] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability Maturity Model for Software. Techn. Rept., version 1.1 CMU/SEI-93-TR-024, Software Engineering Institute, Pittsburgh, PA, USA, Feb. 1993.

[44] Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn Bush. Key Practices in the Capability Maturity Model. Techn. Rept., version 1.1 CMU/SEI-93-TR-025, Software Engineering Institute, Pittsburgh, PA, USA, Feb. 1993.

[45] L.C. Paulson. *Logic and Computation.* Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[46] Ben F. Potter, Jane E. Sinclair, and David Till. *An Introduction to Formal Specification and Z.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.

[47] S. Prehn. From VDM to RAISE. In *[4]*, pages 141–150. Springer-Verlag, Heidelberg, Germany, 1987.

[48] G. Scollo, C.A. Vissers, and A. di Stefano. LOTOS in Pratice. In *IFIP World Congress*, pages 869–875, Dublin, Ireland, Sept. 1986. North-Holland.

[49] E.V. Sørensen, Jens Nordahl, and Niels H. Hansen. From CSP Models to Marov Models: A Case Study. *Trans. on Software Engineering*, 1992–3.

[50] Ib Holm Sørensen. The B Method. Technical report, BP Research, London, England, 1992.

[51] J.M. Spivey. *Understanding Z — a Specification Language and its Formal Semantics.* Tracts in Theo. Comp. Sci. Cambridge Univ. Press, 1988.

[52] J.M. Spivey. *The Z Notation — a Reference Manual.* Prentice-Hall International, 1989.

[53] W.L. Sweet and Watts S. Humphrey. A Method for Assessing the Software Engineering Capability of Contractors. Techn. Rept. CMU/SEI-87-TR-23, Software Engineering Institute, Pittsburgh, PA, USA, Sept. 1987.

[54] Jim C.P. Woodcock. *Using Z – Specification, Refinement and Proof.* Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, 1991. In preparation.

[55] Jim C.P. Woodcock and Martin Loomes. *Software Engineering Mathematics: Formal Methods Demystified.* Pitman Publishing Ltd., London, UK, 1988.

[56] HOOD Working Group. *HOOD User Manual and HOOD Reference Manual.* European Space Agency, Dec., resp. Sept., 1989. Issue 3.0, WME/89-353, resp. 173/JB.

A A Railway System Model

Søren Prehn[3] and Dong Yulin[4]

This appendix, which is composed, quite coarsely, of several PRaCoSy working notes, contains draft, first attempts at modelling two aspects of railways:

1. basic tracks, trains and railway networks
2. train traffic and scheduling

The first part is described by first the raw formulae, followed by a detailed informal explanation, while the second part have formulae and explanations interspersed.

A.1 Basic tracks, trains and networks

A.1.1 Formulae

Abbreviations:

```
U:unit     Uid:Unit id.  Pt:Point     L:Layout
Pa:Path    R:Route       Tk:Tracks    C:Control
S:Station  N:Network     Tnid:Trainid.
```

```
1.0   scheme
 .1     TRACK0 =
 .2       class
 .3         type U,  Pt,  Uid
 .4
 .5         value
 .6           r, l, pts : U → Pt-set,
 .7           uid : U → Uid
 .8
 .9         axiom
 .10          forall u : U •
 .11            [unit_to_point] pts(u) ≡ r(u) ∪ l(u),
 .12
 .13            [unique_point] r(u) ∩ l(u) = {},
 .14
 .15            [one_point] pts(u) ≠ {},
```

