Pinnacles of Software Engineering[1)]

25 Years of Formal Methods

Dines Bjørner

Department of IT, Technical University of Denmark,

Bldg. 344, DK-2800 Lyngby, Denmark.

E–Mail: db@it.dtu.dk; Web: http://www.it.dtu.dk/~db

March 17, 2000

[1)]This title is in contrast to [Zave and Jackson 1997a]

Abstract

In this invited paper[1)] we review 25 years of propagating *formal specification* in *software engineering*. We will do so through outlining a paradigmatic approach to the practice of software engineering. For the sake of contrasting argument we shall claim that this approach stands in sharp contrast to classical engineering — and that there is little help to be obtained from classical engineering in securing the quality of the most important facets of software engineering!

We shall be outlining a *software engineering*[2)] practice in which *formal techniques* are applied in *capturing* the *aplication domain* void of any reference to *requirements* let alone *software*; and in then *capturing requirements: Domain requirements* (*projected, instantiated*, possibly *extended* and usually *initialised* from *domain descriptions*), *interface requirements* and *machine requirements*. The *software engineering* practice then goes on to *design* the *software*: First the *architecture*, then the *program structure*, etc.

Throughout *abstraction* and *modelling*, hand-in-hand, are used in *applicative* (*functional*), *imperative* and *process* oriented *descriptions*, from *loose specifications* towards *concrete, instantiated descriptions*, using *hierarchical* as well as *configurational modelling*, *denotational* as well as *computational modelling*, and in *structuring* even small scale descriptions using appropriate *modularisation* concepts: *Schemes, classes* and *objects*.

All the concepts spelled in *this font* are software engineering "program" description notions that have been honed over the years, starting in 1973 with **VDM** [Bekić *et al.* 1974; Bjørner and Jones 1978; Bjørner and Jones 1982] and continuing with **RAISE** [Group 1992; Group 1995].

The current status of our approach to software engineering, based on extensive, but not exclusive use of formal techniques, developed significantly during my years as UN Director of the UN University's International Institute for Software Technology (UNU/IIST) in Macau, 1992—1997. Many large scale software developments based on the domain/requirements/software design paradigm outlined here were systematically applied to the experimental development of software designs for the computing support of a number of diverse infrastructure components[3)].

Special boxes, scattered throughout the text, highlight 'pinnacle' contribution by named computer and computing scientists as well as by specific R&D projects.

[1)] This is a first version (New Year 1998/1999). During January 1999 it may be edited into a shorter version and with fewer literature references.

[2)] The text high-lighted in *'this font'* identifies important software development principles and techniques — the main contribution and subject of this quarter century overview.

[3)] Railways [Bjørner *et al.* 1997; George 1995], financial service institutions [Bjørner 1997; Bjørner *et al.* 1998], manufacturing [Bjørner 1996; Janowski 1996; Janowski and Acebedo 1996; Janowski and Atienza 1997], ministry of finance [George *et al.* 1995; Dung *et al.* 1996], airlines [Anh and Moore 1996], air traffic [Bjørner 1995], etc.

Keywords

methodology approach to SE, domain knowledge issues in SE, historical review of SE; domain engineering, (domain, interface and machine) requirements engineering, software architecture; abstraction, modelling, loose specifications, representational abstraction, operational abstraction, denotational semantics, operational semantics, hierarchical presentation, configurational presentation; VDM, RAISE, Duration Calculus.

1 INTRODUCTION

Engineering "walks the bridge" between science and technology. Engineers create technological artefacts based on scientific insight; and engineers study 'technological' artefacts with a view towards establishing their scientific content[4]. Mathematics is one of the standard tools of engineering.

The engineering paradigm outlined in this paper is the result of 25 years[5] of concerted effort and builds around a "triptych" of phases: *domain engineering*, *requirements engineering* and *software design*. Traditional software engineering focuses only on the second and third panel of this triptych. Traditional engineering, building on the natural sciences — *which software engineering does not* — relies on these sciences (physics (including mechanics, electricity, hydrodynamics, thermodynamics, nuclear physics, etc.), chemistry, etc.) to provide the domain knowledge. Not so in software engineering: As software is being sought for well-nigh any area of human activitity somebody has to build the scientific foundations for those activities. Domain engineering, a truly cross/inter/multi/ & trans-disciplinary effort, serves this rôle: To provide the possibility of theoretical foundations for, ie. answers to such questions as: *What are the functions and components of* railways, financial service infrastructure, manufacturing, public administration, airlines, air traffic, etc.[6] — questions which seek mathematical answers in order to claim proper understanding.

Whereas traditional engineering, after some steps of development, makes a mental and physical leap into construction (building, manufacturing), software engineering provides for a rather much more smooth sequence of document designs: *domain* descriptions, *requirements* descriptions, *software architecture* descriptions, etc., until an "executable description" is arrived at! So, whereas traditional engineering always have to relate to a physical world, outside the design engineers' office, to make sure that the designs are indeed models of that world, the software engineer primarily has to relate to the world of meta-mathematics, ie. to computability.

A software engineer who does not, acutely understand this is not one I would hire!

In the practical, everyday, "hum-hum" world of engineering activities there may be many facets that may lead us on to believe that much of what we, as software engineers, do, has some

[4] Currently UML and Java are studied in this manner.

[5] In 1999 it is 25 years since the IBM Science Laboratory in Vienna, Austria, issued [Bekić *et al.* 1974].

[6] Railways [Bjørner *et al.* 1997; George 1995], financial service institutions [Bjørner 1997; Bjørner *et al.* 1998], manufacturing [Bjørner 1996; Janowski 1996; Janowski and Acebedo 1996; Janowski and Atienza 1997], ministry of finance [George *et al.* 1995; Dung *et al.* 1996], airlines [Anh and Moore 1996], air traffic [Bjørner 1995], etc.

resemblance to other forms of engineering. But that may be misleading: the collaborative work: OK; the interaction with clients and the recording of their wishes: Mostly this is ordinary human and social interplay or outright petty bureaucratics. As we shall later see: The real results of such interactions are, in the ordinary engineering world rather simply quantifiable, a few numbers here and there — constrained, possibly, by subtle mathematical models. In the world of software engineering, resulting requirements amount to full-fledged computable objects — artefacts of a far more complex and deep nature than the characteristics of an electronic hearing aid, of a train wheel gauge, of the orbit of a satellite, etc.

So, let us be prepared for an altogether different world of the engineering of software from those of the worlds of the engineering of electronics, mechanics, chemical compounds or other.

We will return to the theme outlined above in the concluding section, Section 6.

This paper will go rather straight to its topic: That of outlining a `tried-&-tested` approach to large scale software development using mathematics. It will do so on the basis of reference to a long list of accomplished, including experimental (R&D) as well as commercial software developments.

The R&D projects were undertaken in order to propagate the use of formal techniques in software development and all had three objectives in mind: (i) to test the usefulness of and further R&D first the **VDM** approach later the follow-on and greatly expanded **RAISE** approach [Group 1992; Group 1995]; (ii) to actually develop useful compilers (**CHILL** formal definition and full **CHILL** compiler [Haff 1981; Haff and Olsen 1987], **Ada** formal definition and full **Ada** compiler [Bjørner and Oest 1980b; Bjørner and Oest 1980a; Clemmensen and Oest 1984; Oest 1986]), and, in later years, to the development of software support for large scale infrastructure components[7)]; and (iii) to study, revise and propose further formal techniques based development techniques and tools.

1.1 Formal Specification and Design Calculi

*A **method** is here taken to be a set of principles for analysing and selecting techniques and tools in order efficiently to construct efficient software.*

When we say *formal method* we basically mean *formal specification* and the use of *design calculi* on such specifications.

[7)] Railways [Bjørner *et al.* 1997; George 1995], financial service institutions [Bjørner 1997; Bjørner *et al.* 1998], manufacturing [Bjørner 1996; Janowski 1996; Janowski and Acebedo 1996; Janowski and Atienza 1997], ministry of finance [George *et al.* 1995; Dung *et al.* 1996], airlines [Anh and Moore 1996], air traffic [Bjørner 1995], etc.

Since *"all we, as software engineers, do"* is to create *descriptions*, from *abstract domain* descriptions via *requirements* descriptions to *software design* descriptions, specification and programming notations become our major *tool*. Formal descriptions are the prerogative of the software (the domain, the requirements and the software design) engineer. Casual clients and even procurers of software need not necessarily see the formal descriptions, but "sign off" on informal, well-structured *synopses*, *narratives* and *terminologies*. The software engineer links these informal descriptions to formal ones, and uses the development of formal descriptions to secure conciseness and accuracy of informal documents.

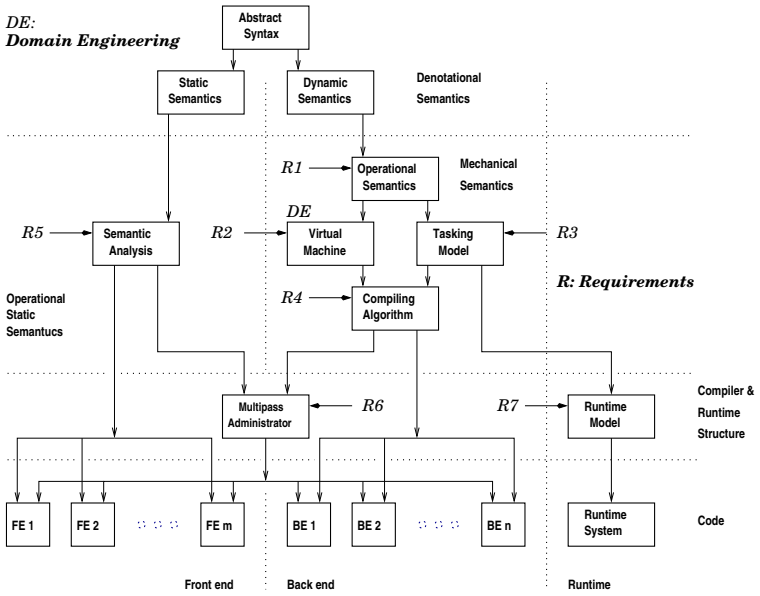
In this paper we shall bring many examples of informal, narrative and formal specifications. We shall not bring examples of calculations (verification and other) over such documents. This may seem strange: *after all, is formal specification not about 'proving' things ?* Well, yes, but. It is also about verification. We have found, however, over the last 25 years, that the *biggest return on investment, so clearly has been on formal specification.* Much (more) could be said about this. Suffice it here to observe that our own contribution has been in the area of principles and techniques for formal specification, including its use in systematic to rigorous development.

1.2 Relevant Current Other Publications

We refer to three recent publications that cover "neigbouring ground" to that of the present paper:

- [Bjørner and Cuéllar 1998] D. Bjørner and Jorge R. Cuéllar. *Software Engineering Education: Rôles of formal specifications and design calculi.* Annals of Software Engineering, Vol.6, 1998 (1999).
- [Bjørner 1997–1998] Dines Bjørner. *Domains as a Prerequisite for Requirements and Software — Domain Perspectives & Facets, Requirements Aspects and Software Views.* Proceedings of a US ONR sponsored International Workshop, RTSE'97: *Requirements Targeted Software and Systems Engineering*, Springer-Verlag Lecture Notes in Computer Science, vol. 1526, pp 1–41, 12–14 October, 1997, Bernried am Staarnberger See, Germany. Ed. Manfred Broy, 1998.
- [Bjørner 1999] Dines Bjørner. *Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re-assessment of Software Engineering ?* South African Journal of Computer Science, January 1999.

Figure 1: Compiler Software Development Graph



These papers provide detailed discussion of some issues that may be covered from a different angle in the present paper. The first two references above contain practically no examples. The last paper above carries through one example to basically illustrate the domain/requirements/software design paradigm otherwise assumed in the current paper. Given that paper we shall be brief in Sections 5 and 5.2.

1.3 A Review of 1980's Domain, Requirements and Software Design Work

The domain is what exists prior to any thought of requirements.

The domain for a compiler is the languages from and to which compiling is to take place.

1. **Example:** *Systematic Compiler Development:* — *Ends on page 7*

First at IBM's Vienna Laboratory, in 1973–1974, a domain development project — which led to **VDM** — and then from 1978, two major examples of *domain-to-requirements-to-software--design* projects took place, unfolding, respectively using formal techniques, albeit not formally, but just systematically.

These were the **PL/I**, **CHILL** and **Ada** projects [Bekić *et al.* 1974], [Haff 1981; Haff and Olsen 1987] respectively [Bjørner and Oest 1980b; Clemmensen and Oest 1984; Oest 1986].

Now 20–25 years later we can review this work. In the **CHILL** and **Ada** cases development proceeded using formal specifications systematically.

Figure 1 shows the *software development graph* of the *phases, stages* and *steps* of the development of a compiler for languages of the **CHILL** and **Ada** kind, ie. typed, imperative, modular and concurrent languages. The concept of *software development graphs* was outlined in [Bjørner and Oest 1980a] and explored in [Bjørner and Nielsen 1985; Bjørner 1986a; Bjørner 1986b; Bjørner 1987].

The graph intends to show that proper, systematic to rigorous development of a compiler for a language like CHILL or Ada can suitably proceed in three major phases: the *domain engineering* phase of developing the denotational semantics of the *source language* and of the *virtual machine* (ie. the *target language*) — the top three boxes and the box labelled *virtual machine*. As shown here the *requirements engineering* phase is shown as successive refinements of the denotational semantics into an operational (also sometimes called a mechanical) semantics — five of the subsequent (from top of the figure down) boxes. The steps — within this (single stage) phase — successively concretises **what** the compiler must do: The *static semantics analysis* of the so-called **front end** and the *code generation* of the **back end**. Finally we show two *stages* of the *software design* phase: In the first stage the multi-pass structure of the compiler is designed, and in the last stage the many passes of the compiler front and back ends are coded.

The *domain requirements* is simply that the translation from source to target code is correct. In this example we show that *machine requirements* refinement occurs in response to externally stated desiderata — indicated by $\mathcal{R}$'s, viz.: *The compiler shall translate indefinitely large source texts in a limited addressing space, and the generated code shall likewise execute in a limited addressing space,* are two major machine requirements. *Interface requirements* primarily deal with compile and run-time diagnostics. We otherwise refer to [Haff 1981; Haff and Olsen 1987; Bjørner and Oest 1980b; Bjørner and Oest 1980a; Clemmensen and Oest 1984; Oest 1986]

for details.

End of *Systematic Compiler Development* **Example** 1

Some SE Pinnacles: *CHILL and Ada Compiler Development:* The CHILL and Ada compilers developed at *Dansk Datamatik Center* as from 1979 still seems to be the only compiler development that have followed the systematic paradigm inherent in formal developments. *DDC Intl.* still develops and markets Ada compilers according to this paradigm — and seems to be the only major provider of Ada compilers today! It also seems very likely, barring information that is hard to obtain, that the *DDC* developments enjoyed the following properties: well within forecasted budget and time, outstanding quality (eg. wrt. "bugs" and maintainability (resilience)). It is still a wonder to this author why most, if not all other compiler development today follows, what we consider rather antiquated development approaches.

What is the lesson learned from these compiler developments ?

In tackling the development of software support for large scale infrastructure components[8)] the basic approach taken was to try capture as much of *the semantics of the professional terms of the stake-holder languages* uttered in the domain: Not the semantics of sentences, just the important nouns, verbs etc.

Some SE Pinnacles: *Algol–W, Euler, Pascal, Modula & Oberon: Niklaus Wirth:* Although functional programming has its merits, so does imperative programming [Hoare and et al. 1987]. Beautiful, industrial-strength imperative programming languages have been the prerogative, it seems of Niklaus Wirth. The immaturity of our industry is, amongst others, illustrated by their lack of support for the languages mentioned in the header of this box. Surely Wirth's work must rank amongst the most important in the last 30 years of software engineering [Wirth 1963; Wirth and Hoare 1966; Wirth and Weber 1966; Wirth 1971b; Wirth 1971a; Wirth 1973; Wirth 1976; Jensen and Wirth 1976; Wirth 1982; Wirth 1988a; Wirth 1988b; Wirth and Gutknecht 1989; Reiser 1991; Wirth and Gutknecht 1992].

