Domain Engineering:* A "Radical Innovation" for Software and Systems Engineering ? A Biased Account**

Dines Bjørner

Computer Science and Engineering (CSE),
Informatics and Mathematical Modelling (IMM)
Building 322, Richard Petersens Plads
Technical University of Denmark (DTU)
DK–2800 Kgs.Lyngby, Denmark
E–Mail: db@imm.dtu.dk

Abstract. Some facts: Before software and computing systems can be developed, their requirements must be reasonably well understood. Before requirements can be finalised the application domain, as it is, must be fairly well understood.

Some opinions: In today's software and computing systems development very little, if anything is done, we claim, to establish fair understandings of the domain. It simply does not suffice, we further claim, to record assumptions about the domain when recording requirements. Far more radical theories of application domains must be at hand before requirements development is even attempted.

In this presentation we advocate a strong rôle for domain engineering. We argue that domain descriptions are far more stable than are requirements prescriptions for support of one or another set of domain activities. We further argue, that once, given extensive domain descriptions, it is comparatively faster to establish trustworthy and stable requirements than it is today. We finally argue that once we have a sufficient (varietal) collection of domain specific, ie. related, albeit distinct, requirements, we can develop far more reusable software components than using current approaches.

* This paper was first drafted in connection with a US DoD Workshop held at Venezia, Italy, October 7–1, 2002. Sections of the present paper relates to the Call for that Workshop.

** The research behind and the presentation of this paper is partially, respectively fully funded under the 5th EU/IST Framework Programme (http://www.cordis.lu/fp5/home.html) of the European Commission, Contract Reference IST-2001-33123: CoLogNET: Network of Excellence in Computational Logic: http://www.eurice.de/colognet/

Thus, in this contribution we shall reason, at a meta-level, about major phases of software engineering: Domain engineering, requirements engineering, and software design.
We shall suggest a number of domain and requirements engineering as well as software design concerns, stages and steps, notably: Domain facets, including domain intrinsics, support technologies, management & organisation, rules & regulations, as well as human behaviour. Requirements: Domain requirements, interface requirements, and machine requirements. Specifically: Domain requirements projection, determination, extension, and initialisation.
We shall then proceed to "lift" our methodological concerns to encompass the more general ones of abstraction and modelling; of informal as well as formal description; of the more general issues of documentation: Informative, descriptive/prescriptive, and analytical; and hence of the importance of semiotics: Pragmatics, semantics, and syntax.
The paper concludes with a proposal for a 'Grand Challenge' for computing science [62, 89, 19].

Paragraphs marked $\star$ in the text are considered somewhat novel.

A Laudatio: To Zohar Manna

It is indeed a great honour to help celebrate Zohar Manna. I first heard about Zohar when, as an IBM employee, working for Gene Amdahl at the IBM Advanced Computing Systems Laboratory at 2800 Sand Hill Road in Menlo Park, I was able, with Gene's kind permission, to sneak over and be enlightened by listening to Zohar's lectures in, I believe it was Polya Hall. Later, at IBM Research, I also followed Zohar's lectures via closed circuit television to the IBM site at Monterey & Cottle Roads in San Jose. At that time I motored up to Stanford to listen to PhD lectures by Zohar's students: Jean Marie Cadiou, Ashok Chandra and Jean Vuillemin. And there were others. 12 years later I was able to have two of my own PhD students, now Drs Bo Stig Hansen and Michael Reichardt Hansen, for a year with Ted Codd at IBM Research, still in San Jose, follow Zohar's lectures at Stanford. He was kind enough to tell me that they were indeed very good students. Yes they damn well be if one sends them to Zohar — if only for lectures. But the crowning moment, as a father, was when our own son, Nikolaj, became a PhD student of Zohar's. Thanks, Zohar, for showing — in a double entendre on the word 'showing' — us that the subject with which we are working is not only fascinating and great, but for lecturing to us in such vivid, beautiful and enticing ways. Thanks also, as the father of Nikolaj, to have ensured that a father can become positively known for the scientific deeds of his son. Thanks for fascinating encounters around the world: Never let one's deep commitment to science overshadow the joy of life. Thanks for helping advicing

and steering "my" UNU/IIST[1] to even greater achievements — as you do through Your never in–active membership of the Board of UNU/IIST. Thanks finally for a wonderful trip to Ulaan Baator and, in Jeeps, for five days, sleeping outdoors, in sub–zero (ie., centigrade) degree weather, across the deserts of Western Mongolia in August 1995. What would we be without our wives: So a cheer and many thanks also goes to Nitza.

1 Introduction

1.1 Background

This paper has been written on the backgound of 30 years[2] of work developing, researching and using formal techniques, notably VDM and RAISE. It is also written on the background of teaching this for 27 years, personally graduating more than 120 MSc candidates most of whom now works in some seven to eight Danish industries where they habitually use formal techniques. All of these companies are either founded by these former students or are significantly staffed and managed by such candidates. The paper is also written on the background, 24 years ago, of the first deployment of formal techniques, from "light" (we then called it systematic) to rigorous, to the development of commercial compilers for CHILL [60] and Ada [41, 79] — with the Ada company, DDC Intl., still in existence 19 years after its formation. The paper is finally written on the background of UNU/IIST's[3] rather successful technology transfer activities, while a founding UN Director, and afterwards, to groups in around 30 developing countries [55].

1.2 Itemised Summary

Some facts:

- Before software and computing systems can be developed, their requirements must be reasonably well understood.
- Before requirements can be finalised the application domain, as it is, must be fairly well understood.

Some opinions:

- In today's software and computing systems development very little, if anything is done, we claim, to establish fair understandings of the domain.

[1] UNU/IIST is the United Nations University's International Institute for Software Technology, located in Macau SAR, China — but active all over the developing world.

[2] The author then joined, May 1973, The IBM Vienna Laboratory at Vienna, Austria, after having worked with Gene Amdahl, John W. Backus and E.F. Codd at IBM Adv.Comp.Sys.Devt., Menlo Park, respectively at IBM Research, San Jose, both in California.

[3] UNU/IIST: The United Nations' University's International Institute for Software Technology, Macau: `www.iist.unu.edu`.

– It simply does not suffice, we further claim, to record assumptions about the domain when recording requirements.
– Far more radical theories of application domains must be at hand before requirements development is even attempted.

In this presentation we advocate a strong rôle for domain engineering.

– We argue that domain descriptions are far more stable than are requirements prescriptions for support of one or another set of domain activities.
– We further argue, that once, given extensive domain descriptions, it is comparatively faster to establish trustworthy and stable requirements than it is today.
– We finally argue that once we have a sufficient (varietal) collection of domain specific, ie. related, albeit distinct, requirements, we can develop far more reusable software components than using current approaches.

Thus, in this contribution we shall reason, at a meta-level, about major phases of software engineering:

– Domain engineering,
– requirements engineering, and
– software design.

We shall suggest a number of domain and requirements engineering as well as software design concerns, stages and steps, notably:

– Domain facets, including
  • domain intrinsics,
  • support technologies,
  • management & organisation,
  • rules & regulations, as well as
  • human behaviour.
– Requirements:
  • Domain requirements,
  • interface requirements, and
  • machine requirements.
– Specifically:
  • Domain requirements projection,
  • determination,
  • extension, and
  • initialisation.

We shall then proceed to "lift" our methodological concerns to encompass the more general ones of

– abstraction and modelling;
– of informal as well as formal description;
– of the more general issues of documentation: Informative, descriptive/prescriptive, and analytical;
– and hence of the importance of semiotics: Pragmatics, semantics, and syntax.

It is our contention that many of these concerns are more proper concerns of researching and teaching software engineering than what is currently en vogue, and hence we shall plead for their study and for them as proper didactic foundations for the field of software engineering. We have omitted, in this paper, even an, however summary, treatment of *documentation* and *semiotics* in order to discuss, on a proper background, the issues of *"radical innovation" of systems and software engineering for the future:* We think that the *triptych of software engineering,* as covered in Section 2 (Pages 5–31), goes hand–in–hand with *documentation* and *semiotics.* Without a proper understanding of also the issues of the latter we will not be developing trustworthy, maintainable software systems.

In this paper we shall only very briefly mention the problem of requirements acquisition: At the end, respectively at the beginning of Section 2.2 and Section 2.3. The concerns of "classical" requirements acquisition are now the concerns both of domain and requirements acquisition. The possibilities of inconsistencies remain: Different stake–holders may have different views. In the domain they mean that a proper "business engineering" need be done. For requirements they entail the usual acts of requirements resolution.

2 A Triptych of Software Engineering

So in what does this new engineering consist ? The next many sections and subsections, etcetera, will outline facts as well as opinions.

2.1 Abstraction and Modelling in General

The ability to abstract is central, we believe, to good software development. But it is hard to learn. And few teach it. Also: Maybe not all can learn it. But those who can, suffice. Not all architects have to be such as Richard Meier, Paul Gehry, Jørn Utzon, etc. A few good "abstractionists", in a software company, can put a good many "lesser" souls to good and meaningful work.

2.2 Domain Engineering

There seems many ways of tackling the description — the modelling — of application domains: Of "entire" such domains as: A railway system, or all railway systems, a healthcare system, a financial service industry, logistics, the background domain for (ie., "before") e–commerce, etc. We will single out some for which we can offer principles, techniques and tools for their methodical description.

Domain Stake–holders & Perspectives :

As far ranging an enumeration need be made at the outset of every phase of software development: Domain, requirements and software deisgn, of the stake–holders "of that phase". A generic list for some enterprise software development could include: (i) The enterprise owners (share holders), (ii) the executive, (iii) line,

(iv) operations and (v) "floor" management, (vi) the "floor" workers (and (vii) their families), (viii) customers of the enterprise, suppliers of (ix) IT, (x) non–IT equipment and services ((xi) banking, (xii) utilities, (xiii) postal, (xiv) trucking, etc.) to the enterprise, (xv) regulators of the industry sector (if relevant), (xvi) politicians, and (xvii) the "public at large".

For each of the relevant ones of these domain knowledge has to be acquired [58, 57, 99, 56, 75, 77]. Inconsistencies in acquired information has to be resolved [64]. Conflicts have to be identified [91]. (It is not a task of domain engineering to resolve domain conflicts. Such is more a task of Business Process (Re–)Engineering.) Obstacles to acquisition has to be overcome [92, 94]. Etcetera. Finally the various perspectives have to be integrated [78].

In this paper we shall not get into this extremely important area. But "simply" assume that the domain knowledge is somehow represented in a form that allow us to identify domain attributes and domain facets.

We shall return, below, to a separate, albeit very brief discussion of Domain Acquisition.

Domain Facets :

How do we structure the description of a domain ? Well, here is one way of doing it. First describe the intrinsics, then go on to describe either the support technologies, or to describe the triplet of management & organisation, rules & regulations and human behaviour in that order. Or do it the other way around. Details follow.

Each of these five facets may need being described from the perspectives of each of the stake–holder classes.

⋆ *Intrinsics* :

By intrinsics we mean *''that which is common, in any description of a domain, to all the other facets"* (treated below).

But the intrinsics of domain depends on the stake–holder view. We take a small example, from railway systems, namely that of railway nets.