[3]UNU/IIST Research Fellow, Sept.1, 1992–Aug.31, 1994; CRI Intl., Inc., Bregnerødvej 144, DK 3460 Birkerød, Denmark; E-mail: sp@iist.unu.edu [sp@csd.cri.dk].
[4]UNU/IIST Fellow, April 15, 1993–April 15, 1995; China Railway Construction Company; Fu Xing Lu 40, Beiing 100855, P.R.of China; E-mail: dyl@iist.unu.edu.

```
.16
.17          [two_unit_connections] ∀ p : Pt • card { u | u : U • p ∈ pts(u) } ≤ 2,
.18
.19          [uuid] ∀ u, u′ : U • uid(u) = uid(u′) ≡ u = u′
.20
.21      type L,  Pa = {| (p, p′) : Pt × Pt • p ≠ p′ |}
.22
.23      value
.24        us : L → U-set,
.25
.26        Upas : U → Pa-set
.27        Upas(u) ≡
.28          { (p, p′) |
.29            (p, p′) : Pa •
.30              ((p ∈ r(u) ∧ p′ ∈ l(u)) ∨ (p ∈ l(u) ∧ p′ ∈ r(u))) ∧ ({p, p′} ⊆ pts(u))
.31          },
.32
.33        Lpas : L → Pa-set
.34        Lpas(l) ≡ { pa | pa : Pa • ∃ u : U • u ∈ us(l) ∧ pa ∈ Upas(u) }
.35
.36      value
.37        xPaUlen : Pa × U ~→ Nat
.38
.39      axiom
.40        [xPaUlen_axiom]
.41          ∀ pa : Pa • ∃ u : U • pa ∈ Upas(u) ⇒ (∃ length : Nat • xPaUlen(pa, u) = length),
.42
.43        ∀ u : U, (p, p′) : Pa •
.44          xPaUlen((p, p′), u) ≡ xPaUlen((p′, p), u) pre (p, p′) ∈ Upas(u)
.45
.46      type R = {| r : Pt* • r ≠ ⟨⟩ |}
.47
.48      value
.49        is_conn : R → L → Bool
.50        is_conn(r)(l) ≡ (∀ i : Nat • {i, i + 1} ⊆ inds r ⇒ (r(i), r(i + 1)) ∈ Lpas(l))
.51    end

2.0  scheme
 .1    TRACK1 =
 .2    extend TRACK0 with
 .3      class
 .4        type Tk
 .5
 .6        value
 .7          layout : Tk → L,
 .8          Tkpas : Tk → Pa-set
 .9
 .10       axiom
 .11         [known_paths] ∀ tk : Tk • Tkpas(tk) ⊆ Lpas(layout(tk)),
 .12
 .13         [single_path]
```

```
.14             ∀ tk : Tk •
.15               ∀ u : U •
.16                 u ∈ us(layout(tk)) ⇒ (∃ p, p′ : Pt • (Upas(u) ∩ Tkpas(tk)) ⊆ {(p, p′)})
.17
.18         value
.19           progress : Pa → Tk → Bool
.20           progress(pa)(tk) ≡ pa ∈ Tkpas(tk),
.21
.22           all_routes : Pt × Pt → Tk → R-set
.23           all_routes(p, p′)(tk) ≡
.24             { r | r : R • is_conn(r)(layout(tk)) ∧ r(1) = p ∧ r(len r) = p′ },
.25
.26           route_open : R → Tk → Bool
.27           route_open(r)(tk) ≡
.28             (∀ i : Nat • {i, i + 1} ⊆ inds r ⇒ (r(i), r(i + 1)) ∈ Tkpas(tk)),
.29
.30           all_open_routes : Pt × Pt → Tk → R-set
.31           all_open_routes(p, p′)(tk) ≡
.32             { r | r : R • r ∈ all_routes(p, p′)(tk) ∧ route_open(r)(tk) },
.33
.34           route_L : R → Tk → Nat
.35           route_L(r)(tk) ≡
.36             let (p, p′) = (hd r, hd tl r), u : U • u ∈ us(layout(tk)) ∧ (p, p′) ∈ Upas(u) in