1.4 Structure of Paper

We shall therefore link the phases, stages and steps of our development paradigm, as it is based on formal techniques, by a number of snapshot examples that together should show the breadth and depth, possibilities and open research issues of our approach.

The structure of the paper follows the structure of proper software development as well as

[8)] Railways [Bjørner *et al.* 1997; George 1995], financial service institutions [Bjørner 1997; Bjørner *et al.* 1998], manufacturing [Bjørner 1996; Janowski 1996; Janowski and Acebedo 1996; Janowski and Atienza 1997], ministry of finance [George *et al.* 1995; Dung *et al.* 1996], airlines [Anh and Moore 1996], air traffic [Bjørner 1995], etc.

of lecture notes for and a course in software engineering. In this paper we shall only cover the main parts of that development cum course (lecture notes) and neither the crucial platform issues (such as use of Java, CORBA, etc.), nor the ancillary issues of quality control, legal issues of software, project & product management, etc.

First we cover the main issue of abstraction and modelling, Section 2. The subsequent three sections, Section 3–5, therefore each cover their phase of development, and within these, the various stages that might be considered and the more minute steps through which we then actually conduct development. A reading of the contents listing should reveal which phases, stages and some of the steps we are referring to.

The emphasis, therefore, of the next sections is, on one hand to exemplify these phases, stages and steps, and, on the other hand to illustrate many of the abstraction and modelling techniques mentioned in the abstract.

We sincerely and strongly believe in the power of examples, and we have selected some rather "different" ones!

We also believe that many software engineering contributions could be greatly improved by placing them in a larger context. Here we offer "a largest" context and show, we claim, how formal techniques span the entire development process. It is quite a *"tour de force"*! The desire for comprehensiveness thus has the cost of this being a rather long paper.

An SE Pinnacle: *Lucas, Bekić [Bekić 1984], Jones et al.: The Vienna Development Method:* VDM grew out of an effort to develop, systematically and based on a domain engineered denotational (-like) semantics of **PL/I** [Bekić *et al.* 1974], a compiler for that mid-1960's programming language for a very high level computer architecture then contemplated by IBM. VDM later showed itself strong as a development concept — with an abstraction and modelling language then called **Meta-IV** and with a notion of data reification — to serve in many diverse, non-compiler related development. **VDM**'s specification language **VDM-SL** has now been standardised by **ISO** [Larsen *et al.* 1996]. Several monographs and text books have been published [Bjørner and Jones 1978; Bjørner and Jones 1982; Jones 1980; Jones 1986; Fitzgerald and Larsen 1997]. VDM became the focus of a number of rather successful industry & academia oriented practice & theory symposia [Bjørner *et al.* 1987a; Bloomfield *et al.* 1988; Bjørner *et al.* 1990; Prehn and Toetenel 1991; Larsen 1993; Naftalin *et al.* 1994; Gaudel and Woodcock 1996; Fitzgerald *et al.* 1997]. From early on (1990) these symposia went well beyond VDM as the formal method, and the VDM Symposia were renamed into the FME, Formal Methods Europe, Symposia.

2 ABSTRACTION AND MODELLING

[Dahl *et al.* 1972] probably is the monograph that has had most influence wrt. abstraction in programming. Tony Hoare's essay [Hoare 1972] epitomises a (model-oriented) abstraction concept. The first mathematics cum abstraction oriented notation, **VDM**, was — independently — made available shortly after the publication of [Dahl *et al.* 1972]. Perhaps the set oriented programming language **SETL** [Schwartz 1973] offered a suitable alternative. **SETL,SETL1**, however, prioritised executability over abstract expressiveness, where the **VDM** notation (first **Meta-IV**, now **VDM–SL**) focused more exclusively on abstract, albeit model-oriented expressivness.

But a notation is not enough: Over the last 25 years a number of abstraction and modellling principles and technques — only one of which will be illustrated as from Section 3 onwards — have emerged. Among the principles and techniques we mention: *applicative* vs. *imperative modelling*, ie. modelling without, respectively with assignable state variables; *hierarchical* ("top-down", decompositional) vs. *configurational* ("bottom-up", compositional) abstraction, *denotational* vs. *computational modelling*, etc.

2. **Example:** *Abstract Syntax for Railway Nets:* — *Ends on page 13*

This example illustrates several abstraction and modelling principles and techniques.

We divide the presentation into two parts: a hierarchical first part followed by a configurational second part.

Hierarchical Presentation :

We focus on the railway net perspective of railway systems.

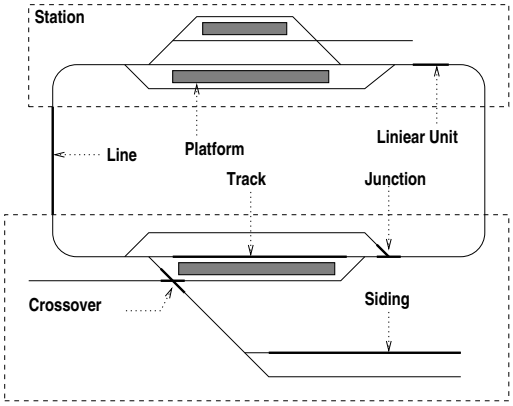
Pls. inspect figure 2 on the following page.

Our natural, professional railway language description proceeds as follows:[9)]

1. A railway net consists of lines and two or more stations.
2. A railway net consists of units.
3. A line is a linear sequence of one or more linear units.
4. The units of a line must be units of a net.

[9)] We enumerate the sentences for reference.

Figure 2: A "Model" Railway Net!



5. A station is a set of units.
6. The units of a station must be units of a net.
7. No two distinct lines and/or stations share units.
8. A station consists of one or more tracks.
9. A track is a linear sequence of one or more linear units.
10. No two distinct tracks share units.
11. The units of a track must be units of the station (of that track, and hence the net)
12. A unit is either a linear unit, or a switch point, or a simple crossover, or a switch-able crossover, etc.[10)]
13. A unit has one or more connectors.
14. For every connector there are at most two units which have that connector in common.
15. Every line of a net is connected to exactly two, distinct stations.

A corresponding, representationally abstract formal specification is:

type

Net, Line, Station, Track, Uni, Connector

[10)] A linear unit has two distinct connectors, a switch point has three distinct connectors, crossovers have four distinct connectors.

value

1. obs_Lines: Net → Line-**set**
1. obs_Stations: Net → Station-**set**
2. obs_Unis: Net → Uni-**set**
3. obs_Unis: Line → Uni-**set**
5. obs_Unis: Station → Uni-**set**
8. obs_Tracks: Station → Track-**set**
12. is_Linear: Uni → **Bool**
12. is_Switch: Uni → **Bool**
12. is_Simple_Crossover: Uni → **Bool**
12. is_Switchable_Crossover: Uni → **Bool**
13. obs_Connectors: Uni → Connector-**set**

axiom

forall n:Net, l,l′:Line, s,s′:Station, t,t′:Track, u:Uni, c:Connector •

1. **card** obs_Stations(n) ≥ 2,
3. l ∈ obs_Lines(n) ⇒ ∀ u:Uni • u ∈ obs_Unis(l) ⇒ is_Linear(u) ∧
7. l′ ∈ obs_Lines(n) ∧ l≠l′ ⇒ obs_Unis(l) ∩ obs_Unis(l′) = {},
7. l ∈ obs_Lines(n) ∧ s ∈ obs_Stations(n) ⇒ obs_Unis(l) ∩ obs_Unis(s) = {},
7. s′ ∈ obs_Stations(n) ∧ s≠s′ ⇒ obs_Unis(s) ∩ obs_Unis(s′) = {},
8. **card** obs_Tracks(s) ≥ 1,
9. s ∈ obs_Stations(n) ∧ t ∈ obs_Tracks(s)
   ⇒ ∀ u:Uni • u ∈ obs_Unis(t) ⇒ is_Linear(u),
10. t′ ∈ obs_Tracks(s′) ∧ t≠t′ ⇒ obs_Unis(t) ∩ obs_Unis(t′) = {},
14. **card** { u | u:Uni • c ∈ obs_Connectors(u) } ≤ 2,
15. ∀ s:Station,∃ s′:Station,l:Line • s≠s′ ⇒
    let sus = obs_Unis(s), sus′ = obs_Unis(s′), lus = obs_Unis(l) **in**
    let u:U • u ∈ sus, u′:U • u′ ∈ sus′, u″,u‴:U • u″ ∈ lus **in**
    let scs = obs_Connector(u), scs′ = obs_Connector(u′),
    lcs = obs_Connector(u″), lcs′ = obs_Connector(u‴) **in**
    ∃! c,c′ • c ≠ c′ ∧ scs ∩ lcs = {c} ∧ scs′ ∩ lcs′ = {c′}

end end end

Configurational Presentation :

16. A path, p:P, is a pair of connectors, (c,c′), of some unit. A path of a unit designate that a train may move across the unit in the direction from c to c′. We say that the unit is open in the direction of the path.
17. A state, $\sigma : \Sigma$, of a unit is the set of all open paths of that unit (at the time observed). The state may be empty: the unit is closed.
18. A unit may, over its operational life, attain any of a (possibly small) number of different states ω, Ω.
19. A route is a sequence of pairs of units and paths —
20. such that the path of a unit/path pair is a possible path of some state of the unit, and such that "neighbouring" connectors are identical.
21. An open route is a route such that all its paths are open.
22. A train is modelled as an open route.
23. Train movement is modelled as a discrete function (map) from time to open routes such that for any two adjacent times the two corresponding open routes differ by at most one of the following: a unit path pair has been deleted from (one or another end) of the open routes, or (similarly) added, or both, or no changes — a total of seven possibilities.

type

16 P = C × C

17 Σ = P**-set**

18 Ω = Σ**-set**

19 R′ = (Uni × P)*

20 R ={| r:R′ • wf_R(r) |}

22 Trn = {| r:R • open_R(r) |}

23 Mov = T $\overrightarrow{m}$ Trn

value

17 obs_Σ: Uni → Σ

18 obs_Ω: Uni → Ω

20 wf_R: R′ → **Bool**

wf_R(r) ≡

$\forall$ i:**Nat** • i $\in$ **inds** r **let** (u,(c,c′)) = r(i) **in** (c,c′) $\in \bigcup$ obs_Ω(u) $\wedge$
  i+1 $\in$ **inds** r $\Rightarrow$ **let** (_,(c″,_)) = r(i+1) **in** c′ = c″ **end end**

21 open_R: R $\rightarrow$ **Bool**

open_R(r) $\equiv$ $\forall$ (u,p):U×P • (u,p) $\in$ **elems** r $\wedge$ p $\in$ obs_Σ(u)

23 wf_Mov: Mov $\rightarrow$ **Bool**

wf_Mov(m) $\equiv$
  card dom m $\geq$ 2 $\wedge$ $\forall$ t,t′:T • t,t′ $\in$ **dom** m $\wedge$ t < t′
  $\wedge$ $\sim\exists$ t″:T • t″ $\in$ **dom** m $\wedge$ t < t″ < t′ $\Rightarrow$
    let (r,r′) = (m(t),m(t′)) **in** clauses (i) − (vii) **end**

End of *Abstract Syntax for Railway Nets* **Example** 2

The description is formalised in **RSL**.

An SE Pinnacle: *RAISE: Rigorous Approach to Industrial Software Engineering:* The above formalisation is expressed in **RSL**, the **RAISE S**pecification **L**anguage. **RAISE** [Group 1992; Group 1995] builds on **VDM** [Bekić *et al.* 1974; Bjørner and Jones 1978; Bjørner and Jones 1982], **OBJ** [Goguen *et al.* 1975; Goguen *et al.* 1977; Burstall and Goguen 1977; Goguen *et al.* 1978; Burstall and Goguen 1980; Futatsugi *et al.* 1985], **Standard ML** [Milner *et al.* 1990] and **CSP** [Hoare 1978; Hoare 1985; Roscoe 1997]. **RAISE, RSL** and the **RAISE Toolset** was researched, developed and industry-tested during the late 1980's to early 1990's by collaborative teamwork notably between the Danish *Dansk Datamatik Center, CRI Inc.* and the British *STL* (later *BNR*).

The property as well as model-oriented rail net model just given illustrates just some **type** concepts of a railway system.

Some SE Pinnacles: *Type Theories: Per Martin-Löf, J.-Y. Girard and Luca Cardelli:* Type theory is, perhaps, the most important contribution that computer science has made to mathematics. From the ordinary programming language type concepts (of for example the languages mentioned in the **Algol-W** to **Oberon** "pinnacle" box [page 7]) to the full-fledged, computationally complete intuitionistic type theory of Per Martin-Löf [Nordström *et al.* 1990] via Jean-Yves Girard's likewise proof-theoretic treatment [Girard *et al.* 1989] to Cardelli's more mundane, but no less exciting connection of type theory to modularity [Cardelli 1987; Cardelli and Wegner 19; Abadi and Cardelli 1996] type theory has shown to be a core concept that must be well understood by the practising software engineer.

VDM, and its follow-on **RAISE**, would not have been thinkable without the work by John McCarthy, Peter Landin and others:

Some SE Pinnacles: *John McCarthy: Abstract Syntax and Computability:* John McCarthy gave decisive impetus also to practical software engineering through the contributions of **LISP** [McCarthy 1960; McCarthy 1962; McCarthy and et al. 1962], *Abstract Syntax* [McCarthy 1963] and correctness of compiler development [McCarthy and Painter 1966]. It is the current authors strong conviction that any software engineering professional must be reasonably aware of McCarthy's mathematical theory of computation including the concept of 'abstract syntax' and the operational semantics of LISP.

A main characteristum of **VDM** is that it offers an applicative ("functional programming") style of description. Functional programming, as we for example know it today through the notations of **Standard ML** [Milner *et al.* 1990], would not have been thinkable without the work of Peter Landin.

An SE Pinnacle: *Peter Landin and Applicative Programming:* Peter Landin's work, at Queen Mary College in London, is seminal in bringing classical meta-matematics to bear on modern computer science. Every software engineer ought be well-versed in λ-Calculus — such as intriduced and studied by Landin [Landin 1964; Landin 1965; Landin 1966b; Landin 1966a]. Landin is also to be credited with elegant, albeit mechanical semantics models of core **ALGOL**-like (hence **Pascal, Modula, C** and **Java**-like) programming languages.

From the 1980's onward additional model-oriented specification notations emerged, notably **Z** [Abrial 1980; Hayes 1987; Woodcock and Loomes 1988]. Variants of **VDM** likewise enmerged: Viz., the so-called 'Irish' School of VDM [an Airchinnigh 1991], and the divergence between the 'British' School as propagated by Cliff Jones [Jones 1980; Jones 1986; Jones 1990] and the European continental 'schools': "Dutch" [Middelburg 1988; Middelburg and de Lavalette 1991] and "Danish" and "German [Schmidt and Völler 1985; Haß 1987; U.Schmidt and R.Völler 1987]".

An SE Pinnacle: *Jean-Raymond Abrial: Z and B:* During his stay at Oxford — discussing, no doubt, with Cliff Jones, one of the "fathers" of **VDM**, Abrial "finalised" thoughts on **Z** [Abrial 1980], an elegant set-oriented description language. Citations in the paragraph prior to this frame attest to the growth of research into usage of **Z**: Oxford, more than Lyngby and Manchester, proved to be the prime breeding ground for formal methods, using **Z** rather than **VDM** — and, although there are great similarities between the seven year older **VDM** and **Z**, the two schools continued in parallel. Abrial went on, later, to create **B** — again an utterly simple, yet "powerfully" elegant and somewhat more concrete programming *cum* specification notation **AMN** (*Abstract Machine Notation*) [Abrial 1996]. It appears that **B** is now mandated, in France, when development of new software for certain safety critical applications is procured.

3 DOMAIN ENGINEERING

Before we can design the software we must know its requirements. Before we can express the requirements we must understand relevant parts of the domain.

A domain specification describes the application area (plus usually quite a bit more) **as it is** — indicatively [Jackson 1995] — void of any reference to requirements, let alone the desired software.