Simple Nets of Single Lines and Simple Stations: A railway net consists of two or more stations and one or more lines. Any line of a net connects exactly two distinct stations of that net.

```
scheme VSN =
  class
    type
      N, L, S
    value
      obs_Ls: N → L-set
      obs_Ss: N → S-set
      obs_Ss: L → S-set
    axiom
      ∀ n:N •
        card obs_Ss(n) ≥ 2 ∧ card obs_Ls(n) ≥ 1 ∧
```

```
        ∀ l:L • l ∈ obs_Ls(n) ⇒ obs_Ss(l) ⊆ obs_Ss(n) ∧ card obs_Ss(l) = 2
end
```

To express constraints on stations and lines: At most one per pair of stations, we extend VSN:

```
scheme SN_0 =
  extend VSN with
  class
      ∀ n:N •
        ∀ l.l′:L • {l,l′} ⊆ ∈ obs_Ls(n) ∧ l≠l ⇒ card(obs_Ss(l) ∩ obs_Ss(l′)) ≤ 1
  end
```

This is the simplest stake–holder prespective — as seen by prospective train passenger or freight forwarding stake–holders. Based on the above we can define routes, connections, etc.

Nets of Multiple Lines and Multiple Platform Stations: As above, but now extended: Each station possesses one or more platforms, and a pair of stations may be connected by more than one line.

```
scheme SN_1 =
  extend VSN with
  class
    type
      P
    value
      obs_Ps: S → P-set
    axiom
      ∀ s:S • card obs_Ps(s) ≥ 1
  end
```

So it was simpler to express the more sophisticated nets, SN_1, than the single line nets, SN_0.

This is the stake–holder perspective that actual passengers need to have in order to enter correct platform for catching a train or for changing trains, or for friends and porters to come greet, respectively collectr your luggage.

Nets, Lines and Station Units and Connectors: As above, but now nets, lines and stations consists of — are further refined into — units, and units have connectors. Linear units have two connectors, switches have three, crossover have four, etc. Units of lines are units of the net, units of stations are units of the net, distinct lines have no units in common, distinct stations have no units in common, units of platforms of stations are units of the net and of their station.

```
scheme NET
  extend SN_1 with
    class
      type
```

```
    U, C,
    W = C×C
    Σ = W-set
    Ω = Σ-set
  value
    obs_Us: (N|L|S|P) → U-set
    obs_Cs: U → C-set
    obs_Ws: U → W-set
    obs_Σ: U → Σ
    obs_Ω: U → Ω
  axiom
    ∀ n:N,l:L,s:S,p:S,u:U,c,c′:C −
      l ∈ obs_Ls(n) ⇒ obs_Us(l) ⊆ obs_Us(n) ∧
      s ∈ obs_Ss(n) ⇒ obs_Us(s) ⊆ obs_Us(n) ∧
      p ∈ obs_Ps(s) ⇒ obs_Us(p) ⊆ obs_Us(s) ∧
      σ ∈ obs_Ω(u) ⇒ ∀ (c,c′):W • (c,c′) ∈ σ ⇒ {c,c′}⊆obs_Cs(u) ∧
      card { u:U • u ∈ obs_Us(n) ∧ c ∈ obs_Cs(u) } ≤ 2
      ∧ ...
end
```

The last axiom expresses the syntax, ie., the topological layout of a net: Any connector of a unit of net is the connector of at most two units of the net.

This is part of the stake–holder perspective that train planners, despatchers and engine men need to have in order to plan, monitor and control secure traffic.

⋆ *Support Technology Behaviours* :

By support technology we understand a hard technology, ie., something other than humans, which "implement" some of the phenomena modelled by an intrinsics.

Example 1: Railway Switches: A switch, as intrinsically abstracted above, had three connectors,c, c', c'',and typically the following ways, $w : W$, through the switch: (c, c'), (c, c''), (c', c),and (c'', c).An intrinsic state space, ω, of a common switch would allow for the following states: { {}, $\{(c, c')\}$, $\{(c, c'')\}$, $\{(c', c)\}$, $\{(c'', c)\}$, $\{(c, c'), (c', c)\}$, $\{(c, c''), (c'', c)\}$, $\{(c, c'), (c', c), (c'', c)\}$,and $\{(c, c''), (c'', c), (c', c)\}\}$.

Switches are then implemented, technologically, by humans throwing the switch (and we shall discuss modelling that below), or by mechanical levers and pullers connected, over a short distance by steel wires, or by electro–mechanic means, or by electronic & electro–mechanic means, the latter typically controlling groups of switches as in interlocked switching.

For each of these technologies we must model their real behaviour: Their time response to actions, their failure to operate properly, etc. For that we can use, for example one of the duration calculi [40]. Etcetera.

Example 2: Railway Traffic: Trains travel the net and occupy sequences of units of the net. At any one time several trains occupy such sequences, and over time trains normally move monotonically.

```
scheme TRAFFIC_0 =
  extend NET with
    class
      type
        T, Train
        Route = U*
        TF_0 = T → N_POS
        N_POS :: s_n:N  s_ps:(Train m→ Route)
      axiom
        ...
    end
```

But how do we observe traffic, TF ? Possibly by means of mechanical or optical or other forms of sensors. The observed traffic is discretised. The sampling frequency being some function of those of the sensors.

```
scheme TRAFFIC_1
  extend TRAFFIC_0 with
    class
      type
        TF_1 = T m→ N_POS
      axiom
        ...
    end
```

How do the two relate: The real, actual, traffic, TRAFFIC_0, and the observed traffic TRAFFIC_!. We say that the sensor technology, st, relates the two:

```
scheme TRAFFIC_2
  extend TRAFFIC_1 with
    class
      type
        ST = TF_0 → TF_1
      value
        st:ST
        close: Route × Route → N → Bool
        ok_st: ST → Bool
        ok_st(st) ≡
          ∀ tf_0:TF_0,tf_1:TF_1 • tf_1 = st(tf_0) ⇒
            ∀ t:T • t ∈ dom tf_1 ⇒
              t ∈ 𝒟 tf_0 ∧ dom s_ps(tf_0(t)) = s_ps(dom tf_1(t)) ∧
              ∀ tn:Tn • tn ∈ s_ps(dom tf_1(t)) ⇒
                close((s_ps(tf_0(t)))(tn),(s_ps(tf_1(t)))(tn))(s_n(tf_0))
    end
```

The ok_st predicate expresses a "goodness" criterion for a sampling technology, st.

The closeness predicate expresses whether two trains are close, ie., "occupy" approximately the same or identical routes of a net.

The above is the perspective of the supplier and control engineer stake–holders who are to provide and develop trustworthy support technology resources that permit safe and secure monitoring, respectivelt control of traffic.

So "whole" theories can be built up around each supporting technology.

⋆ *Management & Organisation* :

By management & organisation we understand the relations between various stake–holders, notably employees of the enterprises at the focus of the chosen application domain: Their protocols of interaction, including models of their "agent" and "speech act" behaviours [80].

Management usually formulate strategies and tactics for the desirable behaviour of enterprise, ie., the domain, and they also formulate the rules & regulations for suitable behaviours of humans and support technologies of that domain. So when support technologies and humans fail to perform as stated management must somehow be informed. Thus management acts "top down" by propagating strategies , tactics, rules and regulations "down" the "pyramid", and management "back stops" problems, at lower levels, by taking over the responsibility for handling unforeseen (ie., un–ruled, un–regulated) behaviours.

We can model this by suitable protocol specifications expressed both in terms of (eg. CSP–like) process behaviours and in terms of suitable agent behaviours, ie., using appropriate modal logics and suitable speech act illocutions, locutions and perlocutions.

⋆ *Rules & Regulations* :

By rules we understand that which guides the work of enterprise stake-holders and their support technologies, as well as their interaction and the interaction with non-enterprise stake-holders

By regulations we understand that which stipulate what is to happen if a rule can be detected not to have been followed when such was deemed necessary.

Some examples are:

– Train Arrivals and Departures at Stations:
  - Rule: In China arrival and departure of trains at, respectively from railway stations are subject to the following rule: *"In any three minute interval at most one train may either arrive or depart."*
  - Regulation: Disciplinary procedures (not detailed here).
– Train Traffic along Lines:
  - Rule: In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving — obviously in the same direction — along the line, then: *"There must be at least one free sector (ie., without a train) between any two such trains."*
  - Regulation: Disciplinary procedures (not detailed here).

There are usually three kinds of syntax involved wrt. (ie. when expressing) rules & regulations (resp. when invoking actions that are subject to rules & regulations): Rules, Regus: The syntaxes describing rules, respectively regulations; and Stimulus: The syntax of [always current] domain external action stimuli. A rule: RUL = Stimulus → Θ × Θ → **Bool**, denotationally, is a predicate over domain stimuli, and current and next domain configurations Θ). A regulation: REG = Stimulus → Θ → Θ, denotationally, is a domain configuration changing function over domain stimuli, and current domain configurations.

type
  Rules, Regus, Stimulus
  Ruls_and_Regs = Rules × Regus
  Θ
  RUL = Stimulus → Θ × Θ → **Bool**,
  REG = Stimulus → Θ → Θ
value
  interpret: Rules → Θ → RUL-**set**,
  interpret: Regus → Θ → REG

  valid: Stimulus × RUL-**set** → Θ × Θ → **Bool**
  valid(sti,ruls)(θ,θ') ≡
    ∀ rul:RUL • rul ∈ ruls ⇒ rul(sti)(θ,θ')
axiom
  ∀ (sy_rls,sy_rgs):Ruls_and_Regs,θ:Θ•
    let se_rls = interpret(sy_rls)(θ),
        se_rgs = interpret(sy_rgs)(θ) **in**
    ∃ sti:Stimulus,θ',θ'',θ''':Θ •
      ∼valid(sti,se_rls)(θ',θ'')
        ⇒ reg(sti)(θ')=θ''' ∧ valid(sti,se_rls)(θ',θ''')
    end

⋆ *Human Behaviour* :

Some people try their best to perform actions according to expectations set by their colleagues, customers, etc. And they usually succeed in doing so. They are therefore judged reliable and trustworthy, good, punctual professionals of their domain. Some people set lower standards for their professional conduct: Are sometimes or often sloppy, make mistakes, unknowingly or even knowingly. And yet other people are outright delinquent in the despatch of their work: Could'nt care less about living up to expectations of their colleagues and customers. Finally some people are explicitly criminal in the conduct of what they do: Deliberately "do the opposite" of what is expected, circumvent rules & regulations, etc. And we must abstract and model, in any given situation where a human interferes in the "workings" of a domain action, any one of the above possible behaviours

By human behaviour we thus understand the way in which domain stakeholders despatch their actions and interactions wrt. an enterprise: professionally, sloppily, delinquently, yes even criminally.

Commensurate with the above formalisation, humans interpret rules & regulations differently, and not always "consistently" — in the sense of repeatedly applying the same interpretations.

```
type
  Action = Θ ~→ Θ-infset
value
  hum_int: Rules → Θ → (RUL-set)-infset

  hum_beha: Stimulus × Rules × Action → Θ ~→ Θ
  hum_beha(sti,syn_rls,α)(θ) as θ′
    post
      θ′ ∈ α(θ) ∧
      let sem_rls:RUL-set • sem_rls ∈ hum_int(srr)(θ) in
        ∀ rule:RUL • rule ∈ sem_rls ⇒ rule(sti)(θ,θ′)
      end
```

The above is, necessarily, sketchy: There is a possibly infinite variety of ways of interpreting some rule[s]. A human, in carrying out an action, interprets applicable rules and choose a set which that person believes suits some (professional, sloppy, delinquent or criminal) intent. "Suits" means that it satisfies the intent, ie., yields **true** on the pre/post configuration pair, when the action is performed — whether as intended by the ones who issued the rules & regulations or not. Such is reality !