.37               route_L(tl r)(tk) + xPaUlen((p, p′), u)
.38             end
.39             pre len r > 1 ∧ is_conn(r)(layout(tk)),
.40
.41           shortest_routes : Pt × Pt → Tk → R-set
.42           shortest_routes(p, p′)(tk) ≡
.43             let rs = all_routes(p, p′)(tk) in
.44               { r | r : R • r ∈ rs ∧ ∼ (∃ r′ : R • r′ ∈ rs ∧ route_L(r′)(tk) < route_L(r)(tk)) }
.45             end
.46       end

3.0   scheme
 .1     TRAIN =
 .2       extend TRACK1 with
 .3         class
 .4           type Tnid,  C
 .5
 .6           value
 .7             enter : U × C ~→ C,
 .8             leave : U × C ~→ C,
 .9             occupied : U × C → Bool,
.10             nsu : Tnid × U × U × C → U × C,
.11             train : Tnid × R × C → C,
.12             can_enter : U × C → Bool,
.13             can_leave : U × C → Bool
.14
.15           axiom
.16             forall u, u′, n : U, c, c′ : C, tnid : Tnid, r : R, l : L •
```

```
.17          [enter_ax]
.18            occupied(u, enter(u′, c)) ≡ u = u′ ∨ occupied(u, c) pre can_enter(u′, c),
.19
.20          [leave_ax]
.21            occupied(u, leave(u′, c)) ≡ u ≠ u′ ∧ occupied(u, c) pre can_leave(u′, c),
.22
.23          [nsu_ax]
.24            let (u′, c′) = nsu(tnid, u, n, c) in
.25              (∀ u″ : U • occupied(u″, c) = occupied(u″, c′)) ∧
.26              (u ≠ u′ ⇒ can_leave(u, c′) ∧ can_enter(u′, c′) ∧ {u, u′} ⊆ us(l))
.27            end ≡
.28              true
.29              pre occupied(u, c) ∧ {u, n} ⊆ us(l),
.30
.31          [train_ax]
.32            train(tnid, r, c) ≡
.33              if len r ≤ 1 then
.34                c
.35              else
.36                let (p, p′) = (hd r, hd tl r), uu : U • p ∈ pts(uu), nn : U • p′ ∈ pts(nn) in
.37                  let (uu′, c′) = nsu(tnid, uu, nn, c) in
.38                    if uu = uu′ then
.39                      train(tnid, r, c′)
.40                    else
.41                      train(tnid, tl r, leave(uu, enter(uu′, c′)))
.42                    end
.43                  end
.44                end
.45              end,
.46
.47          [can_enter_ax] can_enter(u, c) ⇒ ∼ occupied(u, c),
.48
.49          [can_leave_ax] can_leave(u, c) ⇒ occupied(u, c)
.50      end

4.0   scheme
 .1     NETWORK =
 .2       extend TRACK1 with
 .3         class
 .4           type S, N, Sid
 .5
 .6           value
 .7             xSs : L → S-set,
 .8             xLS : S → L,
 .9             xIdS : S → Sid,
.10             xLN : N → L
.11
.12           axiom
.13             [at_least_one_path] ∀ s : S • card { pa | pa : Pa • pa ∈ Lpas(xLS(s)) } ≥ 1,
.14
.15             [unique_station] ∀ s, s′ : S • s ≠ s′ ⇒ us(xLS(s)) ∩ us(xLS(s′)) = {},
```