3.1 Intrinsics Modelling

The domain intrinsics are those properties of the domain that are ("more-or-less") invariant wrt. always changing (i) support technologies, changing (ii) rules & regulations, changing (iii) user behaviour etc. What (i–ii–iii) refers to should be clear in subsequent sections.

The hierarchical part of the rail net example, Example 2 on page 9 of Section 2, illustrates an intrinsics model.

3.1.1 Looseness / Abstraction

3. **Example:** *Airline Time-table:* — *Ends on page 16*

In the domain of airline time-tables (tt:TT) everything is possible. It is all too easy to over--specify, so we choose to under-specify. Queries inspect, but do not change time-tables. Queries result in values (V). Updates alter time-tables (modelled applicatively, as the generation of new time-tables). Updates, besides, result in responses (R).

type V, R

TT

Q = TT → V

U = TT → TT × R

Φ = Q | U

value

q, q′, ..., q″:Q, u, u′, u″:U

query: Q, query(tt) ≡ **let** c = q ∏ q′ ∏ ... ∏ q″ **in** c(tt) **end**

updat: U, updat(tt) ≡ **let** c = u ∏ u′ ∏ ... ∏ u″ **in** c(tt) **end**

The above expresses that there is an indefinite number of ways, $q, q', ..., q''$, of querying, respectively updating, $u, u', ..., u''$, a time-table, and that the general query (respectively update) function non-deterministically selects and perform one.

That is as much as we can say! That may surprise you. But consider that a person posssessing a (copy of a) time-table can do anything to it: count the number of pages, number of characters listed, average first daily departure time from any airport anywhere, meantime between arrivals, etc. There is no end to domain stake-holders' ingenuity.

End of *Airline Time-table* **Example** 3

Some SE Pinnacles: *Edsger Dijkstra: Non-determinism:* Dijkstra has most forcefully advocated the concept of expressing programs using non-deterministic concepts [Dijkstra 1975; Dijkstra 1976]. His techniques also of weakest pre- and strongest post-condition in specification of algorithms are seminal. Together they form indispensable principles and techniques for the practising software engineer.

Non-determinism, and the technique of leaving types of data further unspecified, has proven to allow a very efficient software engineering development principle: Namely that of postponement of design decisions. We find that epitomised in the concept of algebraic semantics.

Some SE Pinnacles: *Lucas, Goguen, Futatsugi et al.: Algebraic Semantics:* It seems that the first publication proposing algebraic semantics was [Lucas 1972]. The most cited first paper was [Liskov and Zilles 1974]. Systematic research into the algebraic approach was spearheaded by Joseph Goguen [Goguen *et al.* 1975; Goguen *et al.* 1977; Burstall and Goguen 1977; Goguen *et al.* 1978; Burstall and Goguen 1980; Futatsugi *et al.* 1985]. Current algebraic semantics development centers around **CafeOBJ** [Nakagawa *et al.* ????; Futatsugi and Diaconescu ????; Nakagawa 1997] and **CASL** [on Semantics 1997; on Language Design 1997; Mosses 1997]. Standard monographs on algebraic semantics are [Guessarian 1981; Ehrig and Mahr 1985; Bergstra *et al.* 1989; Ehrig and Mahr 1990; Horebeek and Lewi 1989; et al. (Eds.) 1991]. **VDM** offers a model-oriented approach to software specification. **OBJ, CafeOBJ** and **CASL** — in contrast — offers an algebraic approach: One in which types are abstracted into sorts, ie. not defined (albeit abstractly) as in VDM, and which emphasises looseness. Proper software engineering needs both approaches. Any professional software engineer need be well-versed in the spectrum as for example offered by **RAISE** [Group 1992; Group 1995].

3.1.2 Concreteness / Processes

4. **Example:** *Fisheries Auction:* — *Ends on page 22*

This examples goes to the other extreme of the abstraction to concretisation spectrum as compared to the previous example. It rather concretely models the "goings-on" at a fisheries auction. We claim that since the behaviour we wish to capture is a rather operational one we must model that behaviour likewise operationally. That is: Not much room for abstraction here!

Domain Synopsis The domain is that of a fisheries industry. We narrow our example down to the activities surroundingthe auctioning of fish.

Domain Narrative: Flow of Material, Sorting and Auctioning. To the auctioning activities we include first the tangible, physical events and actions:

- the return, after a fishing expedition, of fishing boats to a fishing harbour, ie. a harbour with an auction house;
- the unloading of boxed and sometimes also iced fish from the boat by lift-trucks which shuttles between the boat at quay side and an in-buffer area of the auction house, an area where the boxes are temporarily stored;
- the move of the in-buffered boxes to a sorting area;
- the sorting of fish into categories, ie. the transfer of individual fish from one set of boxes to another set of boxes;
- the in-buffering of fish boxes and sorting is grouped by fisher;
- the move of the sorted fish (in their boxes) to an auction area;
- the auctioning of the fish by batches of boxes of fish of same quality; and
- the move of the sold fish boxes to an out-buffer area.

Domain Stake-holders This example will illustrate the following seven stake-holder perspectives:

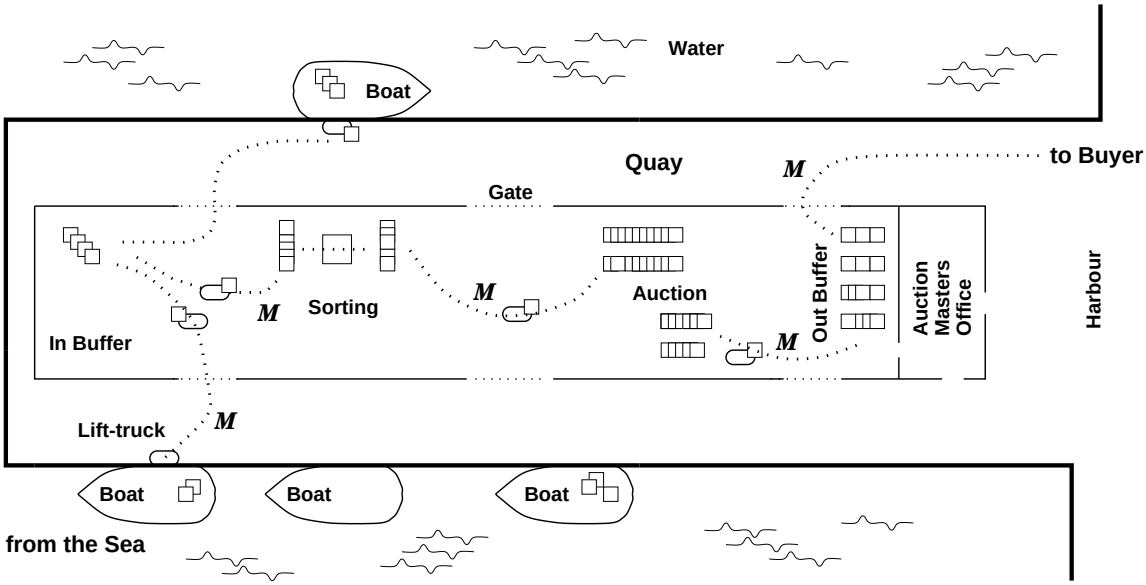
- Fishers Owners

  We omit treatment of staff on the fishing boat.

- Lift-truck Drivers Workers

  The lift-truck drivers usually work for the fishers co-operative of sorters.

Figure 3: Fish Harbour Auction House — $\mathcal{M}$: Movement



- Sorters Workers

  The sorters work for the fishers co-operative of sorters.

- Auctioneer Owner

  The auctioneer owns or has a commission from the state or local authorities to run the auction house.

- Auctioneer's Assistants Workers

- Buyers Owners of Consultants

  Buyers are either, or private consultants who act on behalf of fish whole-salers or fish processing plants.

- Fisheries Inspectors Regulatory Agency

Domain Narrative: Flow of Information In addition to the obvious physical, "hard work" actions of unloading, truck driving, sorting and auctioning, there are a number of less visible actions going on in parallel:

- Optional notification, by telephoning the auction masters office or the co-operative of sorters, from the sea, en route to the harbour, of estimated catch to be landed.
- Usual placement of a telephone message of "tomorrow's" estimated auction quantities by fish type. Prospective buyers (incl. whole-sale dealers and fish processing plants) can then make decisions as to which auctions to attend.
- Sorting also results in a list, by fisher, of weighed quantities and judged qualities; this list goes to the fisher and to the auction masters office.
- Auctioning results either in certain boxes not being sold, either because they are withdrawn by the fisher, or by the sorters on behalf of the fisher, or because they are judged sub-standard, or because there is no bid; or in a price. For each sold or not-sold batch of boxes of fish (still of the same quality, and still grouped by fisher) a fisher, price, quantity and quality entry is made on an auction list; list goes to the auction masters office and to the fishers' sorting co-operative.
- The auction masters office pays cash the amount received for fish sold, to those fishers who report to the office the day of the auction — or otherwise effects a bank-to-bank transfer.
- The auction masters office "same day faxes" detailed, per bid, invoices to all buyers who bought fish that day. Buyers are given lee-way to pay within typically 30 days! Auction master thus must pay fishers immediately but receive buyer payments only (much) later!
- The auction master electronically transfers a summary of fish sold, by fisher, type, quality and quantity, to the ministry of fisheries the same morning of the auction.
- The fisheries inspectors monitor all sorted fish before auctioning and may disallow certain sales, and may, in general, up- or down-grade sorters' quality decisions. They also report to the ministry of fisheries their fisher, fish type, quantity and quality findings.

Domain Analysis The domain is concrete.

It is characterised by agents: Fishers who are here seen as synonymos with their boats, lift-truck drivers, who are here seen as synonymos with their lift-trucks, sorters, the auctioneer, buyers with their states of what they have (so far, during an auction) bought, and fisheries inspectors with their state of recorded catch qualities.

Since lift-truck drivers and sorters all belong to either of a number of cooperatives we model lift-truck drivers and sorters of one cooperative as a cooperative agent.

Some agents have states fishers' boat state, lift-truck load state, auctioneer's offering/-close/next lot transition states, buyers' acquisition state, etc.

Agents are modelled as processes (some with process states).

The domain is also characterised by state components.

There is the state of a boat in terms of its unique fishing boat identification and as a set of fish boxes, these can, in the domain, be characterised by their box identification and content: fish type, quantity, quality, date (of fishing), place (of fishing), etc. We can speak of all these things — the fisher does. It may not be recorded, written down, marked on the box or otherwise — although, of course, the fishing boats' log book implies much. So we put this information into the domain model although we may not be able, technologically or it may not be resource-wise feasible, to actually project this information onto a requirements.[11)]

There is the state of the in-buffer temporary storage area, characterised in terms of its sets of boxes, grouped by fisher.

There is the state of the sorting area characterised by pairs of sets of boxes, again grouped by fisher: unsorted, respectively sorted.

There is the state of the auction area characterised by pairs of sets of boxes, again grouped by fisher: so far unsold, respectively so far sold.

There is the state of the out-buffer temporary storage area, characterised in terms of its sets of boxes, grouped by buyer.

And there is the state of the auction masters office as consisting of recorded report lists on sorted fish (to help generate) report lists on catch statistics (to be sent to the fisheries ministry), and report lists on sold fish, and, based on these latter, as also "building up" buyer invoice sets, and fisher payment slips.

We model all these states as processes.

Domain Formalisation

type FIdx, CIdx, IIdx, BIdx, SAIdx, TIdx

Boats = FIdx $\overrightarrow{m}$ Boat, FiCop = FIdx $\overrightarrow{m}$ CIdx

Coops = CIdx $\overrightarrow{m}$ Coop, Buyrs = BIdx $\overrightarrow{m}$ Buyer

...

value

[11)] Note that we list any fisher both as an agent and the fishers boat as a component. Recall our decision to aggregate the two, for each pair fisher and fishing boat, into one process!

```
  boats:Boats, coops:Coops, buyrs:Buyrs, ficop:FiCop
  ...
  system: Unit → Unit
  system() ≡
        /∗ agents ∗/
    ‖ { fisher(fi)(ficop(fi))(boats(fi)) | fi:FIdx }  ‖
    ‖ { cooperative(c)(coops(c)) | c:CIdx }       ‖
    ‖ { inspector(i) | i:IIdx }
    ‖ auction_master()                            ‖
    ‖ { buyer(b)(buyrs(b)) | b:BIdx }
        /∗ components ∗/
    ‖ in_buffer()                                 ‖
    ‖ { sorting(sa) | sa:SAIdx }                  ‖
    ‖ auction()
    ‖ out_buffer()
    ‖ ministry()

type
  Boat = Box-set
  Box = BoxId×FType×FQuant×FQual×FDate×FPlace×...
channel
  ftco[ ]:{ready}, cotf[ fi ]:({ready}×TIdx)
  ftlt[ li ]:Box-set, lttf[ li ]:
value
  fisher: FIdx → CIdx → Boat → Unit
  fisher(fi)(c)(boat) ≡
    if boat ≠{}
      then
        let bs:Boat • bs ⊆ boat in
        ftco[ c ]!(ready,fi);
        let (ready,t) = cotf[ c ]? in bs!ftlt[ t ]  end
        fisher(fi)(c)(boat \ bs) end
      else skip end
```

Only non-empty fishing boats need have lift-trucks collect boxes (*bs*). The fisher informs his cooperative that fish boxes are ready for collection. The fishers' cooperative, c, schedules which truck, t, is dispatched to collect *bs*. The number of boxes collected is, in the domain, set arbitrarily (ie. non-deterministically).

auction_master() ≡ auctioneer() ‖ office()

coop: CIdx → **Unit**
cooperative(i) ≡ (‖ { lift_truck(t) | t:TIdx }) ‖ (‖ { sort_group(s) | s:SIdx })

We leave the declaration (naming and typing) of channels and the detailing of individual processes as an exercise to the reader.

End of *Fisheries Auction* **Example** 4

The domain model of a fisheries auction (harbour) attempts to capture phenomena that go well beyond those for which we may wish computing support. That is: Domain modelling starts by casting the net wide, venturing well beyond what may be of immediate concern in a subsequent requirements capture. The reason for doing so, systematically, is simple: "one never knows"! Other requirements, hitherto unforeseen, may require domain analysis of other phenomena, etc. The wider a domain model covers an application domain, the more chance there is for any subsequent software to be more readily adaptable to such other, future software.

This point will be elaborated in Section 4.1.1.

Some SE Pinnacles: *Hoare, Milner and Petri: CSP, ccs and Petri Nets:* CSP: Communicating Sequential Processes as a concept presents the essentials of parallel processing and does so in an elegant setting. The CSP notation is frugal and its semantics obeys beautiful laws. The CSP concept is due to Tony Hoare [Hoare 1978; Hoare 1985; Roscoe 1997]. **ccs** — a calculus of communication systems — presents another elegant, and perhaps more foundational approach to concurrency: synchronisation and communication between processes. The **ccs** concept is due to Robin Milner [Milner 1980; Milner 1989]. In our view**CSP** and **ccs** — together with Carl Adam Petri's **Petri Nets** [Reisig 1998; Jensen 1985] — offer indispensable techniques for every professional software engineer's practice.

3.2 Stake-holder Modelling

Example 4 on page 16 hinted at stake-holders and their modelling, but the example was mostly brought to illustrate "operationalism": to contrast with Example 3 on page 15's "loose"

abstraction.

5. **Example:** *Resource Management, three Perspectives::* — *Ends on page 27*

This example shows the ability of formal specification to capture the essential differences between an enterprise's three levels of resource management. The strategic management of resources: Their "down-sizing" (sell-off, divestment) or "upgrading" (acquisition, investment); the tactical management of (physically relocatable [mobile]) resources: Their spatial allocation and scheduling; and the operations management of resources: Their allocation to tasks and the scheduling of these tasks. At the same time we are able to "put our finger on what distinguishes 'algorithmic' software from 'decision support software'.

Resources: Common notions are: resources, *r:R*, locations *l:L* and time intervals *ti:T*×*T*.