Relations to Knowledge Engineering (KE) :

Domain engineering is the engineering of domain descriptions. Domain descriptions "encode" domain knowledge. But that does'nt make domain engineering equal to, or the same as KE. In our view there is a fundamental, perhaps just a historic difference: Knowledge representation researchers and engineers all seem to try express knowledge almost exclusively in a property oriented, read: Mathematical logic style and, and this is also important, *notably one whose acquisition can be partially, and whose formalisation can be mechanised.* In our quest for domain models we accept that our models are not mechanisable. They are formal, but not necessarily immediately "executable". Judge for yourself: You would not qualify the models given above as knowledge models in the traditional, current sense of knowledge engineering. KE seems to have sprung from classical AI research, where domain engineering seems to have sprung from classical denotational engineering. It seems that KE is always searching for new formal KE languages. In domain engineering we have, at the moment, our hands full with just trying to express, within existing formal notations. KE seems not to have actually expressed large scale models of domain in the sense domain engineering has.

Despite the seeming "assuredness" of the above, see [58, 75].

Domain Acquisition :

We refer to an upcoming subsection on 'Requirements Acquisition', barely half a page onwards. We do so — as was also mentioned in the last paragraph of Section 1.2 — since there are many similarities between domain and requirements acquisition. We seriously think that the issues of domain attributes and domain facets, as outlined above, can help guide the domain acquisition process. This also applies to the more general abstraction and modelling techniques, and especially when all this is done in a context of formal specification: That domain acquisition, and, as we shall shortly see, also requirements acquisition, will be "helpfully guided."

2.3 Requirements Engineering

Delineation of "What is Requirements ?" :

From [90] we quote: *"Requirements engineering must address the contextual goals why a software is needed, the functionalities the software has to accomplish to achieve those goals, and the constraints restricting how the software accomplishing those functions is to be designed and implemented. Such goals, functions and constraints have to be mapped to precise specifications of software behaviour; their evolution over time and across software families has to be coped with as well [100]."*

We shall, in this paper, not cover the pragmatics of why software is needed, and we shall, not only in this paper, exclude "the mapping to precise software specifications" as we believe this is a task of the first stages of software design — as will be illustrated in this paper.

Requirements Acquisition :

The process of requirements acquisition will also not be dealt with here. We assume that proper such techniques, if available, will be used. For example [59, 44, 76, 45, 57, 99, 78, 56, 46, 1, 64, 92, 91, 93, 77, 94]. That is: We assume that somehow or other we have some, however roughly, but consistently expressed itemised set of requirements. We admit, readily, that to achieve this is a major feat. The domain requirements techniques soon to be outlined in this paper may help "parameterise" the referenced requirements acquisition techniques.

On the Avaliability of Domain Models :

It is a thesis of this paper that it makes only very little sense to embark on requirements engineering before one has a fair bit of understanding of the application domain. Granted that one may feel compelled to develop both "simultaneously", or that one ought expect that others have developed the domain descriptions (including formal theories) "long time beforehand." Yes, indeed, just as control engineers can rely on Newton's laws and more than three hundred

years of creating improved understanding of the domain of Newtonian physics: The "mechanical" world as we see it daily, so software engineers ought be able, sooner or later, to rely on elsewhere developed models of — usually man–made — application domains. Since that is not yet the situation we, software engineering cum computing science, shall have to make the first attempts at creating such domain–wide descriptions — hoping that eventually the domain specific professions will have reseachers with sufficient computing science education to hone and further develop such models.

Domain Requirements :

It is also a thesis of this paper that a major, perhaps the most important aspects of requirements be systematically developed on the basis of domain descriptions. This 'thesis' thus undercuts much of current requirements engineerings' paradigms, it seems.

By a domain requirements we shall understand those requirements (for a computing system) which are expressed solely by using terms of the application domain (in addition to ordinary language terms). Thus a domain requirements must not contain terms that designate the machine, the computing system, the hardware + software to be deviced.

How do we go about doing this ?

There seems to be two orthogonal approaches. In one we follow the domain facets outlined above. In the other we apply a number of "operators", to wit:

- projection, determination, extension, and initialisation,

to domain required facets. We treat the latter first:

Facet–Neutral Domain Requirements :

- Projections:
  Well, first we ask for which parts of the domain we, the client, wish computing support. Usually we must rely on our domain model to cover far more than just those parts. Hence we establish the first bits of domain requirements by *projecting* those parts of both the informal and the formal descriptions onto — ie., to become — the domain requirements.
- Determinations:
  Then we look at those projected parts: If they contain undesired looseness or non–determinism, or if parts, like types, are just sorts for which we now wish to state more — not implementation, but "contents" — details, then we remove such looseness, such non–determinacy, such sorts, etc. This we call *determination.*
- Extensions:
  Certain functionalities can be spoken of in the domain but to carry them out by humans have either been too dangerous, too tedious, uneconomical, or otherwise infeasible. With computing these functionalities may now be feasible. And, although they, in a sense "belongs" to the domain, we first

introduce them while creating the domain requirements. We call this *domain extension*.

– Initialisations:
In describing a domain, such as we for example described the "space" of all railway nets, we, for each specific net, designate the "space" of all its lines and stations and their units and connectors. If our requirements involved lines, stations and units, then sooner or later one has to initialise the computing system (database) to reflect all these many entities. Hence we need to establish requirements for how to *initialise* the computing system, how to maintain and update it, how to vet the input data, etc.

There may be other domain–to–requirements "conversion" steps. We shall, in this paper, only speak of these.

In doing the above we may iterate between the four (or more) domain–to–requirements "conversion" steps.

We now illustrate what may be going on here.

An Example Domain: We wish to illustrate the concepts of projection, determination, extension and initialisation of a domain requirements from a domain. We will therefore postulate a domain. We choose a very simple domain. That of a traffic time table, say flight time table. In the domain you could, in "ye olde days" hold such a time table in your hand, you could browse it, you could look up a special flight, you could tear pages out of it, etc. There is no end as to what you can do to such a time table. So we will just postulate a sort, TT, of time tables. Airline customers, in general only wish to inquire a time table (so we will here omit treatment of more or less "malicious" or destructive acts). But you could still count the number of digits "7" in the time table, and other such ridiculous things. So we postulate a broadest variety of inquiry functions that apply to time tables and yield values. Specifically designated airline staff may, however, in addition to what a client can do, update the time table, but, recalling human behaviours, all we can ascertain for sure is that update functions apply to time tables and yield two things: Another, replacement time table and a result such as: *"your update succeeded"*, or *"your update did not succeed"*, etc. In essence this is all we can say for sure about the domain of time table creations and uses.

```
scheme TI_TBL_0 =
  class
    type
      TT, VAL, RES
      QU = TT → VAL
      UP = TT → TT × RES
    value
      client_0: TT → VAL, client(tt) ≡ let q:QU in q(tt) end
      staff_0: TT → TT × RES, staff(tt) ≡ let u:UP in u(tt) end
      timtbl_0: TT → Unit
      timtbl(tt) ≡
```

```
        (let v = client_0(tt) in timtbl_0(tt) end)
        ||
        (let (tt′,r) = staff_0(tt) in timtbl_0(tt′) end)
  end
```

The system function is here seen as a never ending process, hence the type **Unit.** It internal non–deterministically alternates between "serving" the clients and the staff. Either of these two internal non–deterministically chooses from a possibly very large set of queries, respectively updates.

Projections: In this case we have defined such a simple, ie., small domain, so we decide to project all of it onto the domain requirements:

```
scheme TI_TBL_1 = TI_TBL_0
```

Determinations: Now we make more explicit a number of things: Time tables record, for each flight number, a journey: a sequence of two or more airport visits, each designated by a time of arrival, the airport name and a time of departure.

```
scheme TI_TBL_2 =
  extend TI_TBL_1 with
    class
      type
        Fn, T, An
        JR′ = (T × An × T)*
        JR = {| jr:JR′ • len jr ≥ 2 ∧ ... |}
        TT = Fn ~m~> JR
    end
```

where we omit (...) to express further wellformedness constraints on journies.

Then we determine the kinds of queries and updates that may take place:

```
scheme TI_TBL_3 =
  extend TI_TBL_2 with
    class
      type
        Query == mk_brow() | mk_jour(fn:Fn)
        Update == mk_inst(fn:Fn,jr:JR) | mk_delt(fn:Fn)
        VAL = TT
        RES == ok | not_ok
      value
        Mq: Query → QU
        Mq(qu) ≡
          case qu of
            mk_brow() →
```

```
                λtt:TT•tt,
            mk_jour(fn)
                → λtt:TT •
                    if fn ∈ dom tt then [ fn↦tt(fn) ] else [ ] end
        end

    Mu: Update → UP
    Mu(up) ≡
        case qu of
            mk_inst(fn,jr) →
                λtt:TT •
                if fn ∈ dom tt then (tt,not_ok) else (tt ∪ [ fn↦jr ],ok) end,
            mk_delt(fn) →
                λtt:TT •
                if fn ∈ dom tt then (tt\{fn},ok) else (tt,not_ok) end
        end
end
```

And finally we redefine the client and staff functions:

```
scheme TI_TBL_4 =
  extend TI_TBL_3 with
    class
      value
        client_2: TT → VAL,
        client_2(tt) ≡ let q:Query in (Mq(q))(tt) end
        staff_2: TT → TT × RES,
        staff_2(tt) ≡ let u:Update in (Mu(u))(tt) end
    end
```

The timtbl function remains "basically" unchanged !

```
scheme TI_TBL_5 =
  extend TI_TBL_4 with
    class
      value
        timtbl_2: TT → Unit
        timtbl_2(tt) ≡
            (let v = client_2(tt) in timtbl_2(tt) end)
            ⌈⌉
            (let (tt′,r) = staff_2(tt) in timtbl_2(tt′) end)
    end
```

Extensions: Suppose a client wishes, querying the time table, to find a connection betwen two airports with no more than n shift of aircrafts. For $n =$

$0, n = 1$ or $n = 2$ this may not be difficult to do *"in the domain":* A few 3m post it's a human can perhaps do it in some reasonable time for $n = 1$ or $n = 2$. But what about for $n = 5$. Exponential growth in possibilities makes this an infeasible query *"in the domain"*. But perhaps not using computers.

```
scheme TI_TBL_6 =
  extend TI_TBL_5 with
    class
      type
        Query == ... | mk_conn(fa:An,ta:An,n:Nat)
        VAL = TT | CNS
        CNS = (JR*)-set
      value
        Mq(q) ≡
          cases q of
            ...
            mk_conn(fa,ta,n) → λtt:TT • ...
          end
    end
```

where we leave it to the reader to define the "connections" function !