.16
.17 [at_least_two_stations] ∀ n : N • **card** { s | s : S • s ∈ xSs(xLN(n)) } ≥ 2,
.18
.19 [ssid] ∀ s, s′ : S • xIdS(s) = xIdS(s′) ≡ s = s′
.20 **end**

A.1.2 Informal explanation

We consider that track layout of a railway system is made up of connected "units", having two sets of points, with the intuition that a train can move from a point in one of the sets to a point in the other set. The units in a complete layout connects by having non-disjoint points. This could be illustated by the following

where there are four units

- $(\{p0\}, \{p1, p4\})$
- $(\{p1\}, \{p2\})$
- $(\{p5\}, \{p6\})$
- $(\{p4, p6\}, \{p7\})$

TRACK0

Unit and Point

From the intuition, unit is composed of two sets of point. We specify the model TRACK0 as follows:

Two types of *U*, reprsenting unit, and *Pt*, representing point are given as sorts since we do not want to say anything about how *U* and *Pt* are represented (we may elaborate them later). For every *U*, we should assign a *Uid* to distinguish them. This is done by defining *Uid*.

Defining three functions: *r, l, pts*, when applied to a *U*, they all return a *Pt* set. The *r, l* is to decompse the *U* into two differents sets of *Pt* set, and *pts* is the union of these two sets(see Introduction). Defining a function *uid*, when applied to a *U*, it returns a *Uid*.

Then in axioms, we express the properties of the value *r, l, pts* and *uuid*:

- *unit_to_point* states the relation between *U* and *Pt*: for all *U u*, *pts* applied to *u* must be equivalent to r(u)∪l(u).

- *unique_point* says that for all U u, the intersection of the two sets of Pts must be empty set. This means that the two sets of Pts have no common elements.
- *one_point* says that for all U u, the union of the two sets of Pts must not be empty set. This means a U must have one point at least.
- *two_unit_connection* says for all Pt p, the number of elements, contained in the set of us of type U such that Pt is a member of $pts(u)$, less than or equal to 2. This means that one point cannot belongs to more than two Us.
- *uuid* says for all u and u′ of type U, if their *uid* is same, then u should be same as u'.

Layout and Path

Then we introduce Layout L and Path Pa here. In our models, the L is defined as a sort, and Pa is defined as a subtype of point product, which represents the type of those pairs (p,p′) of type $Pt \times Pt$ where p is different from p′.

We specify the properties of L and Pa via explicit function definition except *us*.

- The *us* is defined as a function that decomposes L into U set.
- The *Upas* is defined as a function that decomposes U into Pa set. For all U u, *Upas* applied to a u of type U must be equivalent to the set of (p,p′) where (p,p′) is of type Pa such that $\{p,p'\}$ is a subset of $pts(u)$.
- *Lpas* decomposes L into Pa set. For all L l, *Lpas* applied to l of type L must be equivalence to the set of pa where pa is Pa and there exists u of type U such that U is a member of $us(l)$ and pa is a member of $Upas(u)$.

Length

For every path in a unit, we can get its length. For doing this, we define a partial function *xPaUlen*, when applied to $Pa \times U$, it returns a **Nat** which represents the length of the Pa.

The following two axioms, which are predicative, are used to express the properties of *xPaUlen*:

- The first one *xPaUlen_axiom* says: for all Pa pa, if there exists a U u that makes pa a member of $Upas(u)$, then there should exist a nature number equal to *xPaUlen*. This means that we cannot get a length from a pa unless the pa belongs to a U.
- The second one says that the length of a pa will keep the same length no matter what the direction of the pa is.

Route

Now we introduce route R. From the intuition, we think a train should progress from one point to another point. The type of R is the subtype of those Pt list where the Pt list is not empty.

We specify the property of *R* via defining curried function *is_conn*: when applied to *R, L*, it returns a **Bool**. A *R* is connected if any pair of point is a member of *Lpas(l)*.

TRACK1

Track

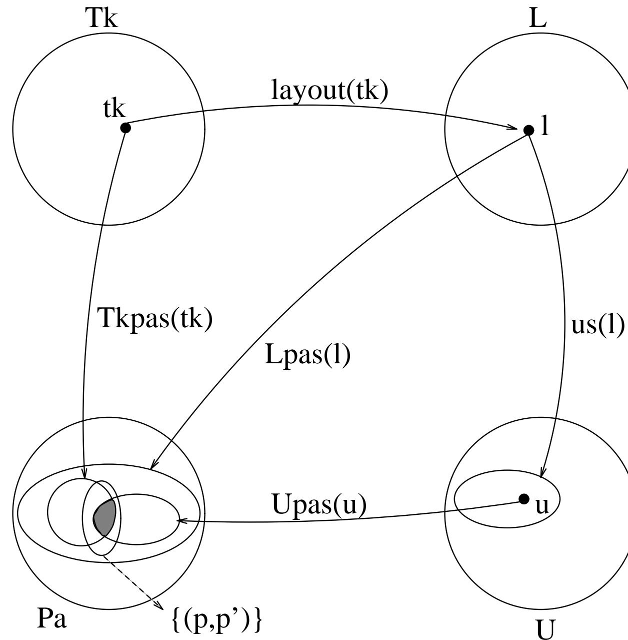
We define the sort *Tk* standing for track.