Scheduled availability of resources (SAR) (in certain time intervals), and scheduled spatial allocation (SSA) can be modelled as:

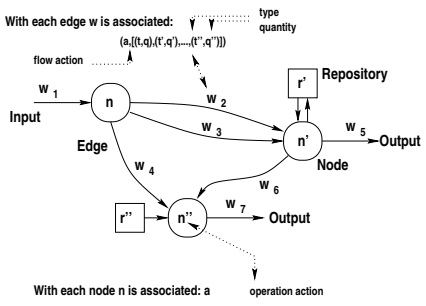
type R, L, T
  SAR = T×T $\xrightarrow{m}$ R-**set**
  SSA = T×T $\xrightarrow{m}$ (R $\xrightarrow{m}$ L)

Production [or Servicing] Plans and Tasks: Production of goods or services follow certain acyclic, graph-like production plans where nodes (N) designate actions (A, like machining or service rendering) and where arcs (W) designate quantified (**Nat**) flow (also a kind of action, A) of typed (RTyp) resources. Each kind (P) of product (or service) plan has its own production plan (PP). External input (inputs) initiate production, external output (outputs) deliver results. At each node there may be many next (NXT) nodes and a local repository of material resources (REP).

Figure 4 shows a production plan.

type P, Pn
axiom P ⊂ R
type N, W
  PPs = (Pn $\xrightarrow{m}$ PP)
  PP = inputs:NXT
    × graph:(N $\xrightarrow{m}$ (NXT × REP))
    × outputs:NXT
    × actions:((N|W) $\xrightarrow{m}$ A)

Figure 4: Production Plan 1



NXT = N $\overrightarrow{m}$ (W $\overrightarrow{m}$ (RTyp $\overrightarrow{m}$ **Nat**))

REP = RTyp $\overrightarrow{m}$ **Nat**

Production (or servicing) resources are needed in order to carry out actions (A). These are (to be) task scheduled and allocated to nodes and edges (NWAS) for [each] product plan (ASPD[s]). The dynamics of a task is now modelled as a pair: the (static) task scheduled and allocated plan and a (the dynamic) trace. For each instance of time the trace records at which nodes and on which edges the locus of control (of executon of the task) resides. A task thus records the past history and planned progress of the production, or servicing of one order.

type

NWAS = (N|W) $\overrightarrow{m}$ ((T×T) $\overrightarrow{m}$ (R $\overrightarrow{m}$ L))

ASPD = PP × NWAS

ASPDs = Pn $\overrightarrow{m}$ ASPD

Task = ASPD × Trace

Trace = T $\xrightarrow{\sim}$ (N|W)**-set**

Tasks = T $\xrightarrow{\sim}$ (Pn $\overrightarrow{m}$ Task**-set**)

The node and edge actions avail themselves of the node, respectively edge scheduled and allocated task resources.

At any moment an enterprise is honouring, for each named product, zero, one or more productions (servicings), ie. tasks.

Orders: Production (etc.) occur in response to, or expectation of customer (C, placed, placeable) orders. An order (O) names a product (resp. service, Pn), the time interval (ti) in which the order is to be delivered and a quantity (q). Each customer order receives a unique order name (On).

type C, On

O = Pn $\overrightarrow{m}$ (ti:(T×T) $\overrightarrow{m}$ q:**Nat**)

COs = C $\overrightarrow{m}$ (On $\overrightarrow{m}$ O)

Statistics and Estimates: A business objective (BO) records for suitable recources the number required. The number may be negative! Statistics (S) and estimates (E) records for suitable time intervals and locations the business objectives for that time interval and at that location.

type

BO = R $\overrightarrow{m}$ **Int**

S,E = (T×T) $\overrightarrow{m}$ (L $\overrightarrow{m}$ BO)

Statistics records past experience. Estimates record future expectations.

Strategic Management: We recall:

type

SAR = (T×T) $\overrightarrow{m}$ R**-set**

SSA = (T×T) $\overrightarrow{m}$ (R $\overrightarrow{m}$ L)

which expresses either a desired or an actual state: in some (planned or actual) interval some (planned or actual) resources are available (or occupied, or to be scheduled) (SAR), respectively at some (planned or actual) locations (SSA).

- **Planning:**

  Strategic planning is based on available resources, statistics and estimates, and yields, for (future) time intervals plans for possibly downsized or upgraded resources. There may be an infinity of "solutions" to strategic planning.

  value sp: R**-set** × S × E $\xrightarrow{\sim}$ (T×T) $\xrightarrow{\sim}$ SAR**-infset**

  We have assumed that this form of strategic planning toccurs in "isolation" from the market.

- **Negotiation:**

  Out of such a possible infinity of choices (or zero, for that matter!) one has to be chosen, if possible. Now negotation takes place in the context of the market. Each "player" (B) has own, strategic resource plans.

  type B

  value ng: (B × SAR**-infset**) × (B $\overrightarrow{m}$ SAR**-infset**) $\xrightarrow{\sim}$ SAR

- **Reconciliation:**

  Negotiation results, seen from each "player" in a possibly new resource schedule which may have to be reconciled with those (sp) originally planned, resulting a yet a possibly other, new resource schedule.

value rec: E × sp:SAR-**infset** × new:SAR $\xrightarrow{\sim}$ result:SAR

- **Revision:**

  value res: E × SAR $\xrightarrow{\sim}$ E

  Finally strategic planning may revise original estimates.

  - **Interpretation:** The "functions" sp, neg, rec and res are just given their signature. A number of **pre/post** conditions can be expressed — typically after some thorough knowledge-engineering acquisition — but not enough for us to be able to actually compute definite, let alone all answers. Together with the signatures and the **pre/post** conditions one can, however, formulate *requirements* to a (so-called "expert-system"-like) **decision software support system**.

Tactical Management: Tactical resource management now allocates scheduled resources in the context of estimates and actual (or expected, current) orders:

value tp: E × COs → SAR $\xrightarrow{\sim}$ SSA

- **Interpretation:** Remarks similarly to those above for the strategic "functions" can be attached to this tactical "function", and again we should be in a position to *knowledge-engineer*, ie. *requirements capture* a *decision support sosftware system*.

Operations Management: Operational management now takes current (and immediately expected future) orders and spatially allocated and scheduled resources and allocates them to production (or servicing) tasks:

value op: COs × SSA $\xrightarrow{\sim}$ ASPDs

- **Interpretation:** The "closer", however, we get to operations, the more our planning functions become algorithmically well-defined. Although one may not arrive at optimal algorithms for the implementation of *operations management* (due to NP completeness), this function is typically realisable using modern operations research analysis techniques.

Operations: The production plans can now be executed:

value ops: APSDs $\xrightarrow{\sim}$ Tasks

- **Interpretation:** The argument to this operations despatch, monitoring and control function, *ops*, and desired **pre/post** conditions, are such that this function is (almost trivially) algorithmically computer supportable.

Discussion: This example shows: (i) that it is possible to make reasonably non-trivial statements about strategic and tactical resource management and (ii) to relate these to operational management, (iii) that we can model at least three stake-holder perspectives within a "smooth" spectrum of models for: (1) strategic (owner and top-level executive) management, (2) tactical, divisional level management, and (3) operations management, and (iv) that we can identify a spectrum of software from *knowledge based decision support software* to *operations research analysis based* task scheduling and production monitoring & control *software*.

End of *Resource Management, three Perspectives:* **Example** 5

3.3 Support Technology Modelling

By support technology we mean technology that support an activity of the domain but where such a technology may change as new technologies replace old ones.

6. **Example:** *Railway Point Machines:* — *Ends on page 27*

The configurational part of the rail net example, Example 2 on page 9 of Section 2, can be claimed to illustrate the intrinsics of a support technology model!

By a point machine we mean a rail junction, a rail switch, which allows routing trains along switched paths through a rail net.

Railway point machine technology has changed: from being manually thrown by a railway worker via being first remote mechanically switched (with levers and pullers), then electro-mechanically, to now being part of a so-called solid state interlock (SSI) route planning system.

The behavioural facets are different in the three cases, but the intrinsic fact of switching remains invariant.

End of *Railway Point Machines* **Example** 6

- **Note:** The examples of this paper are all drawn from the authors own work. At this stage examples would have to feature the work of others.

  Descriptions of domain technologies inherently focus on time, on failure-to-operate, and hence on safety. Various modal logics seem appropriate as modelling tools. In particular we have advocated **Duration Calculi**.

  Thus I would here, in my work use and in my lectures cover, the seminal work of Zhou Chaochen on the **Duration Calculi**. Instead I refer to published papers:

  - **Gas Burner:** [Chaochen *et al.* 1991]
  - **Steam Boiler:** [Widjaja *et al.* 1994; JuAn and XiaoShan 1995a; JuAn and XiaoShan 1995b]
  - **Railways:** [Chaochen and Huiqun 1994; et al 1994; Skakkebæk M.Sc. Thesis; Skakkebæk *et al.* 1992]
  - **Chemical Plant:** [Qiwen and Weidong 1995]
  - **Heating System:** [Qiwen and Zengyu 1996]
  - **Lift:** [Hansen 1992]

An SE Pinnacle: *Zhou Chaochen: The Duration Calculi:* The concept of the duration calculus was first proposed during the **ProCoS** project [Bjørner 1989]. The main research into the family of duration calculi has been and is being done by Zhou Chaochen [Chaochen *et al.* 1991; Hansen and Chaochen 1992; Chaochen 1993; Zhiming *et al.* 1993; Ravn *et al.* 1993; Chaochen *et al.* 1993; Chaochen and Xiaoshan 1994; Chaochen *et al.* 1995]

3.4 Rules & Regulations + User Behaviour Modelling

When the systems — for which computing support is sought — are staffed by humans and/or interface with human clients we find that *rules & regulations* have been set up to guide staff and clients in how to perform certain activities of the domain.

7. **Example:** *Public Administration &c.:* — *Ends on page 29*

Examples — which we shall not formalise — can be taken for example from a country's social welfare system, or from its taxation laws, or customs duty law, or from a fisheries infrastructure's quota directives.

End of *Public Administration* &c. **Example** 7

8. **Example:** *Banking:* — *Ends on page 29*

With each demand/deposit and savings & loan account, to take two examples, there is associated a number of contractual rules & regulations concerning the way the account client and the bank will honour the contract.

With a repayment on a loan, for example, those rules & regulations may stipulate how the bank will handle a loan installment: deduction of the principal part of the repayment from the loan, and transfer of the interest on the loan since last instalment as well as any fees to appropriate accounts of the bank. But bank tellers may make mistakes, and some, or the programmers who wrote the software for supporting this kind of processing back in the 1960's, are outright criminal and divert round offs etc. to own accounts!

Thus there is really very little we can specify, for such a possibly erroneous, possibly mischievious domain other than allowing the greatest degree of interpretation for each kind of rules & regulation.

type

Bnk, Cli, Acc, Bal, ..., Res

Cmd == ... | mk_Repay(n:**Nat**,c:Cli,a:Acc) | ...

value

M: Cmd $\xrightarrow{\sim}$ Bnk $\xrightarrow{\sim}$ Bnk × Res

End of *Banking* **Example** 8

We shall return to the issue of *rules & regulations* and *user behaviour* in the Requirements Engineering section, Section 4.1.2 (Example 11 on page 34). Then any questions, even objections that you might have to the above "looseness" should be answered, respectively satisfied. The point being made here is either: If we define too much wrt. textdbrules & regulations and *user behaviour* in the domain then we might already have entered the *requirements engineering* phase, or we might be on our way not to describe the *domain* **as it is**, but as we might wish it to be, even without computing. That is: we might be on our way to proper *business process re-engineering*.

Some SE Pinnacles: *Christopher Strachey and Dana Scott: Denotational Semantics:* Christopher Strachey's search for and contributions to the fundamentals of programming — and hence software engineering — was, alas, woefully short, but brillant [Strachey 1966; Strachey and Scott 1970; Scott and Strachey 1971; Strachey 1973; Strachey 1974; Milne and Strachey 1976]. His collaboration with Dana Scott, and Dana's independent contributions led to the concept of denotational semantics [Scott 1970b; Scott 1970a; Scott 1972c; Scott 1972a; Scott 1972b; Scott 1973; Scott 1975; Scott 1976; Scott 1981; Scott 1982a; Scott 1982b; Gunter and Scott 1990]. We consider their contributions of seminal importance and such that every professional software engineer must be awares thereof!

3.5 Discussion

We have outlined some principles of domain modelling. Emphasis was here put on abstraction in describing stake-holder perspectives, intrinsics, support technology (which was not illustrated), rules & regulations and user behaviour. Emphasis was also put on the necessity of and interplay between informal and formal descriptions.

4 REQUIREMENTS ENGINEERING

A requirements specification describes expectations to a computing system, the 'machine' [Jackson 1995] **as we would like it to be** — ie. optatively [Jackson 1995] — void of any reference to how the software might be designed,

An SE Pinnacle: *Michael Jackson: Requirements "reside" in the Domain:* Michael Jackson more than anybody else has contributed to the clarification of the concept of requirements engineering [Jackson 1994; Jackson 1995; Jackson 1997; Zave and Jackson 1997b]. Although we suggest a further refinement to Jackson's ideas we confess our dept and inspiration.

4.1 Domain Requirements

Domain requirements are those requirements which are borne directly out of the application domain. Such requirements are expressed solely in terms of domain concepts — hence can be made to stand in a formal relationship to our domain formalisation.

4.1.1 Projection

Usually we describe far more of the domain than may, at any moment be necessary in order to express specific requirements.

9. **Example:** *Air Traffic:* — *Ends on page 32*

The example domain includes that of airspace, with its airports, airdome (or controlled zone) and air-corridors, that of time-tables and that of flights.

An Air Traffic Domain: We formalise, without much ado, ie. without much narrative, but see [Bjørner 1995], the airspace (AS): airports (A), domes (D), corridors (C); the time-tables (TT) and the flight traffic (FT) — and their relationships:

type

V, A, D, Cn, C, Fn, F, T, P

AS = oas: A-**set** /∗ outside V airports of origin ∗/

× V × (A $\overrightarrow{m}$ (D × (A $\overrightarrow{m}$ Cn $\overrightarrow{m}$ C)))

× das:A-**set** /∗ outside V airports of destination ∗/

TT = A $\overrightarrow{m}$ (arrivls:(A $\overrightarrow{m}$ (Fn $\overrightarrow{m}$ T))

× departs:(A $\overrightarrow{m}$ (Fn $\overrightarrow{m}$ T)))

FT = T → (aps:AS × (Fn $\overrightarrow{m}$ P))

value

wf_TT: TT → **Bool**

wf_TT: AS → TT → **Bool**

wf_FT: FT → **Bool**

wf_FT: TT $\xrightarrow{\sim}$ FT → **Bool**

wf_FT: TT $\xrightarrow{\sim}$ T $\xrightarrow{\sim}$ FT → **Bool**

V, D, C and P are topological spaces. T is (real) time.

We only consider airport domes (D) and corridors (C) within V. To every airport (within V) there corresponds an airport dome and zero, one or more named corridors to each of a number of other airports.

The time-table concretely specifies arrival and departure flight times for each airport and indication from where, respectively to where such flights (Fn) come, respectively go.

Flights (FT) is a function from time into the positions (P), in V (or beyond) of flights Fn.

Time-tables may be internally well-formed: flights arrive appropriately before they depart, no two or more flights land within small time-intervals of one another, etc. Time-tables may be well-formed with respect to airspace: the table only lists existing airports, have flight times between airports commensurate with their distance from one another, etc. Flights may be internally well-formed: no mid-air crashes or near-misses, etc. Flights may be well-formed wrt. airspace and timetable. That is: flights depart and (especially) arrive at designated airports and within a maximum deviation of a few minites wrt. time-table specifications, etc. And flights may (or may not) be well-formed at a given time wrt. airspace and time-table. We do not define the various wf_ functions, but note that they are not invariants on the domain since time-tables may be wrong, since flights unfortunately do crash, etc.

Air Traffic Requirements: Let us assume that requirements is about scheduling and rescheduling of flights, respectively (too early or) delayed flights. And let us, for the sake of illustration, assume that flights are only rescheduled after having arrived at airports.