Initialisations: Initialisation here means: From a given input of flight journies to create an initial time table — as insert and deletes have already been defined. But, in their definition we skirted an issues which is paramount also in initialisation: Namely that of vetting the data. That is, checking that a journey flies non–cyclically between existing airports, that flight times are commensurate with flight distances and type of aircraft (jet, supersonic or turbo–prop), that at at all airports planes touch down and take off at most every n minutes, where n could be 2, but is otherwise an airport parameter. To check some of these things information about airports and air space is required.

```
scheme INI_TT =
  extend TI_TBL_2 with
    class
      type
        Init_inp = (Fn × JR)-set
        AP = An ~m~> Airport
        AS = (An × An) ~m~> AirCorridor-set
        Number, Length
      value
        obs_RunWays: Airport → Number
        obs_Distance: AirCorridor → Length
        ...
    end
```

We leave it to the imganination, skills and stamina of the reader to complete the details ! Our points has been made: Initialisation suddenly uncovers a need for enlarging the domain descriptions, and *"there is much more to initialisation than meets the eye."*[4]

Facet–Oriented Domain Requirements :

We may be able to make a distinction between "intended" and un–intended inconsistencies and "intended" and unintended conflicts. The "intended" ones are due to inherent properties of the domain. The un–intended ones are due to misinterpretations by the domain recorders or, are "real enough," but can be resolved through negotiation between stake–holders — thus entailing aspects of business process re–engineering — before requirements capture has started.

We thus assume, for brevity of exposition, that un–intended inconsistencies and un–intended conflicts have been thus resolved, and that otherwise "separately" expressed perspectives have been properly integrated (ie. ameliorated).

A major aspect of domain requirements is that of establishing contractual relationships between the human or support technology 'agents' in the environment of the "software, cum system–to–be", and the software 'agents'. As a result of a properly completed and integrated domain modelling of support technologies, management & organisation, rules & regulations, and human behaviour, we have thus identified domain inherent inconsistencies and conflicts. They appear as a form of non–determinism. These forms of non–determinism typically need either be made deterministic, as in domain requirements determination, or be made part of a contract assumed to be enforced by the environment: Namely a contract that says: *"The environment will promise (cum guarantee) that the inconsistency or the conflict will not 'show up' !"*

These contractual relationships express assumptions about the interaction behaviour — to be further explored as part of the next topic: 'Interface Requirements'. If the environment side of the combined system of the "software, cum system–to–be" does not honour these contractual relationships, then the "software, cum system–to–be" cannot be guaranteed to act as intended !

We thus relegate treatment of some facet–oriented domain requirements to the requirements capture and modelling stage of interface requirements.

⋆ *Towards a Calculus of Domain Requirements* :

We have sketched a "calculus" for deriving domain requirements. So far we have identified four operations in this "calculus": Projection, determination, extension and initialisation. In each derivation step the operation takes two arguments. One argument is the domain requirements developed so far. The other argument is the concerns of that step of derivation: What is, and what is not to be projected, what is and what is not to be determined, what is and what is not to be extended, respectively what is and what is not to be initialised, etc.

[4] Reasonable C code for the input of directed graphs is usually twice the "size" of similarly reasonable C code for their topological sorting !

We have still to further develop: Identify possibly additional domain requirements derivation operators, and to research and provide further principles and detailed techniques also for already identified derivation operations.

It seems that the sequence of applying these derivators is as suggested above, but is that "for sure ?".

Interface Requirements :

By an interface requirements we shall understand those requirements (for a computing system) which concern very explicitly the "things" 'shared' between the domain and the machine: In the domain we say that these "things" are the observable phenomena: the information, the functions, and/or the events of, or in, the domain, In the machine we say that they are the data, the actions, and/or the interrupts and/or the occurrence of inputs and outputs of the machine. By 'sharing' we mean that the latter shall model, or be harmonised with, the former. There are other interface aspects — such as "translates" into "bulk" input/output, etc.

But we shall thus illustrate just the first two aspects of 'sharing'.

External vs. Internal 'Agent' Behaviours :

The objectives of this step of requirements development is the harmonisation of external and internal 'agent' behaviours.

One the side of the environment there are the 'agents', say the human users, of the "software–to–be". On the side of the "software–to–be" there is, say, the software 'agents' (ie. the processes) that interact with environment 'agents'. Harmonisation is now the act of securing, through proper requirements capture negotiations, as well as through proper interaction dialogue and "vetting" protocols, that the two kinds of 'agents' live up to mutually agreed expectations.

Other than this brief explication we shall not treat this area of requirements engineering further in the present paper.

⋆ *GUIs and Databases* :

Assume that a database records the data which reflects the topology of some railway net, or that records the contents of a time table, and assume that some graphical user interface (GUI) windows represent the interface between man and machine such that items (fields) of the GUI are indeed "windows" into the underlying database. We prescribe and model, as an interface requirements, such GUIs and databases, the latter in terms of a relational, say an `SQL,` database.

type
  Nm, Rn, An, Txt
  GUI = Nm $\xrightarrow[m]{}$ Item
  Item = Txt × Imag
  Imag = Icon | Curt | Tabl | Wind
  Icon == mk_Icon(val:Val)
  Curt == mk_Curt(vall:Val*)
  Tabl == mk_Tabl(rn:Rn,tbl:TPL-**set**)

Wind == mk_Wind(gui:GUI)

Val = VAL | REF | GUI
VAL = mk_Intg(i:Intg) | mk_Bool(b:**Bool**)
| mk_Text(txt:**Text**) | mk_Char(c:**Char**)

RDB = Rn $\overrightarrow{m}$ TPL-**set**
TPL = An $\overrightarrow{m}$ VAL
REF == mk_Ref(rn:Rn,an:An,sel:(An $\overrightarrow{m}$ OVL))
OVL == nil | mk_Val(val:VAL)

Icons effectively designate a system operator or user definable constant or variable value, or a value that "mirrors" that found in a relation column satisfying an optional value (OVL). Similar for curtains and tables. Tables more directly reflect relation tuples (TPL). GUIs (Windows) are defined recursively.

If, for example, the names space values of Nm, Rn, and An, and the chosen constant texts Txt, suitably mirror names and phenomena of the domain, then we may be on our way to satisfying a "classical" user interface requirement, namely that *"the system should be user friendly"*.

Thus a definition, much like the one, of GUI, above, is, in a sense, pulled out of the "thin" air and presented, without much further ado, as part of an interface requirements. Where was its domain "counterpart" ? Or one might just be content with the reuse of the above definition.

For a specific interface requirements there now remains the task of relating all shared phenomena and data to one another via the GUI. In a sense this amounts to mapping concrete types onto primarily relations, and entities of these (phenomena and data) onto the icons, curtains, and tables.

Machine Requirements :

By machine requirements we understand those requirements which are exclusively related to characteristics of the hardware to be deployed (and, in cases even designed) and the evolving software. That is, machine requirements are, in a sense, independent of the specific "details" of the domain and interface requirements, ie., "considers" these only with a "large grained" view.

Performance Issues :

Performance has to do with consumption of computing system resources:. Besides time and (storage) space, there are such things as number of terminals, the choice of the right kind of processeing units, data communication bandwidth, etc.

Time and Space: Time and (storage) space usually are single out for particular treatment. Designated functions of the domain and interface requirements are mandated to execute, when applied, within stated time (usually upper) bounds. This includes reaction times to user interaction. And designated domain information are likewise mandated to occupy, when stored, given (stated) quantities of locations.

Dependabilities :

Dependability is an "ility" "defined" in terms of many other "ilities". We single out a few as we shall later demonstrate their possible discharge in the component software system design.

Availability: There might be situations where a domain description or a domain (or interface) requirements prescription define a function whose execution, on behalf of a user, when applied, is of such long duration that the system, to other users, appear unavailable.

In the examle of the time table system, such may be the case when the *air travel connections* function searchers for connections: The computation, with possible "zillions" of database (cum disk) accesses, "grinds" on "forever".

Accessability: There might be situations where a domain description or a domain (or interface) requirements prescription may give the impression that certain users are potentially denied access to the system.

In the example of the time table system, such may be the case when the time table process non–deterministically chooses between "listening" to requests (queries) from clients and (updates) from staff. The semantics of both the internal ($\sqcap$) and the external ($\Box$) non–deterministic operators are such as to not guarantee fair treatment.

Reliability, Fault Tolerance, Robustness, Safety, and Security: We omit treatment of the many other "ilities" of dependability.

Discussion: We refrain from attempting to formalise the machine requirements of availablity and accessability — for the simple reason that whichever way we today may know how to formalise them, we do not yet know of a systematic way of transforming these requirements into, ie., of "calculating" their implementations.

This is clearly an area for much research.

Maintainabilities :

Computing systems have to be maintained: For a number of reasons. We single out one and characterise this and other maintenance issues.

Adaptability: We say that a computing system is adaptable (not adaptive), wrt. change of "soft" and "hard" functionalities, when change of software or hardware "parts" only involves "local" adaptations.

"Locality", obviously, is our hedge. Not having defined it we have said little, if anything. The idea is that whatever changes have to be made in order to accomodate replacement hardware or replacement software, such changes are to be made in one place: One is able, a priori, to designate these places to within, say, a line, a paragraph, or, at most, a page of documentation.

We shall discuss adaptability further when we later tackle component software design issues.

Perfectability, Correctability and Preventability: A computing system is perfectable (not necessarily perfect), wrt. change (usually improvement) of "soft" and "hard" performance issues [time, space], when such change only involves "local" changes.

A computing system is correctable (not necessarily correct), wrt. debugging "soft" and "hard" bugs, when such change only involves "local" corrections.

A computing system has its failure modes being preventable (not necessarily prevented), wrt. "soft" and "hard" bugs, when regular tests can forestall error modes. For hardware, preventive maintenance is an old "profession". Rerunning standard, accumulative test suites, whenever other forms of maintenance has been carried out, may be one way of carrying out preventive maintenance ?

Portabilities :

By portability we understand the ability of software to be deployed on different computing systems platforms: From legacy operating systems to, and between such systems as (Microsoft's) `Windows`, `Unix` and `Linux`.

Development, Execution, and Maintenance Platforms: One can distinguish between the computing systems platforms on which it may be requirements mandated that development, or execution, or maintenance shall take place.

Feature Interaction Inconsistency and Conflict Analysis :

One thing is to "dream" up "zillions" of "exciting" requirements, whether domain, interface, or machine requirements. Another thing is to ensure that these many individually conceived requirements "harmonise": "Fit together", ie., do not create inconsistencies or conflicts when the "software–to–be" is the basis of computations. Proper formal requirements models allow systematic, formal search for such anomalies [101, 39, 102, 100]. Other than mentioning this 'feature interaction' problem, we shall not cover the problem further. But a treatment of some aspects of requirements engineering would not be satisfying if it completely omitted any reference to the problem.

Discussion :

We have attempted a near–exhaustive listing and partial survey of as complete a bouquet of requirements prescription issues as possible. We have done so in order to delineate the scope and span of formal techniques, as well as the relations, "backward", to domain descriptions, and, as we shall later see, "forward" to software design.

A major thesis of our treatment, maybe not so fully convincingly demonstrated here, but then perhaps more so in our forthcoming books [30], is to demonstrate these relationships, to demonstrate that requirements, certainly domain requirements, can be formalised, and to provide sufficiently refined requirements prescription techniques — especially for domain requirements.

We are not convinced by the requirements engineering principles and techniques presented in todays so–called "best–seller" software engineering text books. So we are, instead, offering our approach as a constructive and viable alternative.

2.4 Software Design

Requirements prescriptions do not specify software designs. Where a requirements prescription is allowed to leave open may ways of implementing some entities (ie., data) and functions, a software design, initially an abstract one, in the form of an architecture design, makes the first important design decisions. Incrementally, in stages, from architecture, via program organisation based on identified components, to module design and code, these stages of software design concretises previously abstract entities and functions.