Through functions: *layout* and *Tkpas*, we define the relation between *Tk* and *L*, and decompose *Tk* into *Pa* set, respectively.

The following two axioms are predicative:

- The first one, *known_paths*, says that for all *Tk tk, Tkpas(tk)* is a subset of *Lpas(layout(tk))*. This means that any *pa* in *Tk* should also be contained in the *L*.
- The second one, *single_path*, says that for all *Tk tk* and for all *U u* , if u is a member of *us(layout(tk))*, then there should exist *Pt p* and *p′* such that the intersection of *Upas(u)* and *Tkpas(tk)* is a subset of $\{(p,p\prime)\}$.

So far, we can illustrate the relations among *Tk, L, U* and *Pa* as in the diagram shown on top of the next column:



Value Definition

The following explicit definitions of functions are explained as follows:

- *progress*: curried function. When applied to Pa and Tk, *progress* must be equivalent to $pa \in Tkpas(tk)$.
- *all_routes*: curried function. When applied to $Pt \times Pt$ and Tk, it must be equivalent to a subtype of r where r is R such that *is_conn(r)(layout(tk))* holds, the first element of r is p and the last element of r is $p\prime$.
- *route_open*: curried function. When applied to R and Tk, if the length of r greater than 1, then any pair of Pt in the r is a member of *Tkpas(tk)*.
- *all_open_routes*: curried function. For all $Pt \times Pt$ and Tk, *all_open_routes* applied to *(p,p′)(tk)* must be equivalent to a subtype of r where r is type of R such that r is a member of *all_routes(p,p′)(tk)* and *route_open(r)(tk)* holds. This means that from a Pt p to another Pt $p\prime$, there should have a R set.
- *route_L*: curried function. For these R and Tk which **len** r is greater than 1 and is connected, *route_L* applied to r and tk must be equivalent to the length of the first pa plus the length of the remaining r. From this function, we can get the length of a R.
- *shortest_routes*; curried function. For all $Pt \times Pt$ and Tk, *shortest_routes* applied to *(p,p′)(tk)* must be equivalent to a subtype of r where r is of type R such that r is a member of *all_routes(p,p′)(tk)* and there do not exist R $r\prime$ where $r\prime$ is a member of *all_routes(p,p′)(tk)* such that the length of $r\prime$ is less than the length of r.

TRAIN

This scheme is a extension of *TRACK1*. In it, we introduce *Tnid* which represents Train, and C which is Control that has a lot we have to specify later. These two types we define as sorts.

For a U, we can know whether it is occupied at certain C. And also it can be entered or left.

We define the following seven functions:

- *enter* and *leave* which return the c associated with a paticular u.
- *occupied* which checks whether a u is occupied.
- *nsu* which returns a pair of u and c when a *Tnid* progress along a pair of U u and U n in a certain c.
- *train* which associates a *tnid* with a r in a certain c.
- *can_enter, can_leave* which checks whether a u can be entered and left at a certain c.

The axioms express the properties of these functions.

- *enter_ax*: When a train enters $U\ u\prime$, if the $U\ u$ equals the entered $U\ u\prime$, then *occupied* is **true**. Otherwise occupied is applied to the pair of u and c. This only applies when pre-condition *can_enter(u′, c)* holds.
- *leave_ax*: When a train leaves $u\prime$, if $u \neq u\prime$ holds, then *occupied(u,c)*. This only applies when pre-condition *can_leave(u′, c)* holds.
- *nsu_ax*: This axiom expresses that when a *tnid* moves from $U\ u$ to $U\ n$ at the c, the next stop unit is $u\prime$, then for any $u\prime\prime$, *occupied(u′′,c)=occupied(u′′,c′)* holds and u not equal with $u\prime$ implies *can_leave* holding for *(u,c′)*, *can_enter* holding for *(u′,c′)* and $\{u, u\prime\} \subseteq us(l)$.
- *train_ax*: the right hand is an if expression with two branches:
  - if the length of r equal with and less than 1, then c.
  - if the length of r greater than 1, then a train progress from $U\ uu$ to $U\ nn$ where p and $p\prime$ are the first pair of r, and also p is a member of *pts(uu)*, $p\prime$ is a member of *pts(nn)*, and the next stop U is $uu\prime$ in $c\prime$. if $uu{=}uu\prime$, then *train(tnid, r. c′)*. Otherwise the *tnid* leaves uu and enters $uu\prime$ in $c\prime$. Then it progresses along the left r. This is expressed by *train(tnid,* **tl** *r, leave(uu, enter(uu′,c′)))*
- *can_enter_ax*: For all u and c, *can_enter* holding for *(u,c)* implies *occupied* not holding for *(u,c)*.
- *can_leave_ax*: For all u and c,*can_leave* holding for *(u,c)* implies *occupied* holding for *(u,c)*.