Now we consider only:

type

AS′ = A $\overrightarrow{m}$ (D × (A $\overrightarrow{m}$ Cn $\overrightarrow{m}$ C))

TT, FT /∗ as before ∗/

value

schedule: AS′ → TT → FT-**infset**

schedule(**as**)(tt) **as** fts

pre wf_AS(**as**) ∧ wf_TT(tt) ∧ wf_TT(**as**)(tt) ∧ ...

post forall ft:FT • ft ∈ fts ⇒ wf_FT(ft)(tt) ∧ ...

reschedule: TT → FT ⇒ T → TT × FT

reschedule(tt)(ft)(t) **as** (tt′,ft′)

pre wf_TT(tt) ∧ wf_TT(aps(ft(t)))(tt) ∧ ∼wf_FT(ft)(tt)(t)

post wf_TT(tt′) ∧ wf_TT(aps(ft′(t)))(tt′) ∧ wf_FT(ft′)(tt′)(t) ∧ ...

That is: we have projected AS onto AS′ and introduced — ie. instantiated, see next subsection — *schedule* and *reschedule* functions.

End of *Air Traffic* **Example** 9

4.1.2 Instantiation

By instantiation we mean concretisation of previously loosely defined phenomena. Here we show instantiation by "converting" abstract types (ie. sorts) into concrete types (ie. abstract data structure types).

10. **Example:** *Time-table:* — *Ends on page 34*

We continue Example 3 on page 15 of Section 3.1.1 — where we essentially had:

type TT, V, R

Cmd == mk_Query(...) | mk_Update(...) | ...

value

M: Cmd $\xrightarrow{\sim}$ TT $\xrightarrow{\sim}$ (V | TT×R)

We now *project* all of the domain description onto this stage of domain requirements. And then we *instantiate* time-tables, thereby *extending* also the "domain" with specifics on airports (A), arrival and departure times (T) and flight numbers (Fn). The instantiation introduces a notion of journey (J, two or more airport visits):

type A, T, Fn

TT = Fn $\overrightarrow{m}$ A × J

J = (art:T × A × dpt:T)*

We can now instantiate specific queries and updates:

type

Cmd == mk_Journey(fn:Fn) | ... | mk_Add(fn:Fn,j:J) | ...

R = J | TT | ...

value

M: Cmd → TT $\xrightarrow{\sim}$ R

M(mk_Journey(fn))(tt) ≡ tt(fn) **pre**: fn ∈ **dom** tt

...

M(mk_Add(fn,j))(tt) **as** tt′

pre fn ∉ **dom** tt

post wf_TT(tt′) ∧ tt′ = tt ∪ [fn ↦ j]

...

End of *Time-table* **Example** 10

11. **Example:** *Bank Scripts:* — *Ends on page 35*

We continue Example 8 on page 29. First we *instantiate* the types and then we define the *repay* operation. We simplify the example considerably as compared to [Bjørner 1997; Bjørner *et al.* 1998]:

- **First Domain Requirements Model:**

type

Bnk = Cli $\overrightarrow{m}$ (Acc $\overrightarrow{m}$ (Bal × ...))

value

M(mk_Repay(p,c,a))(b) ≡
  let (principal,interest,fee) = calc_repay(p,c,a)(b) **in**
  let (b′,r′) = subtract(principal,c,a)(b) **in**
  let (b″,r″) = add_interest(interest)(b′) **in**
  let (b‴,r‴) = add_fee(fee)(b″) **in**
  (b‴,response(r′,r″,r‴)) **end end end end**

The ... shall indicate that there is (much, much) more to a bank state than defined explicitly in *Bnk*.

Given a payment, p, a client, c, the clients' account number, a, and the bank, b, there is enough information to calculate the partitioning of the payment into principal, the inetest on the loan since last payment, and the fee. These three amounts now need be deducted from the loan balance, added to a bank owned income account (for interests), respectively added to a bank owned income account (for fees). Each operation results in partial response reports, r', r'', and r'''. These are composed and together with the resulting bank (state) forms the result of the operation.

But: **This is not the way we wish to express our domain requirements!** (And hence we did not bother to define the auxiliary *subtract* and *add* functions.) Instead we decide to require that a special script (programming) language be defined, and that operations like the *repay*ment operation be specified as scripts (ie. as programs) in that language, and that the procurers of the (in this case banking) software "sign off" on requirements specifications expressed in the abstract, albeit executable script language!

- **Second Domain Requirements Model:**

  We now respecify the *repay* operaion in terms of such a script language. We may, eg., assume that any script procedure is executed in the context of the bank (state) and most recent client and account numbers.

```
repay(payment) =
   name p = principal(payment),
          i = interest(payment),
          f = fee(payment);
   subtract p from balance,
   add i to interest,
   add f to fee;
   print report(p,i,f)
```

  We have sketched a possible script in a highly dedicated script language, one that is specially "geared" to a specific banks' account structure and the context of its operations.

  We refrain from presenting the specific syntax and semantics of the illustrated script language, but refer to an internal draft report [Bjørner *et al.* 1998].

End of *Bank Scripts* **Example** 11

A need for script languages to handle the requirements definition of domain aspects and to make precise which user behaviours are acceptable, such a need arises in many contexts:

- **Air Traffic Control:** The civil aviation authorities (CAA, CAAC, FAA, etc.), under the auspicies of the UN led CAO (Civil Aviation Organisation), have drawn up a great number of rules & regulations concerning air traffic control: ie. protocol of communication, monitoring and control necessary between aircraft pilots and ground air traffic control staff.
- **Stock Trading:** State and federal exchange commissions (FEC, etc.) have laid down rules & regulations concerning the trade, brokerage and exchange of securities, ie. procedures to be followed by buyers, sellers and floor traders of stocks etc.
- **Train Despatch:** Train despatch rules & regulations vary from province to province, from state to state. In China, for example, a rule states that at any station, at most one train can be received or depatched in any two minute interval. There are literally hundreds of more-or-less formalisable rescheduling rules, and also they vary from region to region, reflecting hard-won experience wrt. special terrain and railnet conditions.

- **Fisheries Inspection:** Rules & regulations concerning fisheries quota: allowable catch, total allowable catch, etc. are also subject to script formulations: one set of rules determine, for each sovereign state participating in a fisheries zone, its total allowable catch, and other sets of rules, for each combination of fishing grounds and fishers (or firshers' cooperatives) their quotas.
- *&c.*

Common to all these rules & regulations is that they often change and that the human computer interface via which they are presented likewise often change. Instead, therefore, of "hard wiring" the handling of these rules & regulations into the base programs of the software system to be developed, we embed a script interpreter in that system in order to allow for easy re-programming of such rules & regulations.

4.1.3 Extension

By extension, as a domain requirement, we understand, really an extension of the domain, but with properties that were infeasible without computing.

12. **Example:** *Airflight Connections:* — *Ends on page 36*

Given example 10 on page 33 we can for example suggest a query: 'connection', which given two airport names, one of origin and one of destination, find all sequences of one or more journies that lead from origin to destination.[12)] In the domain, without computing, finding all such connections is not believable.

type

Query == ... | mk_Conn(fa:A,ta:A) | ...

R = ... | (J*)**-set**

value

M(mk_Conn(fa,ta))(tt) ≡ ...

End of *Airflight Connections* **Example** 12

[12)] A precise definition of a connection between two airports might run as follows: A journey which encompasses the two airports is a direct or a non-stop connection of length 1. Let a be the name of a third airport, distinct from the origin and destination. If there is exists a connection of length m from origin to a, and a connection of length n from a to destination, then the catenation of these two connections is a connection of length $m + n$ from origin to destination.

4.1.4 Initialisation

13. **Example:** *Railway Net:* — *Ends on page 37*

Example 2 on page 9 illustrated the syntax of all railway nets. In practice delivered software applies to particular, initialised nets. These must first be input to the computer. They will then usually be kept as a database. A separate sub-system (ie. a program package) therefore has to be developed, one which supports the initial input of such a database as well as its maintenance: update, etc.

This amounts to a particular application. The domain of that application has been defined. The requirements are different from those of the originally intended requirements for railway system support (where these could have been: despatch of trains, or marshalling, or rolling stock control, etc.).

End of *Railway Net* **Example** 13

An SE Pinnacle: *Interpretation vs. Compilation: Modelling "all" versus executing "one"!:* A semantics for a programming language ascribes meaning to all well-formed programs of that language. When we capture the meaning of professional terms of an application domain, an infrastructure component, we do likewise: ascribe meaning to all instances of such terms. We treat the terms as names of syntactic categories. Interpretation leaves us that degree of freedom: to define interpreter functions which apply to instances of a whole category. An interpreter is allowed to inquire as to the type (ie. structure) of the data to which it is applied. A compiler is a specialisation of an interpreter in that all such tests (inquieries) have been "compiled out"! The strength of the domain modelling approach of this paper is exactly this: It assumes interpretation, hence it invites categorisation.

4.1.5 Discussion

We have outlined some principles and techniques of domain requirements modelling. Although listed in a sequence they are not, as is usual in the creation of software artefacts, separable, but intermingle.

4.2 Interface Requirements

4.2.1 GUI & Multi–media Interface

14. **Example:** *Airline Time-table:* — *Ends on page 39*

We continue example 10 on page 33. The commands have to be input and the results (values) displayed. There are two aspects to a command: its name and its arguments. Hence we decide, for the sake of illustration, to let the so-called Human Computer [graphical, ie. visual display (GUI)] Interface (HCI) consist of three parts:

type

HCI = CNm × prompt:Arg × result:RV
CNm == journey | connection | ... | add | ...
Arg == mk_Fn(fn:Fn) | mk_As(oa:A,da:A) | ... | mk_FnJ(fn:Fn,j:J) | ...
RV == mk_R(r:R) | mk_V(v:V)

The above is intended to model that the HCI represent a visual display screen (window) with three fields. The first field is here envisaged as an icon which, when depressed (ie. "clicked"), shows a "roll-down curtain" with command name entries. At most one of these can be selected. When a specific command name entry is selected then the *arg*ument portion of the display "lights up" and "prompts" the user for appropriate argument(s). Once provided the result/value portion (*RV*) of the HCI will [eventually] provide a response (result, value):

value

```
Mhci: HCI ~→ TT ~→ HCI
Mhci(,,) ≡
  let cnm = journey ⌈⌉ connection ⌈⌉ ... ⌈⌉ add ⌈⌉ ... in
  cases cnm:
    journey
      → let fn:Fn • ... in (journey,mk_Fn(fn),M(mk_Journey(fn))(tt)) end
    ...
    add
      → let fn:Fn • fn ∉ dom tt, j:J • ... in
          (add,(mk_FnJ(fn,j)),M(mk_Update(fn,j))(tt)) end
    ... end end
```

We have achieved, we believe, our aim: to hint at how to model HCI's, albeit abstractly (*Mhci*), and as a step of development from earlier requirements definitions (viz.: M).

The internal non-deterministic choice of command names and arguments shall model that we do not know which commands and what arguments are selected by users.

End of *Airline Time-table* **Example** 14

4.2.2 User Dialogue

By a dialogue we understand a sequence of one or more HCI interactions.

15. **Example:** *Banking:* — *Ends on page 39*

We continue Example 8 on page 29. With respect to for example a demand deposit account and ordinary client "undergoes", ie. is subject to the following regular expression of transactions:

$$e\ (\ d \mid w \mid s\ )^*\ c$$

where e, d, w, s and c designate establish, deposit, withdraw, (request) statement, respectively close transactions.

This is a much simplified picture. To capture more sophisticated dialogues and to capture their context-dependency (viz.: withdrawals may not be possible if the account is [excessively] negative, etc.) regular expressions may not suffice. Instead one may use **CSP** [Hoare 1978; Hoare 1985; Roscoe 1997], **StateChart**s [Harel 1987; Harel *et al.* 1987; Harel *et al.* 1990], or other techniques and tools.

End of *Banking* **Example** 15

4.2.3 "User Friendliness"

Many properties of software may qualify for the predicate: "User Friendly". Here we shall just focus on one property (on one set of properties):

- *A software system, a program package, is said to be "user friendly" if, besides possibly other properties, it exhibits, reflects, the concepts of the domain in which it serves, only these, and in an "isomorphic" manner.*

It is probably impossible to attain "100 % user friendliness"! By the *software exhibiting, reflecting a domain concept* we mean to say that the two-way human computer interface somehow prompts or displays that domain concept in a texttual, visual or animated fashion and void of any "adornment" with computing system concepts.

Our example may first be thought of as being far from the requirements notion of "friendliness". First, the example is a domain description. And we are in the section on requirements !

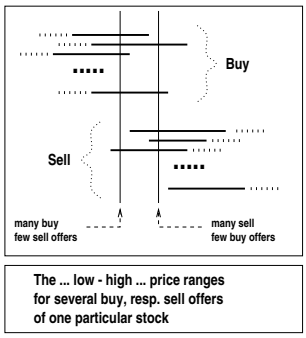
16. **Example:** *Stock Exchange:* — *Ends on page 45*

We model core concepts of a domain of securities (stock and bond) trading.

Domain-wise we shall model a simple stock exchange. 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 intution we "suggest" some concrete types for all sorts, but they are only suggestions. 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 Sll, 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 5: A "Snapshot" Stock Exchange



type

SE, Buy, Sll, ClRm, S, O, Ofrs, Ofr, T, Q, P, R, Clrd, Rmvd

SE = (Buy × Sll) × ClRm

Buy, Sll = 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

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 oberver"! Given the suggested concrete types for the correspondingly named abbstract 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_Sll: SE → Sll,

obs_ClRm: SE → ClRm

obs_Ss: (Buy|Sll) → S-**set**

obs_Ofrs: S × (Buy|Sll) $\xrightarrow{\sim}$ 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 guding price interval (lo,hi), such that the offer is valid in some time (bt,et) interval.[13)]

[13)] 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

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,

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.

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), sll = obs_Sll(se) in
    s ∈ obs_Ss(buy) ∩ obs_Ss(sll)
    ∧ bs ⊆ obs_Ofrs(s,buy) ∧ ss ⊆ obs_Ofrs(s,sll)
    ∧ buysll(p,bs,ss)(t)
    ∧ let (bq,sq) = (obs_Qs(bs),obs_Qs(ss)) in
        acceptable_difference(bq,sq,s,se) end end

  buysll: P×Ofrs×Ofrs → T → Bool
  buysll(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), sll = obs_Sll(se) in
    let buy′ = rm_Ofrs(s,bs,buy), sll′ = rm_Ofrs(s,ss,sll) in
    obs_Buy(se′) =
      if bq > sq
        then updbs(buy′,s,bq−sq,tt_buy(s,bq−sq)(t,se))
        else buy′ end ∧
    obs_Sll(se′) =
      if bq < sq
        then updss(sll′,s,sq−bq,tt_sll(s,bq−sq)(t,se))
        else sll′ 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. Thta 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 expresion, 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!

"User Friendliess": Finally we can discuss the issue of "user friendliness". What was presented so far in this example was some facet of the domain of stock (and bond) trading.

With respect to "user friendly" interface requirements for computing support of these facets we now, informally, state:

- *. . . The software for the support of stock exchange actions shall be limited to visualising the "state", for each stock, of buy and sell offers, viz.: Figure 5 on page 40, and to display exactly and only when buy and sell offers are input, withdrawn and effected . . .*

Our elaborate domain model above thus explicates what can be illustrated, and what cannot: only the concepts shown by this model: placement and withdraw commands, and the circumstances surrounding the *next* function: the predicates *pclr*, *apclr*, *prm* and functions *aclr*, *srm*, including the non-deterministic choices offered.

End of *Stock Exchange* **Example** 16

4.2.4 Discussion

Domain and interface requirements find their solution, usually, in the first stage of software design, the one we shall call software architecture design.

In contrast machine requirements, to be dealt with next, usually have their solution "postponed" till a stage after: the program organisation, also know as the program structuring stage.

4.3 Machine Requirements

Sometimes machine requirements are referred to as non-functional requirements. In order to enable a clearer distinction, and to use meaningful terms, we have chosen the term 'machine requirements'.

Usually machine requirements are "of the same kind" — "from application to application"! The next subsections list some while illustrating the dilemma: that oftentimes machine requirements cannot easily be formalised — and when so, it is often, as we shall see in Section 5.2, difficult to "derive" from such a formalisation a suitable program organisation.