Where requirements selected parts of a domain for computerisation by only stating such requirements for which a computable representation can be found, software design, one–by–one selects these representations.

Architectures :

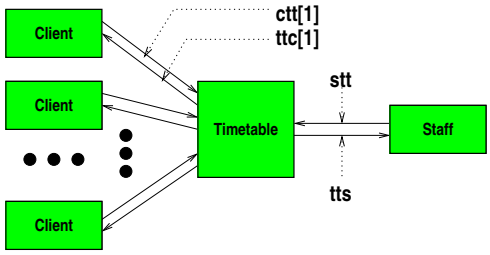
By an architecture design we understand a software specification that implemenents the domain and, maybe, some of the interface requirements. The domain requirements of client_2, staff_2, and timtbl_2, are first transformed, and this is just a proposal, as a system of three parallel processes client_3, staff_3, and timtbl_3. Where client_2 and staff_2, embedded within timtbl_2, we now "factor" them out of timtbl_3, and hence we must provide channels that allow client_3 and staff_3 to communicate with timtbl_3. The communicated values are the denotations, cf. `aplets,` of query and update commands. Whereever client_3 and staff_3 had time tables as arguments they must now communicate the function denotations, that were before applied to time tables, to the timtbl_3 process.

```
scheme ARCH =
  extend ... with
    class
      channel
        ctt QU, ttc VAL, stt UP, tts RES
    value
      system_3: TT → Unit
      system_3(tt) ≡ client_3() ‖ staff_3() ‖ timtbl_3(tt)

      client_3: Unit → out ctt  in ttc  Unit
      client_3() ≡ let q:Query in ctt!Mq(q); ttc?; client_3() end

      staff_3: Unit → out stt  in tts  Unit
      staff_3() ≡ let u:Update in stt ! Mu(u); tts?; staff_3() end

      timtbl_3: TT → in ctt,stt  out ttc,tts  Unit
      timtbl_3(tt) ≡
        (let q=ctt? in ttc!q(tt) end timtbl_3(tt))
        □
        (let u=stt? in let (tt′,r)=u(tt) in tts!r; timtbl_3(tt′) end end)
    end
```

Notice how we have changed the non–deterministic behaviour from being internal ⌈⌉ for timtbl_3 to becoming external [] for timtbl_4. One needs to argue some notion of correctness of this.

An interface requirements was not stated above, so we do it here, namely there shall be a number of separate client_4 processes, each having its identity as a constant parameter. Figure 1[5] illustrates the idea.

Fig. 1. Diagrammatic View of Software Architecture



The system_4 now consists of n client_4 parallel processes in parallel with a basically unchanged staff_4 process and a slightly modified timtbl_4 process. The slightly modified timtbl_4 process expresses willingness to input from any client_4 process, in an external non–deterministic manner. Etcetera:

```
value
  n:Nat
type
  CIdx = {| 1..n |}
channel
  ctt[ 1..CIdx ] QU, ttc[ 1..CIdx ] VAL, stt UP, tts RES
value
  system_4: TT → Unit
  system_4(tt) ≡ ‖ { client_4(i) | i:CIdx } ‖ staff_4() ‖ timtbl_4(tt)

  client_4: CIdx → out ctt  in ttc  Unit
  client_4(i) ≡ let q:Query in ctt[ i ] ! Mq(q) ; ttc[ i ] ? ; client_4(i) end
```

[5] Figures 1–5 also illustrates the use of a diagrammatic language. It is very closely related to the CSP subset of RSL. Other than showing both **scheme** ARCH and Figure 1 we shall not "explain" this diagrammatic language — but it appears to be straightforward. We shall hence 'reason' over constructs (complete diagrams) of this diagrammatic language.

```
staff_3: Unit → out stt  in tts  Unit
staff_3() ≡ let u:Update in stt ! Mu(u) ; tts ? ; staff_3() end

timtbl_4: TT → in { ctt[i],stt[i] i:CIdx }  out ttc,tts  Unit
timtbl_4(tt) ≡
    [] { let q = ctt[i] ? in ttc[i] ! q(tt) end timtbl_4(tt) | i:CIdx }
    [] (let u = stt ? in let (tt′,r) = u(tt) in tts ! r ; timtbl_4(tt′) end end)
```

Component and Object Design :

By a component and object design we understand a transformation of a software architecture design that implements the remaining interface requirements and major machine requirements. Whereas a software architecture design may have been expressed in terms of rather comprehensive processes, component design, as the name intimates, seek to further decompose the architecture design into more manageable parts. Modularisation (ie., module design) goes hand–in–hand with component design, but takes a more fine–grained approach.

One may say, clloquially speaking, that where component design decomposes a software design, and as guided by (remaining interface and by) machine requirements, into successively smaller parts, module design composes these parts from initially smallest modules. The former is, so–to–speak "top–down", where the latter seems more "bottom–up"[6].

At this stage we will just sketch the introduction of new processes that handle the machine requirements of accessability, availability and adaptability. But, as it turns out, it is convenient to first tackle an issue of many users versus just one interface.

Multiplexing :

Instead of designing a time table subsystem that must cater to $n+1$ users we design one that caters to just two users. Hence we must provide a multiplexor, a component which provides for a more–or–les generic interface between, "to one side" n identical (or at least similar) processes, and, "to the other side" one process.

Figure 2 illustrates the idea.

What we have done is to factor out the external non–deterministic choice amongst client process interactions, as documented in timtbl_4 by the distributed choice:

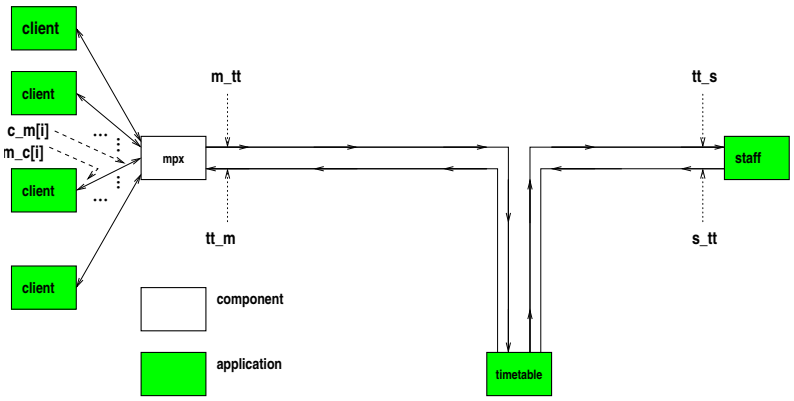
```
[] { let q = ctt[i] ? in ... end | i:CIdx }
```

from that function into the mpx function. The external non–deterministic choice (remaining) among the one "bundled" client input and the staff will, see next, below, later be "moved" to an arbiter function.

We call such a component a multiplexor and leave its definition to the reader.

[6] But we normally refrain from these "figurations" as they depend on how one visualises matters: As a root of further roots, or as a tree of branches.

Fig. 2. Diagrammatic View of Multiplexor Component



Accessability :

To "divide & conquer" between requests for interaction with the time table process from either the ("bundled") clients (via the multiplexor) or the staff, we insert an arbiter component.

Figure 3 illustrates the idea.

Its purpose is to create some illusion of fairness in handling non–determinism. If the arbiter ensures to "listen fairly" to the ("bundled") client and the staff "sides", for example for every f times it handles requests from the client side to then switch to handling one from the staff side, then perhaps some such fairness is achieved. The determination of f, or, for that matter, the arbiter algorithm, is subject to statistical knowledge about the traffic from either side and the service times for respective updates.

This issue of requiring 'fairness' also "spills" over to multiplexor function.

We leave further specification to the reader.

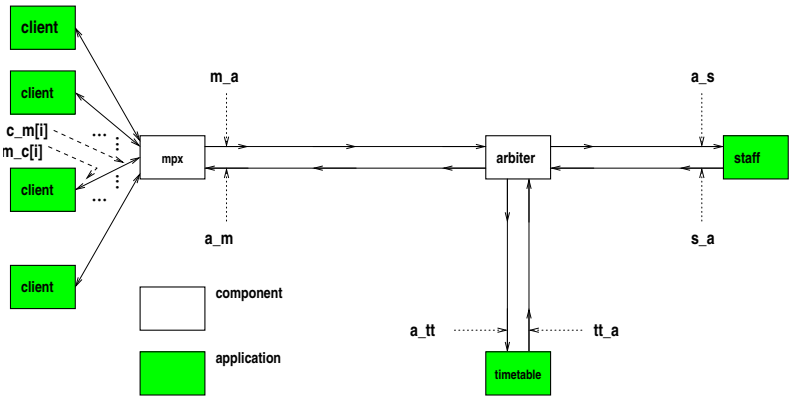
Availability :

The only component (ie., process) that may give rise to "loss of availability" is the time table process. Computing, for example the "at most n change of flight" connections may take several orders of magnitude more time than to compute any other query or update. The idea is therefore to time–share the time table process, and, as a means of exploiting this time–sharing, to redesign (also) the multiplexor component and add a queue component.

Figure 4 illustrates the idea.

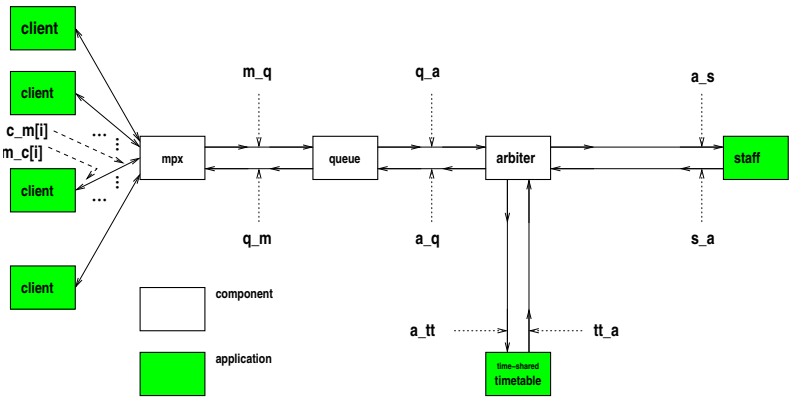
The multiplexor is now to accept successive requests for interaction from multiple clients (or even the same client). And the queueing component is to queue outstanding requests that are, at the same time sent to the time table process. It

Fig. 3. Diagrammatic View of Arbiter Component



Program Organisation with Clients, Multiplexor, Arbiter, Staff, Timetable, and Channels

Fig. 4. Diagrammatic View of Queueing and Time-shared Components



Program Organisation with Clients, Multiplexor, Queue, Arbiter, Staff, time–shared Timetable, and Channels

may respond to previously received requests, "out–of–order". The queueing component will know "back to which clients" request–responses shall be returned.

We leave further specification to the reader.