Network

Then we further introduce station S, Station id. *Sid* and network N which are defined as sort. What we want to specify are the following:

- A layout can be decomposed into a set of station.
- A station is a layout.
- Every station has a unique station id..
- The network is also a layout.

We elaborate the above via defining function *xSs, xLs, xIDs, XLN*;

- *at_least_one_path* means for all $S\ s$, it has more than one pa of type Pa.
- *unique_station* means the different Ss has no common Us.
- *at_least_two_stations*: means for all N, it must has more than two Ss.
- *ssid*: means every station has an unique station id.

A.2 Train traffic and scheduling

The *traffic* in a train system, i.e. within a given *layout*, can be (partially) described by the position of trains within the layout, over time. A detailed description, based on continuous (real) time is perhaps warranted, but at least for now I'll sidestep quite a few details.

The basic problem in planning train traffic, aside for physical details like acceleration & braking power, maximum speed vs. curvature, &c., is to make sure that trains can progress without conflicts. In relation to the model of layouts and *tracks* based on units connected at points, and with tracks having at most one path trough any unit, the basic scheduling problem is to avoid conflicts over access to units. The basic events of interest therefore seems to be trains entering and leaving units.

This note attempts to view time in terms of arbitrary slices. Within a slice, no train can progress more than one unit. We do not know whether the discrete time points are based on a fixed clock or not; all we know is that the time slices are fine-grained enough to make any movement between adjoining units obervable.

In this simple scheme of things, the position of a train can therefore be described by the *path* in a unit it is currently at, and traffic as evolving over time as a sequence of positions of active trains, with each set of positions being a snapshot, between two time-slices.

An important implication of this model is that a train can only be in one unit a a time. Complex crossing, e.g. in shunt-yards, may not be able to support that!

Formation and dismantling of trains are here implicitly captured quite coarsely: train ids may merely come and go! It may be advisable to have a particular train make only one journey per sequence[5]!

```
TPs : Train Positions
HS  : History Sequence
HSu : History Sequence with unique journeys
TkS : Track Sequence
```

A.2.1 Traffic—Basic definitions

5.0 **scheme**
.1 TRAFFIC =
.2 **extend** TRAIN **with**
.3 **class**
.4 **type**
.5 $H = \{| \ i : \mathbf{Nat} \bullet i > 0 \ |\}$,
.6 $TPs = \{| \ m : Tnid \ \overrightarrow{m} \ Pa \bullet m \neq [\,] \ |\}$,
.7 $HS = \{| \ hs : TPs^* \bullet hs \neq \langle\rangle \ |\}$,
.8 $HSu = \{| \ hs : HS \bullet uniq_journeys(hs) \ |\}$
.9