4.3.1 Performance

Performance is an issue that we can only relate to the machine. Usually performance requirements are expressed, informally, in the form: *response time to such-and-such a query, or event, must be less than μ seconds ...*

4.3.2 Dependability

Dependability concerns the machine issues of *accessability*, *availability*, *security*, *safety* etc. One should not confuse, for example the *machine requirements* issue of *security* with those of a possible *domain requirements* issue of *security*. The latter **exists** in the domain, prior to computing; the former **arises** as a consequence of introducing computing. Here we deal only with the *machine* consequence.

We illustrate three aspects of machine dependability.

17. **Example:** *Accessability:* — *Ends on page 47*

We continue Examples 8 on page 29 and 15 on page 39.

To present an example of *accessability* we first need express the *platform* (ie. *system configuration*) requirements of, for example: *"there is one bank, with a definite number of automatic teller machines (ATM), but an indefinite number of clients. The computing system is to give the appearance that it can simultaneously serve such a number of clients.* Now *accessability* is the issue of "simulataneous access".

We may formalise this as follows.

There is a definite, "sufficiently large" number of 'client ATM' processes, and one 'bank' process:

value

system() ≡ (‖ { client(i) | i:CIdx }) ‖ bank()

channel

ctb[i:CIdx] Cmd, btc[i:CIdx] Res

In this case we could say that the above predicates a software architecture, see Section 5.1, which we could visualise as shown in Figure 6 on page 51.

End of *Accessability* **Example** 17

18. **Example:** *Availability:* — *Ends on page 47*

In continuation of Example 17 on the facing page we state: *availability* ensures that failure in bank processing is masked by replication of bank handling resources.

Figure 7 on page 52 of the 'Program Organisation' Section 5.2 indicates our design decisions wrt. this machine requirements.

End of *Availability* **Example** 18

19. **Example:** *Security:* — *Ends on page 47*

In continuation of Example 18 we state: *Only authorised clients are given access to the bank ATMs.*

Figure 8 on page 52 of the 'Program Organisation' Section 5.2 indicates our design decisions wrt. this machine requirements.

End of *Security* **Example** 19

4.3.3 Maintenance

Three forms of maintenance requirements can be isolated: *adaptive* — preparing the software for future adaptations to other external equipment and other software, *perfective* — preparing the software for future improvement wrt. performance, once resource "bottlenecks" can be identified, and *corrective* — preparing the software for future "debugging" should domain conception, requirements elicitiation or software design mistakes be uncovered.

20. **Example:** *Adaptive Maitenance:* — *Ends on page 48*

In continuation of Example 19 on the preceding page we state: *ATM and authorisation technology changes from time to time. The software must be resilient to such support technology changes.*

Figure 9 on page 52 of the 'Program Organisation' Section 5.2 indicates our design decisions wrt. this machine requirements.

End of *Adaptive Maitenance* **Example** 20

4.3.4 Platforms

Development as well as execution, including portability platforms must likewise be requirements targeted.

4.3.5 Discussion

We have briefly outlined issues and aspects of machine requirements. We refrained from formalising these — as that will be amply illustrated during *program organisation design*. See Section 5.2.

4.4 Discussion

We have shown how requirements "split" into several domain, interface and machine aspects. We have noted that domain and some interface requirements lend themselves to formalisation in terms of domain concepts.

5 SOFTWARE DESIGN

Software specification describes the design of software **as it is going to be** — ie. imperatively [Jackson 1995].

Software design consists of several stages: *software architecture* design, *program organisation* design, one or more steps of *concretisation design*s (*refinement*s), finally concluded by *coding*, ie. *implementaiton* in terms of existing *platform*s (**CORBA, Java** etc.).

5.1 Software Architectures

A *software architecture* is defined as the set of concepts and facilities offered the user of the software (humans, or other software). *Software architecture* design therefore addresses issues of domain and interface requirements.

21. **Example:** *A Distributed Process Architecture:* — *Ends on page 50*

We continue Example 14 on page 38. The *Mhci* function of that example clearly stated what was required. In order to express a requirement (stated somewhere!) that all query as well as update users share the same time-table, we suggest in the *software architecture* to provide for a(n in)definite number of user processes and one time-table process. Where the *Mhci* function had *TT* in its signature, the *user* process has **Unit** in its signature. Where the *Mhci* function body referred to *tt* (as in *M(...)(tt)*) an **output** from the *user* process is specified to the *time_table* process followed immediately by an **input** from that process. And vice-versa: the *time_table* process expects, amongst others, **input** from *user* processes, and reply with **output**s.

```
type
  Φ = TT → (TT×R | V)

channel
  utt[ i|i:UIdx ] Φ
  ttu[ i|i:UIdx ] (R|V)

value
  system: Unit → Unit
```

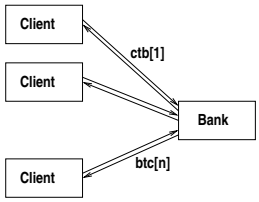
```
system() ≡ ‖ { user(i) | i:UIdx } ‖ time_table(tt)

user: HCI → UIdx → out Φ in (R|V) Unit
user(,,)(i) ≡
  let cnm = journey ⌈] connection ⌈] ... ⌈] add ⌈] ... in
  cases cnm:
    journey
      → let fn:Fn • ... in
          (journey,mk_Fn(fn),utt[i]!M(mk_Journey(fn));ttu[i]?) end
    add
      → let fn:Fn • fn ∉ dom tt, j:J • ... in
          (add,mk_FnJ(fn,j),utt[i]!M(mk_Update(fn,j));ttu[i]?) end end end

time_table: TT → in Φ out RES Unit
time_table(tt) ≡
  let tt'' =
    ...
    [] { let φ = utt[i]? in let (tt',rv) = φ(tt) in ttu[i]!rv ; tt' end end | i:UIdx }
    ... in
  time_table(tt'') end
```

End of *A Distributed Process Architecture* **Example** 21

In the next example we do not show any formulae. Instead we show a process/channel diagram. By changing box and arow labels from Client to user, Bank to time_table, etc., we arrive at a diagram which could as well fit the example just completed! And hence: vice-versa: It should now be easy to write down the formulae for the enxt example!

This "proves" a point: That software architectures have "styles" that "cut across domain specifics" [Abowd *et al.* 1993; Abowd *et al.* 1995; Garlan 1996].

Figure 6: Banking System Software Architecture



An SE Pinnacle: *David Garlan: Software Architecture:* David Garlan and his colleagues (Gregory Abowd, Robert Allen, Mary Shaw, et al.) [Allen and Garlan 1992; Abowd *et al.* 1993; Garlan and Shaw 1993; Allen and Garlan 1994; Shekaran *et al.* 1994; Abowd *et al.* 1995; Garlan 1995; Allen and Garlan 1996; Garlan 1996] must be credited with bringing new insight to and delineating, as a separable study object that of *software architecture*. Our definition of software architecture "narrows" that of Garlan's. Together with our definition of *program organisation*, see Section 5.2, we "equal" Garlan's definition.

22. **Example:** *Banking:* — *Ends on page 51*

We continue Example 17 on page 46 and its predecessor.

All domain and interface requirements are, and one (machine) **accessability** requirements is now consolidated into the picture of Figure 6 for which we do not show the synopsis, narrative, terminology nor formalisation. Instead we refer to Example 21 on page 49 and the comments made right after that example.

The CSP process definitions and channel declarations, with types, was given on page 47.

End of *Banking* **Example** 22

5.2 Program Organisation

Program Organisation design especially addresses issues of machine requirements.

A *program organisation* is therefore defined as the *software architecture* plus the set of concepts and facilities offered at internal interfaces, ie. a *program organisation* includes the further decomposition of software into internal, possibly replicated or distributed components (objects, processes, data structures).

23. **Example:** *Banking:* — *Ends on page 53*

Figure 7: Distributed Banking

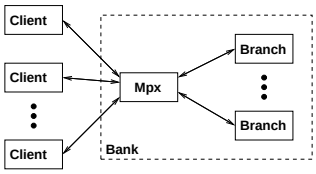
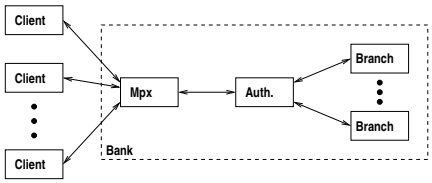


Figure 8: Secure Banking



We continue Example 22 on the preceding page while now focusing on the machine requirements of Examples 18 on page 47, 19 on page 47 and 20 on page 48.

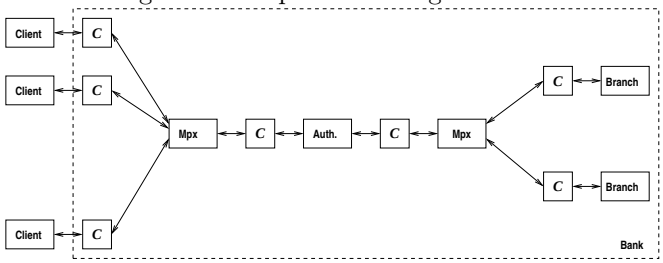
Availability : The *availability machine requirements* finds its resolution, as shown in Figure 7: The bank has been replicated into a possibly distributed set of branches. A multiplexor (*Mpx*) finds, for each client transaction an appropriate branch.

Security : The *security machine requirements* finds its resolution, as shown in Figure 8.

An additional *authorisation* "mechanism" "filters" transaction requests based on further requirements specified (but here undefined) authorisation (say password etc.) notions.

Adaptability : The *adaptive maintenance* (ie. machine) *requirements* finds its resolution, as shown in Figure 9: each of the specialised "boxes" (software components, processes, data structures) are connected to one abother through special *connectors*. These serve the purpose of "standardising" component interfaces. We refer to [Allen and Garlan 1994; Abowd *et al.* 1995; Garlan 1996] for more on the *connector, glue, port* etcetera issues of this aspect of *program organisation*.

Figure 9: Adaptive Banking Software



We refrain from formalising the above into proper CSP specifications.

End of *Banking* **Example** 23

Some SE Pinnacles: *Futamura, Ershov, Neil Jones and Cousot: Abstract Interpretation:* The notions of **partial evalation, mixed computation** [Bjørner *et al.* 1988] and **abstract interpretation** coincide. They all deal with various forms of execution over mathematical structures. That is: one and the same structure may give rise to a variety of interpretations. A program text is subjected to many forms of partial evaluation in the course of being compiled into for example efficient code: lexical scanning, syntactical parsing, type analysis, data flow analysis, control flow analysis, etc.

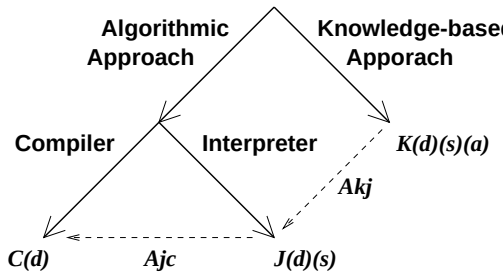
It was Y. Futamura [Futamura 1971; Futamura *et al.* 1991] who first expressed a hierarchy of laws concerning partial evaluation. Independently Harald Andersson [Haraldsson 1977] discovered similar notions of partial evaluation. Andrei P. Ershov started the Russian School (we mention but a few:) [Ershov 1977; Ershov *et al.* 1988],of mixed computation.

Neil D. Jones gathered this and contributed significantly to the "clean-up" of notions (again we mention but a very few:) [Jones *et al.* 1993].Patrick and Nadia Cousot re-interpreted the partial evaluation cum mixed computation as abstract interpretation (we mention aain a few:) [Cousot and Cousot 1977; Cousot 1996; Cousot 1997b; Cousot 1997a]. We find that the notion of abstract interpretation — as we prefer to label the entire area referred to above — is in no way near being exploited; that there are many exciting applications of abstract interpretation yet to be uncovered, and more to be applied!

5.3 Discussion

We have illustrated how software architectures "develops" into program organisations, adopting increasingly more machine requirements. Complexity is mastered through separation of concerns.

Figure 10: Algorithmic "vs." Knowledge–based Programs



An SE Pinnacle: *Algorithmic vs. Knowledge-based Systems:* This note is in continuation of the "pinnacle" note on *Interpretation vs. Compilation* page 37 In [Bjørner and Nilsson 1992] we investigated the classical Algorithmic approach to software design in contrast to the Knowledge-based approach. See Figure 10 on the page before. In — the extreme — knowledge-based programs contain all the domain knowledge embedded in a, the strucure of the information at hand in s and the particular data values in d. K — the inference machine — is a rather general purpose program — in theory applicable across widely different domains! In — again in the extreme — algorithmic interpreters the domain knowledge (a) is embedded in the interpreter J, retaining s and d as arguments. And — again in the extreme — in compiled programs both the domain knowledge (a) and the (type) structuring (s) is embedded in the program C. The relations between K, Js and Cs can be expressed in terms of abstract interpreters (A), see "pinnacle" box on page 53. These are shown as A_{kj}, respectively A_{jc}.

6 CONCLUSION

In the introductory section we contrasted, sharply, the world of software engineering to the classical, traditional worlds of electronics, automotive, chemical, civil and other engineering.

In this concluding section we will now, on the background of the "evidence" conjured by intervening sections, take up this thread.

In the introductory section we very briefly outlined a facet of the conceptual nature of engineering and of engineers. We now continue: Engineers (including software engineers) produce components — resorting to the occasional, or, as for software engineers, the systematic use of mathematics. Technicians put such boxes together, essentially without a need for understanding the mathematics of these. Technologists are usually former engineers. They direct the R&D of new technologies. In their pursuit of this, technologists do not resort to the sciences, but rather to the market: To economic, social and other non-natural science issues.

A computing scientist studies, explores and experiments with either domains or technologies (recall: the engineer "walks the bridge between science & technology — both ways!"). The software engineer in charge of, or deeply pursuing for example domain engineering, besides doing so in an assumed tight collaboration with stake-holder professionals of that domain, is also expected, for years to come, to be a computing scientist.

We have given a number of examples, stressing the domain point-of-view as we believe it is the more novel. These example have necessarily been sketchy, but as the references show: There is much work behind these brief examples. However brief they are, they cover substantial fields, but

our comments, explications and observations must be brief. We expect the reader to study them carefully and to draw the conclusion that domain modelling is a rich field, much more need be done: Modelling principles, techniques and tools need be established and area-by-application-area one need establish wide and deep such models.

From the outline, therefor,e of main software engineering tasks, as presented in Sections 2–5, we conclude, rather crudely perhaps, the following:

- **Universes of Material Quantity:**

  The worlds of traditional engineering, insofar as they practice the classical, ie. the non-IT core of their profession, represent "universes of material quantity": The engineering goals are those of designing and providing the way for constructing artefacts that were either smaller, or bigger, cheaper, faster, or other such measures.

- **Universes of Intellectual Quality:**

  The worlds modern software engineering, insofar as they aim at the development of new software for the computing support of new infrastructure component activities, represent "universes of intellectual quality": The engineering goals are those of designing, cum constructing software that is "better", correct, fit ("hand-in-glove") human activities (ie. the domain), pleasing, etc.

The above view, we think, is a major characteritics of software engineering. Surely there are others. But we think that our view is at the heart of the profession.

Other, complimentary and cohesive views are possible. Some will undoubtedly be presented in the current issue of *Annals of Software Engineering*. Together the reader may find what such readers like to find!

I have listed some pinnacles of software engineering in separate, framed boxes scattered over the text.

Perhaps, in all modesty (!), some of the examples may point to other forms of pinnacles: more down-to-earth, everyday ones, but of the kind that makes being a software engineer perhaps the most exciting profession so far created by man.

7 ACKNOWLEDGEMENTS

The author, besides acknowledging his indebtitude to the many scientists who have been mentioned in the various "pinaccle" boxes of this paper, and in a somewhat overlapping way,

explicitly acknowledges the following colleagues, for or with whom I have worked, and who over the years have greatly influenced my thinking and acting: Gerald M. Weinberg, Peter Lucas, Gene Amdahl, John W. Backus, Lotfi Zadeh, E.F. (Ted) Codd, Cliff B. Jones, Heinz Zemanek, the late Andrei Ershov, Hans Langmaack, Andrzej Blikle, Neil D. Jones, Jørgen Fischer Nilsson, Søren Prehn, C.A.R. (Tony) Hoare, Michael Jackson, Zhou Chaochen and Chris George — listed more-or-less chronologically.