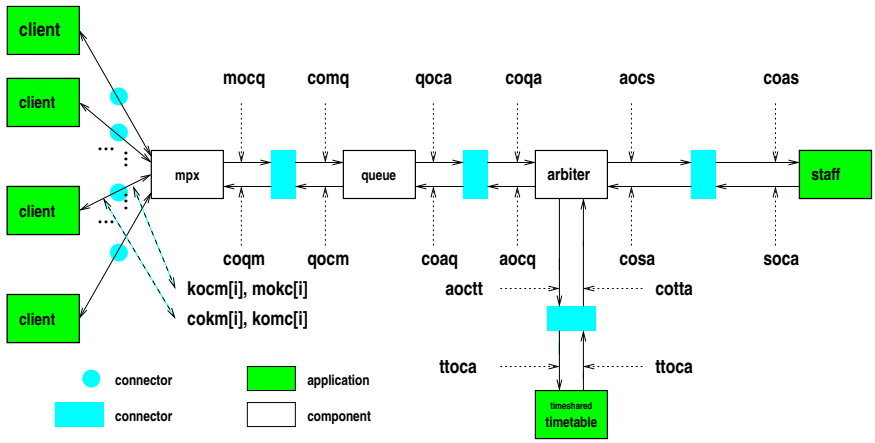
Adaptability :

We have seen how the software design has evolved, on paper, in steps of component design development, into successively more components. Each of these, including those of the client, time table and staff processes may need be replaced. The client and staff components in response to new terminal (ie., PC) equipment, and the time table process in response, say to either new database management systems or new disks, or "both and all" !

If each of these components were developed with an intimate knowledge of (and hence dependency on) the special interfaces that these components may offer, then we may find that adaptability is being compromised. Hence we may decide to insert between neighbouring components so–called connectors. These are in fact motivated last, as in this "example sample development", but are suitably developed first. They "set standards" for exchange of information and control between components. That is, they define protocols.

Figure 5 illustrates the idea.

Fig. 5. Diagrammatic View of Connector Components



Program Organisation with Clients, Multiplexor, Queue, Arbiter, Staff, Timetable, Connectors and Channels

We leave further specification to the reader.

Discussion: Architecture vs. 'Componentry' :

We refer to work by David Garlan and his colleagues, work that relate very specifically to the above [5, 2, 52, 6, 85, 3, 51, 7]. What Garlan et al. call software architecture is not what we call software architecture. Ours is more abstract. Theirs is more at the level of interfacing components. The CMU (ie., the Garlan et al.) work is much appreciated.

Towards a Component Structure Calculus :

We have sketched a "calculus" for deriving component structures. in each step of derivation the "operations" of the "component structure calculus" takes two "arguments". One "argument" is a specific machine (or interface) requirement. The other "argument" is a component structure (or, for the first step, the software architecture). The result of applying the "operation" is a new component structure.

We have still to develop: Identify, research and provide principles and more detailed techniques for when and how to deploy which machine (or interface) reqirements to which component structures. To wit: *"Should one apply the 'availability' requirements before or after the 'accessability' requirements, etc.* It is not yet clear whether the adaptability (and other maintenance "ility") requirements should be discharged, before, in step with, or after the discharge of each of the dependability "ilities". Etcetera.

2.5 Discussion

Summary :

We have completed a *"tour de force"* of example developments. Stepwise 'refinements' of domain descriptions, here for railway nets, and phasewise transformation of domain descriptions into requirements prescriptions and the latter into stages of software designs: Architecture and component designs. It is soon time to conclude and to review our claims.

Validation and Verification :

We have presented aspects of an almost "complete" chain of phases, stages and steps of development, from domains via requirements and software architecture to program organsation in terms of components and connectors. In all of this we have skirted the issues of validation and verification: Validating whether we are developing the right "product", and veryfying whether we develop that "product" right.

An issue that ought be mentioned, in passing, is that of some requirements, typically machine requirements, only being implementable in an approximate manner. One may, for example, have to check with runtime behaviour as to the approximation with which such machine requirements have been implemented [49].

Obviously more than 30 years of program correctness have not gone behind our back: With formalisations of many, if not most, phases, stages and steps it is now easier to state lemmas and theorems of properties and correctness.

Properties of individual descriptions, prescriptions and specifications; correctness of one phase of development wrt. to the previous phase, respectively the same for stages and steps.

We have found, however, that developing software "light": Formally specifying phases, stages and steps, and, in a few, crucial cases, formulating lemmas and theorems (concerning "this and that") seem to capture "most" development mistakes. In any case it is appropriate to end this, the 'triptych' section with the following:

Let $\mathcal{D}$, $\mathcal{R}$ and $\mathcal{S}$ stand for related $\mathcal{D}$omain descriptions, $\mathcal{R}$equirements prescriptions, repectively $\mathcal{S}$oftware specifications. Correctnes of the $\mathcal{S}$oftware with respect to its $\mathcal{R}$equirements can then be expressed as:

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

which, in words, imply: Proofs of correctness of $\mathcal{S}$ with respect to $\mathcal{R}$ typically require assumptions about the domain $\mathcal{D}$.

What could those assumptions be ? Are they not already part of the requirements ? To the latter the answer could be no, in which case it seems that we may have projected those assumptions "away" ! And then these assumptions could be expressed, in the domain descriptions, in the form, for example, of constrained human or support technology behaviours, or of management behaviours, or they could be in the form of script languages in which to express rules & regulations, or they may be properties of the $\mathcal{D}$omain that can be proved in $\mathcal{D}$.

In [90] van Lamsweerde complements the above approximately as follows (our interpretation[7]):

Let $\mathcal{A}$ stand for a notion of 'Accuracy': *Non–functional goals requiring that the state of the input and output software objects accurately reflect the state of the corresponding monitored, respectively controlled objects they represent,* and let $\mathcal{G}$ stand for the set of goals:

$$\mathcal{A}, \mathcal{S} \models \mathcal{R} \quad \text{with:} \quad \mathcal{A}, \mathcal{S} \not\models \textbf{false} \qquad \text{and} \qquad \mathcal{R}, \mathcal{D} \models \mathcal{G} \quad \text{with:} \quad \mathcal{R}, \mathcal{D} \not\models \textbf{false}$$

We find this a worthwhile "twist", and expect more work done to fully understand and exploit the above.

3 Conclusion

It is high time to conclude: To review claims on the background, now, of the approach to software engineering outlined in some detail in this long paper, and on the background of this approach having been deployed for quite some years now, in embryo some 20 years ago, and, since then to an increasing spectrum of the software development phases and stages outlined in Section 2. Some claims were made, notably in Sections 1.2.

[7] As there are unexplained occurrence of $\mathcal{D}$ in van Lamsweerde formula: He additionally uses $\mathcal{A}s$ where we use $\mathcal{D}$

3.1 An *FPO:* A Frequently Posited Objection

It is often claimed that *"formally specifying requirements is a waste of effort as requirements always change, and change frequently."* This statement lacks logic: So what about the required software, it possibly changes even more often, so we should not code it ?

A counter claim, that we shall raise, is that domains change much less, if at all ! The terminology of the financial service industry, or of railways, or of health care — of the intrinsics of these domains — has not changed for maybe perhaps a hundred years.

When you look at what has changed, in the *domain,* it is the *supporting technologies,* the *management & organisation,* and the *rules & regulations.* But these latter depend on very sizable intrinsics. So at least we need formalise the intrinsics. And, before any requirements, for any software that is to monitor or control some supoprting technology, some management & organisation, or some rules & regulations — even though they may change "more often" it probably, as hinted at here, is a good idea to first formalise it, or them, in order to better structure (project, determine, extend, and initialise) the "derivable" rewuirements.

3.2 Business Process Re–engineering (BPR)

BPR is now a fashionable field. We suggest here that it be conducted on the basis of first doing *Business Process Engineering* (BPE), namely domain engineering — but now not followed by requirements engineering (RE), but by BPR. It is "almost the same as RE, but instead of setting up requirements for a machine, it sets up requirements for most human conduct.

3.3 Review and Disposal of Posited Claims

We have put forward a number of claims in Section 1.2. We shall now comment on those.

No Domain Engineering in Current "Best Practices" :

"Today's software development hardly builds on precise domain descriptions": Now that you have seen examples of such descriptions, you will most likely agree. It may be countered that it takes time to create domain descriptions. To which we counter: That time is "reusable": You basically only have to do it once. A software house nomally specialises in products for a specific domain, so why not, "once and for all", or, better, as product after product, version after version, is being requirements prescribed, also continue the development of increasingly comprehensive domain descriptions. They are more stable. They represent the corporate memory, that software house's "cutting, competitive edge".

Domains are "Wider" than Requirements :

"Domain descriptions must be 'wider' than 'related' requirements prescriptions": Whenever a domain descriptions does not cover "significantly more" than the area to be "covered" by a requirements prescription, we find that the requirements prescription development is hampered: The "larger" domain description puts, it seems, the requirements engineer much more at ease as to exactly what should and what can go into a requirements prescription.

Domains before Requirements :

"Radical domain theories must be 'at hand' before requirements prescription is even attempted": As outlined below, a domain theory, ideally, reveals fundamental properties about a domain, properties that may be subconsciously known, but which are often misinterpreted in requirements, leading to inconsistent requirements.

Stability of Domains vs. Stability of Requirements :

"Domain descriptions are more stable than requirements prescriptions": This claim was already disposed of in Section 3.1 above.

Speed–up of Requirements Development :

"Once domain descriptions are available it is faster to develop requirements prescriptions': Obviously. Much work has already been done, and much of requirements is now a projection, determination, extension and initialisation of parts of a donmain description — as outlined in Section 2.3's subsection on 'Domain Requirements'.

Proper Identification of Components :

"Varieties of requirements prescriptions lead to more stable identification of proper components": We hope that the development of components and connectors for the, albeit simple minded time table system of Section 2.4's subsection on 'Component and Object Design', "visualised" in Figures 2–5, can illustrate this claim: Each of the components — other than the client, time table and staff components, are components that relate primarily to machine (or, not shown, interface) requirements. Machine requirements are usually almost identical from application to application, and hence their components are "usually" reusable. But also the domain requirements components of clients, staff and time–shared time table, "cleaned" for all concerns of interface and machinerequirements, now appear in a form that is easier to parameterise and thus make reusable.

• • •

"Problems in systems and software development" :

Our claim here was, and is, that many problems in development stems from lack of established domain descriptions. We have, in Denmark, analysed successful, respectively disastrous software development projects. The successfull ones, although also they "lacked" properly recorded domain descriptions, except for one (which indeed have had such a domain description for almost 20 years), all the successfull projects were characterised by the developers having accumulated, in their practice, and ence in their "brains", an approximate doman description. The failed projects we "new" in the sense that their developers had no prior experience in the domain.

"New paradigms for systems development and processes" :

Our claim was, and is, that domain engineering, and that domain requirements engineering offer such new paradigms. But they must be taken "radically". By that we mean: There must be substantial work done on domain descriptions, both narratives and formalisations. Rough sketching and terminologisation must be taken serious, and careful, seemingly tedious, bureaucratic support must be done for 'documentation'. But it is not tedious: When care is taken to ensure good abstractions and sensible refinements, then such "recording" documents can be a joy to work with.

"New paradigms for modelling and specification languages" :

It should now be abundantly clear: The modelling are based on a rather large variety of abstraction and modelling techniques, and we do not really have to search for new specification languages.

The modelling techniques are well recorded in the published literature — and can be found in my forthcoming text book [30] Chapter 4 (*Phenomena, Concepts, Models and Modelling*), Chapter 6 (*Property and Model Oriented Abstractions*), Chapter 7 (*Specification Programming*), Chapter 8 (*Abstraction and Modelling* — with topics such as *Hierarchical and Compositional Modelling, Denotational vs. Computational Modelling, Configurations: Contexts and States, Time, Space and Space/Time, Agents and Agencies* and *Speech Acts*) — with some domain engineering abstraction and modelling techniques having already been mentioned in this paper.