[5]This may be seen as saying that a train identifier is expected to include more specifics than "Wednesday Shaghai Express".

```
.10        value
.11          /∗ Train positions are ok if they are unique and currently enabled ∗/
.12          is_ok_TPs : TPs → Tk → Bool
.13          is_ok_TPs(tps)(tk) ≡ card dom tps = card rng tps ∧ rng tps ⊆ Tkpas(tk),
.14
.15          /∗ a history sequence is ok if trains progress at most one step per slice∗/
.16          /∗ and for each slice there is a track of the layout supporting it ∗/
.17          is_ok_HS : HS → L → Bool
.18          is_ok_HS(hs)(l) ≡
.19            (
.20              ∀ i : H •
.21                (
.22                  {i, i + 1} ⊆ inds hs ⇒
.23                  (
.24                    ∀ t : Tnid •
.25                      t ∈ (dom hs(i) ∩ dom hs(i + 1)) ⇒ adj(hs(i)(t), hs(i + 1)(t))
.26                  )
.27                ) ∧
.28                (∃ tk : Tk • (layout(tk) = l ∧ rng hs(i) ⊆ Tkpas(tk)))
.29            ),
.30
.31          /∗ any train have at most one contiguous (sub−) sequence of time points ∗/
.32          uniq_journeys : HS → Bool
.33          uniq_journeys(hs) ≡
.34            (
.35              ∀ t : Tnid •
.36                let js = { i | i : H • i ∈ inds hs ∧ t ∈ dom hs(i) } in
.37                  ∀ j : H • j ∈ js ∧ j + 1 ∉ js ⇒ ∼ (∃ j′ : H • j′ > j ∧ j′ ∈ js)
.38                end
.39            ),
.40
.41          /∗ two paths are adjoining iff they have at least one point in common ∗/
.42          adj : Pa × Pa → Bool
.43          adj((p1, p1′), (p2, p2′)) ≡ {p1, p1′} ∩ {p2, p2′} ≠ {}
.44
.45        type TkS = {| tks : Tk∗ • tks ≠ ⟨⟩ |}
.46
.47        value
.48          /∗ a sequence of tracks is ok if each track setting is of the given layout ∗/
.49          is_ok_TkS : TkS → L → Bool
.50          is_ok_TkS(tks)(l) ≡ (∀ tk : Tk • tk ∈ elems tks ⇒ layout(tk) = l)
.51      end
```

A.2.2 Traffic—Auxiliary definitions

Based on these basic definitions we may introduce various useful observation functions, which should be self-explanatory. Note that a notion of route based on paths rather than points has been introduced. I think this is perhaps more useful.

```
6.0  scheme
 .1    TRAFFIC_PROPS =
 .2      extend TRAFFIC with
 .3        class
 .4          type
 .5            /* maybe better than Pt−list */
 .6            Rt = {| r : Pa* • is_conn(r) |}
 .7
 .8          value
 .9            is_conn : Rt → Bool
 .10           is_conn(r) ≡ (∀ i : Nat • {i, i + 1} ⊆ inds r ⇒ adj(r(i), r(i + 1)))
 .11
 .12         value
 .13           Tnis : Tnid → HS → H-set
 .14           Tnis(t)(hs) ≡ { i | i : H • i ∈ inds hs ∧ t ∈ dom hs(i) },
 .15
 .16           has_journey : Tnid → HS → Bool
 .17           has_journey(t)(hs) ≡ Tnis(t)(hs) ≠ {},
 .18
 .19           Tnpas : Tnid → HS → Pa-set
 .20           Tnpas(t)(hs) ≡ { hs(i)(t) | i : H • i ∈ Tnis(t)(hs) },
 .21
 .22           starti : Tnid → HSu ~→ H
 .23           starti(t)(hs) ≡
 .24             let i : H • i ∈ Tnis(t)(hs) ∧ i − 1 ∉ Tnis(t)(hs) in i end
 .25             pre has_journey(t)(hs),
 .26
 .27           endi : Tnid → HSu ~→ H
 .28           endi(t)(hs) ≡
 .29             let i : H • i ∈ Tnis(t)(hs) ∧ i + 1 ∉ Tnis(t)(hs) in i end
 .30             pre has_journey(t)(hs),
 .31
 .32           startPa : Tnid → HSu ~→ Pa
 .33           startPa(t)(hs) ≡ hs(starti(t)(hs))(t) pre has_journey(t)(hs),
 .34
 .35           endPa : Tnid → HSu ~→ Pa
 .36           endPa(t)(hs) ≡ hs(starti(t)(hs))(t) pre has_journey(t)(hs),
 .37
 .38           Tnrt : Tnid → HSu ~→ Rt
 .39           Tnrt(t)(hs) ≡
 .40             ⟨ hs(i)(t) | i in ⟨ starti(t)(hs) .. endi(t)(hs) ⟩ ⟩
 .41             pre has_journey(t)(hs),
 .42
 .43           allTns : HS → Tnid-set
 .44           allTns(hs) ≡ { t | t : Tnid • Tnis(t)(hs) ≠ {} }
 .45       end
```