REFERENCES

Abadi, M. and L. Cardelli (1996), *A Theory of Objects*, Springer-Verlag, New York, NY, USA.

Abowd, G., R. Allen, and D. Garlan (1993), "Using style to understand descriptions of software architecture," *SIGSOFT Software Engineering Notes 18*, 5, 9–20.

Abowd, G., R. Allen, and D. Garlan (1995), "Formalizing style to understand descriptions of software architecture," *ACM Transactions on Software Engineering and Methodology 4*, 4, 319–364.

Abrial, J. (1980), "(1) The Specification Language Z: Basic Library, 30 pgs.; (2) The Specification Language Z: Syntax and "Semantics", 29 pgs.; (3) An Attempt to use Z for Defining the Semantics of an Elementary Programming Language, 3 pgs.; (4) A Low Level File Handler Design, 18 pgs.; (5) Specification of Some Aspects of a Simple Batch Operating System, 37 pgs." Internal reports, Programming Research Group.

Abrial, J.-R. (1996), *The B Book: Assigning Programs to Meanings*, Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, England.

Allen, R. and D. Garlan (1992), "A formal approach to software architectures," In *IFIP Transactions A (Computer Science and Technology); IFIP Wordl Congress; Madrid, Spain*, volume vol.A-12, IFIP, North Holland, Amsterdam, Netherlands, pp. 134–141.

Allen, R. and D. Garlan (1994), "Formalizing architectural connection," In *16th International Conference on Software Engineering (Cat. No.94CH3409-0); Sorrento, Italy*, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, pp. 71–80.

Allen, R. and D. Garlan (1996), "A case study in architectural modeling: the AEGIS system," In *8th International Workshop on Software Specification and Design; Schloss Velen, Germany*, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, pp. 6–15.

an Airchinnigh, M. M. (1991), "Tutorial Lecture Notes on the Irish School of the VDM," In *VDM'91 – Formal Software Development Methods*, S. Prehn and W. Toetenel, Eds., Springer-Verlag, pp. 141–237, LNCS 552.

Anh, D. N. and R. Moore (1996), "Formal Modelling of Large Domains — with an Application to Airline Business," Technical Report 74, UNU/IIST, P.O.Box 3058, Macau, Revised: September 1996.

Anon (1980 – 1985), *C.C.I.T.T. High Level Language (CHILL), Recommendation Z.200, Red Book Fascicle VI.12*, See [Haff 1981], ITU (Intl. Telecmm. Union), Geneva, Switzerland.

Bekić, H. (1984), "Programming Languages and Their Definition," In *Lecture Notes in Computer Science, Vol. l77*, C. Jones, Ed., Springer-Verlag.

Bekić, H., D. Bjørner, W. Henhapl, C. Jones, and P. Lucas (1974), "A Formal Definition of a PL/I Subset," Technical Report 25.139, IBM Laboratory, Vienna.

Bergstra, J., J. Heering, and P. Klint (1989), *Algebraic Specification*, Addison-Wesley, ACM Press.

Bjørner, D., Ed. (1980), *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, Springer-Verlag.

Bjørner, D. (1986a), "Project Graphs and Meta-Programs: Towards a Theory of Software Development," In *Proc. Capri '86 Conf. on Innovative Software Factories and Ada, Lecture Notes on Computer Science*, N. Habermann and U. Montanari, Eds., Springer-Verlag.

Bjørner, D. (1986b), "Software Development Graphs — A Unifying Concept for Software Development?" In *Vol. 241 of Lecture Notes in Computer Science: Foundations of Software Technology and Theoretical Computer Science*, K. Nori, Ed., Springer-Verlag, pp. 1–9.

Bjørner, D. (1987), "The Stepwise Development of Software Development Graphs: Meta-Programming VDM Developments," In *See [Bjørner* et al. *1987b]*, volume 252 of *LNCS*, Springer-Verlag, Heidelberg, Germany, pp. 77–96.

Bjørner, D. (1989), "A ProCoS Project Description," *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Procedings: Intl. Conf. on AI & Robotics, Strebske Pleso, Slovakia, Nov. 5-9, 1989, North-Holland, Publ.*, Dept. of Computer Science, Technical University of Denmark.

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

Bjørner, D. (1996), "Models of Enterprise Management: Strategy, Tactics & Operations — Case Study Applied to Airlines and Manufacturing," Technical Report 60, UNU/IIST, P.O.Box 3058, Macau.

Bjørner, D. (1997), "Models of Financial Services & Industries," Research Report 96, UNU/IIST,

P.O.Box 3058, Macau, Incomplete Draft Report.

Bjørner, D. (1997–1998), "Domains as a Prerequisite for Requirements and Software — Domain Perspectives & Facets, Requirements Aspects and Software Views," Springer-Verlag, LNCS vol.1526, pp 1–42, Editor: Manfred Broy. Intl. workshop on *Requirements Targeting Software and Systems Engineering,* Bernried am Starnberger See, Bavaria, Germany, 12–14 October 1997.

Bjørner, D. (1999), "Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re–assessment of Software Engneering ?" *South African Journal of Computer Science* Editor: Chris Brink.

Bjørner, D. and J. R. Cuéllar (1998), "Software Engineering Education: Rôles of formal specification and design calculi," *Annals of Software Engineering 6*, 365–410, Published April 1999.

Bjørner, D., A. Ershov, and N. Jones, Eds. (1988), *Partial Evaluation and Mixed Computation*, Gl. Avernæs, Denmark, IFIP TC2 Working Conference, North-Holland Publ.Co., Amsterdam.

Bjørner, D., C. George, B. Hansen, H. Laustrup, and S. Prehn (1997), "A Railway System, Coordination'97, Case Study Workshop Example," Research Report 93, UNU/IIST, P.O.Box 3058, Macau.

Bjørner, D., C. Hoare, and H. Langmaack, Eds. (1990), *VDM and Z – Formal Methods in Software Development*, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Springer-Verlag, Lecture Notes in Computer Science, Vol. 428.

Bjørner, D. and C. Jones, Eds. (1978), *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*, Springer-Verlag.

Bjørner, D. and C. Jones, Eds. (1982), *Formal Specification and Software Development*, Prentice-Hall.

Bjørner, D., C. Jones, M. M. an Airchinnigh, and E. Neuhold, Eds. (1987a), *VDM – A Formal Method at Work*, Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer-Verlag, Lecture Notes in Computer Science, Vol. 252.

Bjørner, D., C. Jones, M. M. an Airchinnigh, and E. Neuhold, Eds. (1987b), *VDM – A Formal Method at Work*, Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer-Verlag, Lecture Notes in Computer Science, Vol. 252.

Bjørner, D. and M. Nielsen (1985), "Meta Programs and Project Graphs," In *ETW: Esprit Technical Week*, Elsevier, pp. 479–491.

Bjørner, D. and J. Nilsson (1992), "Algorithmic & Knowledge Based Methods — Do they "Unify" ? — with some Programme Remarks for UNU/IIST," In *International Conference on Fifth Gen-*

eration Computer Systems: FGCS'92, ICOT, pp. (Separate folder, "191–198").

Bjørner, D. and O. Oest (1980a), "The DDC Ada Compiler Development Project," *[Bjørner and Oest 1980b]* , 1–19.

Bjørner, D. and O. Oest (1980b), *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*, Springer-Verlag, This book became a forerunner for the unpublished — but available — Formal Definition of Ada researched and developed by a Danish/Italian team during 1984–1985 (involving, amongst others Egidio Astesiano, Hans Bruun and Jan Storbank Pedersen — besides the author of these lecture notes).

Bjørner, D., V. Rosario, and M. Helder (1998), "A Normative Model of Concrete banking Operations — Banking Rules & Regulations and Staff/Client Behaviours," Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark.

Bloomfield, R., L. Marshall, and R. Jones, Eds. (1988), *VDM – The Way Ahead*, Proc. 2nd VDM-Europe Symposium 1988, Dublin, Ireland, Springer-Verlag, Lecture Notes in Computer Science, Vol. 328.

Burstall, R. and J. Goguen (1977), "Putting Theories together to Make Specifications," In *Proc. of (IJCAI) Int'l. Joint Conf. on AI*, Boston.

Burstall, R. and J. Goguen (1980), "The Semantics of CLEAR: A Specification Language," *[Bjørner 1980]* , 292–332.

Cardelli, L. (1987), "Basic Polymorphic Type-checking," *Science of Computer Programming 8*, 2, 147–172.

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

Chaochen, Z. (1993), "Duration Calculi: An Overview," In *Proceedings of Formal Methods in Programming and Their Applications, D. Bjørner, M Broy, and I.V. Pottosin (Eds.)*, LNCS 735, Springer-Verlag, pp. 256–266.

Chaochen, Z., C. Hoare, and A. Ravn (1991), "A Calculus of Durations," *Information Processing Letters 40*, 5, 269–276.

Chaochen, Z. and Y. Huiqun (1994), "A duration Model for Railway scheduling," Technical Report 24b, UNU/IIST, P.O.Box 3058, Macau.

Chaochen, Z., D. V. Hung, and L. Xiaoshan (1995), "A Duration Calculus with Infinite Intervals," In *Fundamentals of Computation Theory, Horst Reichel (Ed.)*, LNCS 965, Springer-Verlag, pp. 16–41.

Chaochen, Z., A. Ravn, and M. Hansen (1993), "An Extended Duration Calculus for Hybrid Systems," In *Hybrid Systems*, R. Grossman, A. Nerode, A. Ravn, and H. Rischel, Eds., volume 736 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 36–59.

Chaochen, Z. and L. Xiaoshan (1994), "A Mean Value Calculus of Durations," In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, A. Roscoe, Ed., Prentice Hall International, pp. 431–451.

Clemmensen, G. and O. Oest (1984), "Formal Specification and Development of an Ada Compiler – A VDM Case Study," In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, IEEE, pp. 430–440.

Cousot, P. (1996), "Abstract Interpretation," *ACM Computing Surveys 28*, 2, 324–328.

Cousot, P. (1997a), "Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract)," *Theoretical Computer Science 6*, 25, Electronic Notes: `http://www.elsevier.nl/locate/entcs/volume6.html`, Elsevier Science.

Cousot, P. (1997b), "Design of Semantics by Abstract Interpretation," In *Mathematical Foundations of Programming Semantics*, MFPS XIII, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, USA.

Cousot, P. and R. Cousot (1977), "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," In 4^{th} *POPL*, Principles of Programming and Languages, ACM Press, pp. 238–252.

Dahl, O.-J., E. Dijkstra, and C. Hoare (1972), *Structured Programming*, Academic Press.

Dijkstra, E. (1975), "Guarded Commands, Non-Determinacy and Formal Program Derivation," *Communications of the ACM 18*, 8, 453–457.

Dijkstra, E. (1976), *A Discipline of Programming*, Prentice-Hall.

Dung, D. T., L. L. Chi, N. L. Thu, P. P. Nam, T. M. Lien, and C. George (1996), "Developing a Financial Information System," Technical Report 81, UNU/IIST, P.O.Box 3058, Macau.

Ehrig, H. and B. Mahr (1985), *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag.

Ehrig, H. and B. Mahr (1990), *Fundamentals of Algebraic Specification 2, Module Specifications and Constraints*, EATCS Monographs on Theoretical Computer Science, vol. 21, Springer-Verlag.

Engeler, E. (1971), *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, Springer-Verlag.

Ershov, A. (1977), "On the Essence of Translation," *Computer Software and System Programming*

3, 5, 332–346.

Ershov, A., D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, Eds. (1988), *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*, Ohmsha Ltd. and Springer-Verlag.

et al, Y. X. (1994), "Stability of Railway Systems," Technical Report 28, UNU/IIST, P.O.Box 3058, Macau.

et al. (Eds.), M. B. (1991), *Algebraic system specification and development A survey and annotated bibliography*, Lecture notes in computer science, vol. 501, Springer-Verlag.

Fitzgerald, J., C. B. Jones, and P. Lucas, Eds. (1997), *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, Springer-Verlag, ISBN 3-540-63533-5.

Fitzgerald, J. and P. G. Larsen (1997), *Developing Software using VDM–SL*, Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England.

Futamura, Y. (1971), "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler," *Systems, Computers, Controls 2*, 5, 45–50.

Futamura, Y., K. Nogi, and A. Takano (1991), "Essence of Generalized Partial Computation," *Theoretical Computer Science 90*, 1, 61–79, Also in: D. Bjørner and V. Kotov: Images of Programming, North-Holland, 1991.

Futatsugi, K. and R. Diaconescu (????), *CafeOBJ Report — Definition of the Language*, To appear in 1999 as book in the AMAST series at World Scientific Publs., Singapore.

Futatsugi, K., J. Goguen, J.-P. Jouannaud, and J. Meseguer (1985), "Principles of OBJ-2," In *12th Ann. Symp. on Principles of Programming*, ACM, pp. 52–66.

Garlan, D. (1995), "Research directions in software architecture," *ACM Computing Surveys 27*, 2, 257–261.

Garlan, D. (1996), "Formal approaches to software architecture," In *Studies of Software Design. ICSE '93 Workshop. Selected Papers*, Springer-Verlag, Berlin, Germany, pp. 64–76.

Garlan, D. and M. Shaw (1993), *An introduction to software architecture*, World Scientific, Singapore, pp. 1–39.

Gaudel, M.-C. and J. Woodcock, Eds. (1996), *FME'96: Industrial Benefit and Advances in Formal Methods*, Springer-Verlag.

George, C. (1995), "A Theory of Distributing Train Rescheduling," Research Report 51, UNU/IIST, P.O.Box 3058, Macau, Published in: Marie-Claude Gaudel and James Woodcock (eds.), *FME'96:*

Industrial Benefit and Advances in Formal Methods, LNCS 1051, Springer-Verlag, 1996, pp. 499–517.

George, C. W., T. Janowski, and R. Moore (1995), "Domain Analysis for a Budgetary System," Technical note, UNU/IIST, P.O.Box 3058, Macau.

Girard, J.-Y., Y. Lafont, and P. Taylor (1989), *Proofs and Types*, volume 7, Cambridge Tracts in Theoretical Computer Science Edition, Cambridge Univ. Press, Cambridge, UK.

Goguen, J., J. Thatcher, E. Wagner, and J. Wright (1975), "Abstract Data Types as Initial Algebras and Correctness of Data Representations," In *ACM Conf. on Computer Graphics*, pp. 89–93.

Goguen, J., J. Thatcher, E. Wagner, and J. Wright (1977), "Initial Algebra Semantics and Continuous Algebras," *Journal of the ACM 24*, 1, 68–95.

Goguen, J., J. Thatcher, E. Wagner, and J. Wright (1978), "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," In *Current Trends in Programming Methodology*, R. Yeh, Ed., Prentice-Hall.

Group, T. R. L. (1992), *The RAISE Specification Language*, The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England.

Group, T. R. M. (1995), *The RAISE Method*, The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England.

Guessarian, I. (1981), *Algebraic semantics*, Lecture notes in computer science, vol. 99, Springer-Verlag.

Gunter, C. and D. Scott (1990), "Semantic Domains," In *[van Leeuwen 1990] — vol.B.*, J. Leeuwen, Ed., North-Holland Publ.Co., Amsterdam, pp. 633–674.

Haff, P., Ed. (1981), *The Formal Definition of CHILL*, See [Anon 1980 – 1985], ITU (Intl. Telecmm. Union), Geneva, Switzerland.

Haff, P. and A. Olsen (1987), "Use of VDM within CCITT," In *[Bjørner* et al. *1987a]*, Springer-Verlag, pp. 324–330.

Hansen, K. M. (1992), "Designing a Lift Control System," Technical Report ProCoS, Dept. of Computer Science, Technical University of Denmark.

Hansen, M. and Z. Chaochen (1992), "Semantics and Completeness of Duration Calculus," In *Real-Time: Theory in Practice, REX Workshop*, J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, Eds., volume 600 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 209–225.

Haraldsson, A. (1977), "A Program Manipulation System Based on Partial Evaluation," Ph.D.

thesis, Linköping University, Sweden, Linköping Studies in Science and Technology Dissertations 14.