As for new languages we have a few comments: (i) There are already enough. New proposals are mere variants and integrations of previous languages. What is needed seems to be (ii) graphic subsets of some of these languages, and (iii) unification techniques for the combined use of two or more otherwise disparate languages. We have more to say about this below.

• • •

3.4 A Programme of Current Research

The above section may have presented too "rosy" a picture. As if all problems were "almost solved". Such is, of course, not the case. In this section we will, from our perspective, outline some areas of current research.

Domain Theories :

One–by–one "common domains" — such as transportation domains (raiways, airlines, shipping, trucking, metropolitan passenger transport (busses, taxis, metros, etc.)), logistics, health–care, the financial service industry (banking, insurance, securities trading (stock etc. brokers, traders and exchanges), etc.), "the market" (of consumers, retailers, wholesalers, goods transport, producers, supply chanins, etc.), etc. — need be precisely narrated, terminologised and frormalised. This need be done, not by software houses, but, ideally, by the academic institutions: universities and academy research centres entrusted with research into respective of the listed "commin domains". For a start we, the computing scientists and software engineering researchers and practitioners need lead this new research. Simply because we have the means: We know what is a formal specification, we master abstraction, etc. But eventually, as most of the "common domain" research groups already are competent in "their" mathematics, so they should soon, within 10 years, be competent in "their" informatics.

Rôles for Modal Logics :

One of the lacks, despite more "rosy" claims above wrt. existence of specification languages, is that little of modal logics has "crept" into these languages. Not that I believe that these, existing languages should be extended, see subsection on 'Unifying Theories of Programmin' below, but there simply has not been enough examples of reasonably large scale descriptions using appropriate modal logics: Authors of papers on exciting such modal logics show "factorial function" like tiny examples from which we cannot draw any conclusion as to their specification methodologies. So we need embark on much larger scale deployments of varieties of modal logics for domain engineering-

Multi–agencies and Speech Acts :

Remarks similar to those just above, for 'Modal Logics' can be made for 'Speech Acts', a much "newer" development. We refer to [80].

Unifying Theories of Programming :

It has been mentioned above, in several places, that we probably, most likely have most of the specification paradigms appearing in, that is: made accessible through, one or another specification language. It has then also been remarked that there is little hope to come up with one, "grand" specification lanuage that "features" all ! Instead it has been hinted at, that uses of two or more specification languages, each representing a clean, distinct specification paradigm

(functional, concurrency, modal, etc.), shoukd be favoured — such that one, somehow, can show a consistency "across & between" these uses. This is one of the aspects of what is meant by *'Unifying Theories of Programming'* [63]. The problem is this, "popularly" explained: Let two specification be present. One in for example VDM–SL, the other in TCSP. Let an identifier a be present in both. And let the intention be that "they stand for the same phenomenon" ! In VDM–SL that a may denote a function from environments to functions from states to states, whereas in TCSP it may denote a refusal set (of traces). Which are now the conditions under which one can relate the two occurrences of a. The *'Unifying Theories of Programming'* [63] purports to help give answers to questions like the above. The idea of "combining" the uses, as hinted at, is also referred to as 'Integration of Formal Methods' [38].

Components :
The systematic identification and development of provably corrrect compoents seems to also be an important area of both research, development and propagation.

3.5 A[nother] Grand Challenge

The following is quoted from [19]:

"Inspired by Tony Hoare [62] and Martyn Thomas [89] we put forward a Grand Challenge for computing science."

"Our Grand Challenge takes its departure point in two facts: The current Babel Tower of so–called methods for the development of computing systems, and the *Unified Theories of Programming* quest (also) launched by Tony Hoare (and He JiFeng) [63]."

"The Grand Challenge builds on the assumptions (i) that it is desirable to develop provably correct computing systems, cum software; (ii) that it is desirable to develop such software, in several stages of development, from models of domains via models of requirements to stage and stepwise design of computing systems cum software; (iii) that no one method, cum abstract specification language can ever be devised to cover all foreseeable applications; and (iv) that, hence we can assume the need for varieties of methods cum abstract specification languages."

"The UToPiA acronym here stands for Unified Theories of Programming in Action."

"In brief, our Grand Challenge proposal has three parts:"

Unification of Methods cum Abstract Specification Languages :
"First the Grand Challenge proposes that as many of the current computing systems, cum software, development methods, cum abstract specification languages, be unified so that developments that use two or more of these methods (languages) can integrate their use so as to be able to prove all desired properties."

Unification with Programming Languages & Compilers :

"The Grand Challenge further adapts Tony Hoare's and Martyn Thomas' ideas. It does so in order to complete the use of abstract specification languages. Namely (completion) by final coding in one or more programming languages in which everything that can be expressed can be proved (cf. Martyn Thomas [89]), and for which verifying compilers (cf., Tony Hoare [62]) can be provided. The provably correct steps from abstract specification language specifications to programming language codes must be ensured."

Future "Compliance" :

"The Grand Challenge finally challenges every proposer: Researcher and developer of new methods, new languages — whether specification or programing languages — and new tools, to show that they fit the UToPiA challenge."

Some Comments :

"The Grand Challenge is not a project, and is not aimed at a product for users or industry. The Grand Challenge has as its sôle aim the science of the artifacts specific to the computer cum computing science itself, not to any applications thereof. Pursuing the Grand Challenge shall bring forward that science. Perhaps, as a beneficial side–effect other advances may result, but such goals are merely of seconday nature."

"The present document discusses scientific, technical, financial and collaborative issues connected to any pursuit of the Grand Challenge."

"A rough guess estimates that the present Grand Challenge, if pursued, world wide, by at least some 50 groups, could reach a state, some 10 to 15 years from now, where it could be said: *'It has been achieved.'* "

Final Comments :

These 'final comments' relate to the Grand Challenge. They serve to relate that Grand Challenge to the present paper.

You will see that the Grand Challenge is "in line" with the general ghist of the present paper: There is "a lot out there" that need be accepted — basically "as is" —, yet consolidated and unified.

The 'Programme of Future Research' outlined in Section 3.4 is commensurate with the Grand Challenge, but for each item in that programme can be effected in terms of joint projects. With shorter "time horizons".

3.6 Closing — Some Rather Personal Opinions

This paper has "spread" wide. And it has hinted at somme technical approaches.

The paper was written explicitly to the "Call" as issued by the organisers of the *2002 Monterey Workshop.*

It should be obvious, to the reader, that we are taking explicit issue with the wording of that call. That call had a ring of "optimism": That "radical innovations" will save the industry and its users.

The current author, on one hand, thinks that most of the so–called "radical innovations" were made 10, 20, yes even 30 years ago, but that still few take heed. And, on the other hand, believes that the 'radical innovation' called for, is acceptance of these "old discoveries" ! That the software crisis is not one of lack of methods, principles, techniques and tools. But is one of *"legacy" in the minds of the powers that be,* both in university software engineering departments and in industry. The former seems to kow–tow more to an industry that is not professional (cf. UML), than to computing science, which, on the other hand, oftentimes despairs: *"So why don't they use our wonderful ideas ?"* Much can be said, has been said and written, and much more will go on to be said and written about the sociology (and psychology) of software engineering.

The current author does not despair. He is blessed with former students who commercially successfully "light weight" uses these "old discoveries" !

In universities we see software engineering researchers happily going on producing paper after paper with little attempt, as far as the current author is concerned, to see their oftentimes very worthwhile contributions in a broader programming methodological context. We believe this is done in the present paper (cf. 'The Triptych' and its ramifications).

So we, in academia, have a problem. Also one of "Unification" !

4 Bibliographical Notes

4.1 Own Work

We first give a listing of reports (intended for publication) — report which focus on principles and techniques of *'Abstraction and Modelling'* [26], *'Models, Semiotics, Documents and Descriptions'* [25], *'Domain Modelling'* [20], respectively *'Requirements Modelling'* [29]. In a sense they provide a "capsule view" of [30].

Then we give a listing of mostly reports which focus on domain models. For a railway domain, reports [12, 32–34, 28] and published papers [31, 17], "the market" (the domain for $\mathcal{E}$–Commerce) [18] (being published), a resource management domain [16] (published), a projects (ie., a project management) domain [15, 27], a sustainable development domain [14] (published), a logistics domain [24], a health–care domain [23], a fisheries (industry) domain [13], a financial service–industry domain [35, 22], and many others.

The present stage of most of these "first beginnings" is that they are all somewhat sketchy: Their ongoing development is pursued as much for the reason of testing development principles and techniques — and discovery of new or alternative such, as for the reason of these conjectured, respective domain theories.

4.2 Four Leading "Formal Methods": VDM, Z, RSL, and B

Basically four major, mostly model oriented approaches to formal specification "dominate the market." The simplest, "cleanest" approaches are those of Z

and B. Perhaps the more versatile, and, certainly for RAISE (RSL), comprehensice approaches are VDM (VDM-SL) and RAISE (RSL). Books on these four approaches are as follows:

- VDM: [43, 69, 11, 67, 68, 9, 10, 86, 50].
- Z: [86–88, 97, 47, 81, 65, 70, 42, 73, 95, 74, 61, 98, 8, 83, 84, 48, 36, 37, 82, 96, 66].
- RAISE (besides my three volume text book [30]): [53–55].
- B: [4].

4.3 Temporal Logics

The Duration Calculi (DCs): We have found that many machine and some interface requirements, as well as real–time domain attributes can profitable be expressed in one or another of the Duration Calculi. [40] represents the definite, comprehensive, albeit more foundations than programming methodology oriented, introduction to the Duration Calculi.

Temporal Logic of Reactive Systems: Remarks completely analogous to the above for DCs can be said about the similarly elegant and beautiful Temporal Logic of Reactive Systems — as also embodied in the Stanford Temporal Prover, STeP, [71, 72].

References

1. Special Issue on Scenario Management. IEEE Trans. on Software Engineering, December 1998.
2. G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. SIGSOFT Software Engineering Notes, 18(5):9–20, December 1993.
3. G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology, 4(4):319–364, Oct 1995.
4. J.-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.
5. R. Allen and D. Garlan. A formal approach to software architectures. In IFIP Transactions A (Computer Science and Technology); IFIP Wordl Congress; Madrid, Spain, volume vol.A-12, pages 134–141, Amsterdam, Netherlands, 1992. IFIP, North Holland.
6. R. Allen and D. Garlan. Formalizing architectural connection. In 16th International Conference on Software Engineering (Cat. No.94CH3409-0); Sorrento, Italy, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.
7. R. Allen and D. Garlan. A case study in architectural modeling: the AEGIS system. In 8th International Workshop on Software Specification and Design; Schloss Velen, Germany, pages 6–15, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.
8. R. Barden, S. Stepney, and D. Cooper. Z in Practice. BCS Practitioner Series. Prentice Hall, 1994.
9. J. Bicarregui, J. Fitzgerald, P. Lindsay, R. Moore, and B. Ritchie. Proof in VDM: A Practitioner's Guide. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