A.2.3 Traffic—Scheduling

Basic scheduling is now concerning with formation of history sequences in which trains are at certain places at certain times. Below this is expressed quite directly; a large number of funstions could be added, as could cost or other quality measures on history sequences.

Many details can be added. For example, the possible times for a train's positions should induce a total ordering on the positions, thereby defining a start and an end position, in between which the train should move only!

```
7.0  scheme
 .1    SCHEDULING =
 .2      extend TRAFFIC_PROPS with
 .3        class
 .4          type
 .5            /* Requirements for a set of trains */
 .6            RQs = Tnid ⇸ RQ,
 .7            /* Requirements for a train is sets of permissable sets of times at which the
 .8
 .9              train */
 .10           /* should be at certain places */
 .11           RQ = Pa ⇸ (H-set)-set
 .12
 .13         value
 .14           Sat : RQs × HSu → Bool
 .15           Sat(r, hs) ≡
 .16             (
 .17               ∀ t : Tnid, p : Pa •
 .18                 t ∈ dom r ∧ p ∈ dom r(t) ⇒
 .19                 (
 .20                   let
 .21                     hss = r(t)(p),
 .22                     hs′ = { h | h : H • h ∈ inds hs ∧ t ∈ dom (hs)(h) ∧ hs(h)(t) = p }
 .23                   in
 .24                     ∃ hs″ : H-set • hs″ ∈ hss ∧ hs′ ⊆ hs″
 .25                   end
 .26                 )
 .27             ),
 .28
 .29           SatE : RQs × HSu → Bool
 .30           SatE(r, hs) ≡ dom r = allTns(hs) ∧ Sat(r, hs),
 .31
 .32           Solve : RQs → HSu-set
 .33           Solve(r) ≡ { hs | hs : HSu • Sat(r, hs) },
 .34
 .35           SolveE : RQs → HSu-set
 .36           SolveE(r) ≡ { hs | hs : HSu • SatE(r, hs) }
 .37
 .38         value
 .39           badness : RQs × HSu → Nat,
```

```
.40
.41        schedule : RQs ~→ HSu
.42        schedule(r) as hs
.43          post
.44            hs ∈ SolveE(r) ∧
.45            ∼ (∃ hs′ : HSu • hs′ ∈ SolveE(r) ∧ badness(r, hs′) < badness(r, hs))
.46          pre SolveE(r) ≠ {}
.47      end
```

A.2.4 Rescheduling

Based on the above we must now consider *re-scheduling*, in response to certain events. This is still up in the air, but a possible way of attacking this could be to consider events as *RQs* values, to be interpreted under *Sat*, at a certain point in time. Access to original requirements is probably a good idea.

```
8.0  scheme
 .1    RE_SCHEDULING =
 .2      extend SCHEDULING with
 .3        class
 .4          type EV = {| (r, i) : RQs × H • later(r, i) |}
 .5
 .6          value
 .7            /* any time in RQs must be later than or at time i*/
 .8            later : RQs × H → Bool
 .9
 .10         value
 .11           π : HS × H ~→ HS
 .12           π(hs, i) as hs′ post len hs′ = i ∧ (∀ j : H • j ≤ i ⇒ hs(i) = hs′(i)) pre len hs ≥ i
 .13
 .14         value
 .15           reschedule : EV → (HSu × RQs) ~→ HSu
 .16           reschedule(e, i)(hs, r) as hs′
 .17             post let hsi′ = π(hs′, i) in π(hs, i) = hsi′ ∧ Sat(e, hs′) ∧ SatE(r, hs′) end
 .18       end
```