Harel, D. (1987), "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* .

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot (1990), "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng. 16*, 4, 403–414.

Harel, D., A. Pnueli, J. Schmidt, and R. Sherman (1987), "On the Formal Semantic of Statecharts," In *Proc. of IEEE Symp. on Logic in Computer Science*.

Haß, M. (1987), "Development and Application of a Meta IV Compiler," In *[Bjørner* et al. *1987a]*, Springer-Verlag, pp. 118–140.

Hayes, I. J., Ed. (1987), *Specification Case Studies*, International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK.

Hoare, C. (1972), "Notes on Data Structuring," In *[Dahl* et al. *1972]*, pp. 83–174.

Hoare, C. (1978), "Communicating Sequential Processes," *Communications of the ACM 21*, 8.

Hoare, C. (1985), *Communicating Sequential Processes*, Prentice-Hall International.

Hoare, C. and et al. (1987), "Laws of Programming," *Communications of the ACM 30*, 8, 672–686, 770.

Horebeek, I. and J. Lewi (1989), *Algebraic specifications in software engineering An introduction*, Springer-Verlag, New York, N.Y.

Jackson, M. (1994), "Problems, methods and specialisation," *Software Engineering Journal* , 249–255.

Jackson, M. (1995), *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, ISBN 0-201-87712-0; xiv + 228 pages.

Jackson, M. (1997), "The meaning of requirements," *Annals of Software Engineering 3*, 5–21.

Janowski, T. (1996), "Domain Analysis for Manufacturing: Formalization of the Market," Research Report 63, UNU/IIST, P.O.Box 3058, Macau.

Janowski, T. and C. Acebedo (1996), "Virtual Enterprise: On Refinement Towards an ODP Architecture," Research Report 69, UNU/IIST, P.O.Box 3058, Macau.

Janowski, T. and R. V. Atienza (1997), "A Formal Model For Competing Enterprises, Applied to Marketing Decision-Making," Research Report 92, UNU/IIST, P.O.Box 3058, Macau.

Jensen, K. (1985), *Coloured Petri Nets*, volume 1–2–3 of *EATCS Monographs in Theoretical Com-*

puter Science, Springer–Verlag, Heidelberg.

Jensen, K. and N. Wirth (1976), *Pascal User Manuala and Report*, volume 18 of *Lecture Notes in Computer Science*, Springer-Verlag.

Jones, C. (1980), *Software Development: A Rigorous Approach*, Prentice-Hall.

Jones, C. (1986), *Systematic Software Development Using VDM*, Prentice-Hall, Superceded by [Jones 1990].

Jones, C. (1990), *Systematic Software Development using VDM*, Second Edition, Prentice Hall International.

Jones, N. D., C. Gomard, and P. Sestoft (1993), *Partial Evaluation and Automatic Program Generation*, C.A.R.Hoare Series in Computer Science, Prentice Hall International.

JuAn, W. and L. XiaoShan (1995a), "A Duration Calculus Approach to Specifying the Steam-boiler Problem," Technical Report 38, UNU/IIST, P.O.Box 3058, Macau.

JuAn, W. and L. XiaoShan (1995b), "Specifying Optimal Design of the Steam-boiler System," Technical Report 39, UNU/IIST, P.O.Box 3058, Macau.

Landin, P. (1964), "The Mechanical Evaluation of Expressions," *Computer Journal 6*, 4, 308–320.

Landin, P. (1965), "A Correspondence Between ALGOL 60 and Church's Lambda-Notation, (in 2 parts)," *Communications of the ACM 8*, 2-3, 89–101 and 158–165.

Landin, P. (1966a), "A Formal Description of ALGOL 60," In *[Steel 1966]*, pp. 266–294.

Landin, P. (1966b), "A Lambda Calculus Approach," In *Advances in Programming and Non-Numeric Computations*, L. Fox, Ed., Pergamon Press, pp. 97–141.

Larsen, P. G., Ed. (1993), *Formal Methods*, volume LNCS ??? of *Formal Methods Europe Symposium, Odense, Denmark*, Heidelberg - Berlin, Germany, Springer–Verlag.

Larsen, P. G., B. S. Hansen, H. B. N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, *et al.* (1996), "Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language," .

Liskov, B. and S. Zilles (1974), "Programming with Abstract Data Types," *'Very High Level Languages', SIGPLAN 9*, 4, 59–59.

Lucas, P. (1972), "On the Semantics of Programming Languages and Software Devices," In *Formal Semantics of Programming Languages*, Rustin, Ed., Prentice-Hall.

McCarthy, J. (1960), "Recursive Functions of Symbolic Expressions and their Computation by Machines, Part I," *Communications of the ACM 3*, 4, 184–195.

McCarthy, J. (1962), "Towards a Mathematical Science of Computation," In *IFIP World Congress*

Proceedings, C. Popplewell, Ed., pp. 21–28.

McCarthy, J. (1963), "A Basis for a Mathematical Theory of Computation," In *Computer Programming and Formal Systems*, North-Holland Publ.Co., Amsterdam.

McCarthy, J. and et al. (1962), *LISP 1.5, Programmer's Manual*, MIT Press, Cambridge, Mass.

McCarthy, J. and J. Painter (1966), "Correctness of a Compiler for Arithmetic Expressions," In *[Schwartz 1967]*, pp. 33–41, Dept. of Computer Science, Stanford University, California, USA.

Middelburg, C. (1988), "The VIP VDM specification language," In *VDM '88 VDM – The Way Ahead*, LNCS 328, Springer-Verlag, pp. 187–201.

Middelburg, K. and G. R. de Lavalette (1991), "LPF and MPL_ω – A Logical Comparison of VDM SL and COLD-K," In *VDM '91: Formal Software Development Methods*, VDM-Europe, Springer-Verlag, pp. 279–308.

Milne, R. and C. Strachey (1976), *A Theory of Programming Language Semantics*, Chapman and Hall, London, Halsted Press/John Wiley, New York.

Milner, R. (1980), *Calculus of Communication Systems*, volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag.

Milner, R. (1989), *Communication and Concurrency*, C.A.R. Hoare Series in Computer Science, Prentice Hall.

Milner, R., M. Tofte, and R. Harper (1990), *The Definition of Standard ML*, MIT Press, Cambridge, Mass. and London, England.

Mosses, P. (1997), "COFI: The Common Framework Initiative for Algebraic Specification and Development," In *TAPSOFT'97*, M. Bidoit and M. Dauchet, Eds., volume 1212 of *LNCS*, Springer–Verlag.

Naftalin, M., T. Denvir, and M. Bertran, Eds. (1994), *FME'94: Industrial Benefit of Formal Methods*, Formal Methods Europe Symposium, Barcelona, Spain, Heidelberg - Berlin, Germany, Springer–Verlag.

Nakagawa, A. T., T. Sawada, and K. Futatsugi (????), *CafeOBJ manual (for system version 1.3)*, 142 pages.

Nakagawa, K. F. A. (1997), "An Overview of CAFE Specification Environment – An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Networks," In *ICFEM'97: International Conference on Formal Engineering Methods*, IEEE Computer Society, IEEE CS Press.

Nordström, B., K. Petersson, and J. M. Smith (1990), *Programming in Martin-Löf's Type The-*

ory An Introduction, volume 7 of *International Series of Monographs on Computer Science*, Clarendon Press, Oxford University Press, Oxford, England,.

Oest, O. (1986), "VDM From Research to Practice," In *Information Processing '86*, H.-J. Kugler, Ed., IFIP World Congress Proceedings, North-Holland Publ.Co., Amsterdam, pp. 527–533.

on Language Design, C. T. G. (1997), "CASL — The Common Algebraic Specification Language Summary," Available at http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/.

on Semantics, C. T. G. (1997), "CASL — The CoFI Algebraic Specification Language (version 0.97) Semantics," Available at http://www.brics.dk/Projects/CoFI/Notes/S-4/.

Prehn, S. and W. Toetenel, Eds. (1991), *VDM ???*, Fourth International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October, 1991, Springer-Verlag, Lecture Notes in Computer Science, Vol. 551.

Qiwen, X. and H. Weidong (1995), "Hierarchical Design of a Chemical Concentration Control System," Research Report 41, UNU/IIST, P.O.Box 3058, Macau.

Qiwen, X. and Y. Zengyu (1996), "Derivation of Control Programs: a Heating System," Research Report 73, UNU/IIST, P.O.Box 3058, Macau.

Ravn, A., H. Rischel, and K. Hansen (1993), "Specifying and Verifying Requirements of Real-Time Systems," *IEEE Trans. Softw. Eng. 19*, 1, 41–55.

Reiser, M. (1991), *The* OBERON *System, User Guide and Programmer's Manual*, ACM Press, Addison-Wesley Publishing Company.

Reisig, W. (1998), *Theory and Practice of Petri Nets*, Springer–Verlag, Berlin Heidelberg.

Roscoe, A. (1997), *Theory and Practice of Concurrency*, Prentice–Hall.

Schmidt, U. and R. Völler (1985), "The Development of a Machine Independent Multi Language Compiler System Applying the Vienna Development Method," In *Proc. IFIP Working Conference on Software Specification Methodologies*, North-Holland Publ.Co., Amsterdam.

Schwartz, J. (1967), *Mathematical Aspects of Computer Science, Proc. of Symp. in Appl. Math.*, American Mathematical Society, Rhode Island, USA.

Schwartz, J. (1973), "The SETL Language and Examples of its Use," Technical report, Courant Institute of Mathematics, New York University.

Scott, D. (1970a), "The Lattice of Flow Diagrams," In *[Engeler 1971]*, pp. 311–366.

Scott, D. (1970b), "Outline of a Mathematical Theory of Computation," In *Proc. 4th Ann. Princeton Conf. on Inf. Sci. and Sys.*, p. 169.

Scott, D. (1972a), "Continuous Lattices," In *Toposes, Algebraic Geometry and Logic*, F. Lawvere, Ed., Springer-Verlag, Lecture Notes in Mathematics, Vol. 274, pp. 97–136.

Scott, D. (1972b), "Lattice Theory, Data Types and Semantics," In *Symp. Formal Semantics*, R. Rustin, Ed., Prentice-Hall, pp. 67–106.

Scott, D. (1972c), "Mathematical Concepts in Programming Language Semantics," In *Proc. AFIPS, Spring Joint Computer Conference, 40*, pp. 225–234.

Scott, D. (1973), "Lattice-Theoretic Models for Various Type Free Calculi," In *Proc. 4th Int'l. Congr. for Logic Methodology and the Philosophy of Science,* Bucharest, North-Holland Publ.Co., Amsterdam, pp. 157–187.

Scott, D. (1975), "λ-Calculus and Computer Science Theory," In *Lecture Notes in Computer Science, Vol. 37*, C. Böhm, Ed., Symp. Proc., Springer-Verlag.

Scott, D. (1976), "Data Types as Lattices," *SIAM Journal on Computer Science 5*, 3, 522–587.

Scott, D. (1981), "Lectures on a Mathematical Theory of Computation," Techn. Monograph 19, Programming Research Group.

Scott, D. (1982a), "Domains for Denotational Semantics," In *International Colloquium on Automata, Languages and Programming, European Association for Theoretical Computer Science*, Vol. 140 of Lecture Notes in Computer Science, Springer-Verlag, pp. 577–613.

Scott, D. (1982b), "Some Ordered Sets in Computer Science," In *Ordered Sets*, I. Rival, Ed., Reidel Publ., pp. 677–718.

Scott, D. and C. Strachey (1971), "Towards a Mathematical Semantics for Computer Languages," In *Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia*, pp. 19–46.

Shekaran, C., D. Garlan, and et al. (1994), "The role of software architecture in requirements engineering," In *First International Conference on Requirements Engineering (Cat. No.94TH0613-0); Colorado Springs, CO, USA*, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, pp. 239–245.

Skakkebæk, J., A. Ravn, H. Rischel, and Z. Chaochen (1992), "Specification of Embedded, Real-Time Systems," In *Proceedings of 1992 Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, pp. 116–121.

Skakkebæk, J. U. (M.Sc. Thesis), "Development of Provably Correct Systems," Technical report, Dept. of Computer Science, Technical University of Denmark.

Steel, T. (1966), *Formal Language Description Languages,* IFIP TC-2 Work.Conf., Baden, North-Holland Publ.Co., Amsterdam.

Strachey, C. (1966), "Towards a Formal Semantics," In *[Steel 1966]*, pp. 198–220.

Strachey, C. (1973), "The Varieties of Programming Languages," Techn. Monograph 10, Programming Research Group.

Strachey, C. (1974), "Continuations: A Mathematical Semantics which can deal with Full Jumps," Techn. monograph, Programming Research Group.

Strachey, C. and D. Scott (1970), "Mathematical Semantics for Two Simple Languages," Technical report, Princeton Univ.

U.Schmidt and R.Völler (1987), "Experience with VDM in Norsk Data," In *[Bjørner* et al. *1987a]*, Springer-Verlag, pp. 49–62.

van Leeuwen, J. (1990), *Handbook of Theoretical Computer Science: vol.A: Algorithms and Complexity, vol.B: Formal Models and Semantics*, North-Holland Publ.Co., Amsterdam.

Widjaja, B. H., H. Weidong, C. Zongji, and Z. Chaochen (1994), "A Cooperative Design for Hybrid Systems," Technical Report 36, UNU/IIST, P.O.Box 3058, Macau, Presented at and published in *Proceedings from the Second European Workshop on Real-time and Hybrid Systems*, Grenoble, France, June 1995.

Wirth, N. (1963), "A Generalization of ALGOL," *Communications of the ACM 6*, 547–554.

Wirth, N. (1971a), "Program Development by Stepwise Refinement," *Communications of the ACM 14*, 4, 221–227.

Wirth, N. (1971b), "The Programming Language PASCAL," *Acta Informatica 1*, 1, 35–63.

Wirth, N. (1973), *Systematic Programming*, Prentice-Hall.

Wirth, N. (1976), *Algorithms + Data Structures = Programs*, Prentice-Hall.

Wirth, N. (1982), *Programming in Modula-2*, Springer-Verlag, Heidelberg, Germany.

Wirth, N. (1988a), "From Modula to Oberon," *Software — Practice and Experience 18*, 661–670.

Wirth, N. (1988b), "The Programming Language Oberon," *Software — Practice and Experience 18*, 671–690.

Wirth, N. and J. Gutknecht (1989), "The Oberon System," *Software — Practice and Experience 19*, 9, 857–893.

Wirth, N. and J. Gutknecht (1992), *The Oberon Project*, ACM Press, Addison-Wesley Publishing Company.

Wirth, N. and C. Hoare (1966), "A Contribution to the Development of ALGOL," *Communications of the ACM 9*, 6, 413–432.

Wirth, N. and H. Weber (1966), "EULER: A Generalization of ALGOL, and its Formal Definition," *Communications of the ACM 9*, 1-2, 13–23, 89–99.

Woodcock, J. C. and M. Loomes (1988), *Software Engineering Mathematics: Formal Methods Demystified*, Pitman Publishing Ltd., London, UK.

Zave, P. and M. Jackson (1997a), "Four dark corners of requirements engineering," *ACM Trans-*

actions on Software Engineering and Methodology 6, 1, 1–30.

Zave, P. and M. Jackson (1997b), "Requirements for telecommunications services: an attack on complexity," In *Proceedings of the Third IEEE International Symposium on Requirements Engineering (Cat. No.97TB100086)*, IEEE Comput. Soc. Press, pp. 106–117.

Zhiming, L., A. Ravn, E. S. rensen, and Z. Chaochen (1993), "A Probabilistic Duration Calculus," In *Responsive Computer Systems*, H. Kopetz and Y. Kakuda, Eds., volume 7 of *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag Wien New York, pp. 30–52.

- Short Running Title: Pinnacles of Software Engineering: 25 Years of Formal Method
- Contact Author: Dines Bjørner
- Address: Department of IT, Technical University of Denmark, Bldg. 344, DK-2800 Lyngby, Denmark.
- Phone: +45-45.25.37.20
- Fax: +45-45.88.45.30
- E–Mail: db@it.dtu.dk

List of Figures