10. J. C. Bicarregui, J. S. Fitzgerald, R. Moore, and B. Ritchie. Proof in VDM: Reader's Notes. The Universities of Newcastle upon Tyne and Manchester, and the Rutherford Appleton Laboratory, UK, 1994. Hardcopy available from JSF at The Dept. of Computing Science, University of Newcastle upon Tyne, NE1 7RU, UK, or from BR or JCB at The Informatics Dept., Rutherford Appleton Laboratory, Oxfordshire OX11 0QX, UK. Compressed Postscript is available by ftp from `ftp.cs.man.ac.uk` in directoty `/pub/Proof-in-VDM`.
11. D. Bjøner and C. Jones, editors. Formal Specification and Software Development. Prentice-Hall International, 1982.
12. D. Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. In First ACM Japan Chapter Conference, Singapore, March 7–9 1994. World Scientific Publ. Appendix in collaboration with Søren Prehn and Dong Yulin.
13. D. Bjørner. FISH: A Fisheries Infrastructure — Hardware/Software Concept. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 1998. This document provides a basis for an M.Sc. Thesis project carried out by Audur Thorun Rögnvaldsdottir, Sept. 1998 — Aug. 1999.
14. D. Bjørner. A Triptych Software Development Paradigm: Domain, Requirements and Software. Towards a Model Development of A Decision Support System for Sustainable Development. In E.-R. Olderog and B. Steffen, editors, Festschrift to Hans Langmaack: Correct Systems Design: Recent Insight and Advances, volume 1710 of Lecture Notes in Computer Science, pages 29–60. University of Kiel, Germany, Springer–Verlag, October 1999. Postscript document[8].
15. D. Bjørner. Project Information, Monitoring and Control Systems — A Domain Analysis. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 1999.
16. D. Bjørner. Domain Modelling: Resource Management Strategics, Tactics & Operations, Decision Support and Algorithmic Software. In J. Davies, B. Roscoe, and J. Woodcock, editors, Millenial Perspectives in Computer Science, Cornerstones of Computing (Ed.: Richard Bird and Tony Hoare), pages 23–40, Houndmills, Basingstoke, Hampshire, RG21 6XS, UK, 2000. Palgrave (St. Martin's Press). An Oxford University and Microsoft Symposium in Honour of Sir Anthony Hoare, September 13–14, 1999. Postscript document[9].
17. D. Bjørner. Formal Software Techniques in Railway Systems. In E. Schnieder, editor, 9th IFAC Symposium on Control in Transportation Systems, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk. Postscript document[10].
18. D. Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski), page 34 pages, The Netherlands, December 2002. Kluwer Academic Press.
19. D. Bjørner. UToPiA: Coherent Sets of Computing Systems Development Methods, A Grand Challenge for Computing Science, August 2002.

[8] http://www.imm.dtu.dk/ db/langmaack/hans.ps
[9] http://www.imm.dtu.dk/ db/hoare/tony.ps
[10] http://www.imm.dtu.dk/ db/documents/2ifacpaper.ps

20. D. Bjørner. Domain Engineering — A Prerequisite for Requirements Engineering — Principles and Techniques. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 29, 23, 21, 24, 27, 28, 22]. DRAFT Postscript document.
21. D. Bjørner. E–Business. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 29, 23, 24, 27, 28, 22].
22. D. Bjørner. Financial Service Institutions: Banks, Securities Trading, Insurance, &c. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 29, 23, 21, 24, 28, 27].
23. D. Bjørner. Health–care Systems. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 29, 21, 24, 27, 28, 22].
24. D. Bjørner. Logistics. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 29, 23, 21, 27, 28, 22].
25. D. Bjørner. Models, Semiotics, Documents and Descriptions — Towards Software Engineering Literacy. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [26, 20, 29, 23, 21, 24, 27, 28, 22]. DRAFT Postscript document.
26. D. Bjørner. Principles and Techniques of Abstract Modelling — Some Basic Classifications. — Towards a Methodology of Software Engineering. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 20, 29, 23, 21, 24, 27, 28, 22]. DRAFT Postscript document.
27. D. Bjørner. Projects & Production: Planning, Plans & Execution. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 29, 23, 21, 24, 28, 22].
28. D. Bjørner. Railways Systems: Towards a Domain Theory. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 29, 23, 21, 24, 27, 22].
29. D. Bjørner. Requirements Engineering — Some Principles and Techniques — Bridging Domain Engineering and Software Design. Technical report, Informatics and

Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2003. This paper is one of a series of papers currently being submitted for publication: [25, 26, 20, 23, 21, 24, 27, 28, 22]. DRAFT Postscript document.

30. D. Bjørner. The SE Book: Principles and Techniques of Software Engineering, volume I: Abstraction & Modelling (750 pages), II: Descriptions and Domains (est.: 500 pages), III: Requirements, Software Design and Management (est. 450 pages). [Publisher currently (June 2003) being negotiated], 2003–2004.
31. D. Bjørner, C. George, and S. Prehn. Scheduling and Rescheduling of Trains, chapter 8, pages 157–184. *Industrial Strength Formal Methods in Practice,* Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999. Postscript document[11].
32. D. Bjørner, D. Y. Lin, and S. Prehn. Domain Analyses: A Case Study of Station Management. In KICS'94: Kunming International CASE Symposium, Yunnan Province, P.R.of China. Software Engineering Association of Japan, 16–20 November 1994. .
33. D. Bjørner, S. Prehn, and C. W. George. Formal Models of Railway Systems: Domains. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM. Postscript document[12].
34. D. Bjørner, S. Prehn, and C. W. George. Formal Models of Railway Systems: Requirements. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM. Postscript document[13].
35. D. Bjørner, V. Rosario, and M. Helder. A Normative Model of Concrete Banking Operations — Banking Rules & Regulations and Staff/Client Behaviours. Research, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, June 1998. (Need be revised: Some typos etc. !).
36. L. Bottaci and J. Jones. Formal Specification Using Z: A Modelling Approach. International Thomson Publishing, London, 1995.
37. J. P. Bowen. Formal Specification and Documentation Using Z: A Case Study Approach. International Thomson Computer Press, 1996.
38. A. Bryant and L. Semmens, editors. Methods Integration, Electronic Workshops in Computing. Springer-Verlag, 1996.
39. M. J. Butler. Feature interaction analysis using Z. Åbo Akademi University, Finland, 1994.
40. Z. Chaochen and M. R. Hansen. Duration Calculus: A formal approach to real–time systems. Monographs in Theoretical Computer Science. Springer–Verlag, 2002 (2003). A 238 page manuscript was sent to the potential publisher Monday 15 July 2002. This book collects the work of the main originator and one of the main contributors to the theory of duration calculi. As such the book represents a dozen years of research.

[11] http://www.it.dtu.dk/ db/racosy/scheduling.ps
[12] http://www.imm.dtu.dk/ db/racosy/domain.ps
[13] http://www.imm.dtu.dk/ db/racosy/requirements.ps

41. G. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida, pages 430–440. IEEE, 1984.
42. I. Craig. The Formal Specification of Advanced AI Architectures. AI Series. Ellis Horwood, Sept. 1991.
43. C. J. e. D. Bjørner. The Vienna Development Method: The Meta-Language, volume 61 of Lecture Notes in Computer Science. Springer-Verlag, 1978.
44. A. Dardenne, S. Fikas, and A. van Lamsweerde. Goal–Directed Concept Acquisition in Requirements Elicitation. In Proc. IWSSD–6, 6th Intl. Workshop on Software Specification and Design, pages 14–21, Como, Italy, 1991. IEEE Computer Society Press.
45. A. Dardenne, A. van Lamsweerde, and S. Fikas. Goal–Directed Requirements Acquisition. Science of Computer Programming, 20:3–50, 1993.
46. R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal–Driven Requirements Elaboration. In Proc. FSE'4, Fourth ACM SIGSOFT Symp. on the Foundations of Software Enginering, pages 179–190. ACM, October 1996.
47. A. Diller. Z: An Introduction to Formal Methods. John Wiley & Sons, 1990.
48. A. Diller. Z: An Introduction to Formal Methods. John Wiley & Sons, 2nd edition, 1994.
49. M. Feather, S. Fikas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behaviours. In Proc. IWSSD'98, 9th Intl. Workshop on Software Specification and Design, Isobe, Japan, April 1998. IEEE Computer Society Press.
50. J. Fitzgerald and P. G. Larsen. Software System Design: Formal Methods into Practice. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1997. To appear.
51. D. Garlan. Formal approaches to software architecture. In Studies of Software Design. ICSE '93 Workshop. Selected Papers, pages 64–76, Berlin, Germany, 1996. Springer-Verlag.
52. D. Garlan and M. Shaw. An introduction to software architecture, pages 1–39. World Scientific, Singapore, 1993.
53. C. George, P. Haff, K. Havelund, A. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. The RAISE Specification Language. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
54. C. George, A. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. The RAISE Method. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
55. C. W. George, H. D. Van, T. Janowski, and R. Moore. Case Studies using The RAISE Method. FACTS (Formal Aspects of Computing: Theory and Software) and FME (Formal Methods Europe). Springer–Verlag, London, 2002. This book reports on a number of case studies using RAISE (Rigorous Approach to Software Engineering). The case studies were done in the period 1994–2001 at UNU/IIST, the UN University's International Institute for Software Technology, Macau (till 20 Dec., 1997, Chinese Teritory under Portuguese administration, now a Special Administrative Region (SAR) of (the so–called People's Republic of) China).
56. J. A. Goguen and M. Girotka, editors. Requirements Engineering: Social and Technical Issues. Academic Press, 1994.
57. J. A. Goguen and C. Linde. Techniques for Requirements Elicitation. In Proc. RE'93, First IEEE Symposium on Requirements Engineering, pages 152–164, San Diego, Calif., USA, 1993. IEEE Computer Society Press.

58. S. J. Greenspan, J. Mylopoulos, and A. Borgida. Capturing More World–Knowledge in Requirements Specification. In Proc. 6th ICSE: Intl. Conf. on Software Engineering, Tokyo, Japan, 1982. IEEE Computer Society Press.
59. S. J. Greenspan, J. Mylopoulos, and A. Borgida. A Requirements Modelling Language. Information Systems, 11(1):9–23, 1986. (About RML).
60. P. Haff, editor. The Formal Definition of CHILL. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1981.
61. I. J. Hayes. Specification Case Studies. Prentice Hall International Series in Computer Science, 2nd edition, 1993.
62. C. Hoare. A Grand Challenge for Computer Science. Presented at the UNU/IIST 10th Anniversary Symposium, Lisboa, Portugal, March 19, 2002, and at the IFIP WG2.3 Meeting, Åbo/Turku, Finland, August 12, 2002. March, August 2002.
63. C. Hoare and H. J. Feng. Unifying Theories of Programming. Prentice Hall, 1997.
64. A. Hunter and B. Nuseibeh. Managing Inconsistent Specifications: Reasoning, Analysis and Action. ACM Transactions on Software Engineering and Methodology, 7(4):335–367, October 1998.
65. M. Imperato. An Introduction to Z. Chartwell-Bratt, 1991.
66. J. Jacky. The Way of Z: Practical Programming with Formal Methods. Cambridge University Press, 1997.
67. C. Jones. Systematic Software Development Using VDM. Prentice-Hall, 1986.
68. C. Jones. Teaching notes for systematic software development using vdm. Technical Report UMCS 86-4-2, Univ. of Manchester, 1986.
69. C. B. Jones. Software Development A Rigorous Approach. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.
70. D. Lightfoot. Formal Specification using Z. Macmillan, 1991.
71. Z. Manna and A. Pnueli. The Temporal Logic of Reactive Systems: Specifications. Addison Wesley, 1991.
72. Z. Manna and A. Pnueli. The Temporal Logic of Reactive Systems: Safety. Addison Wesley, 1995.
73. J. A. McDermid and P. Whysall. Formal System Specification and Implementation using Z. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1992. Withdrawn.
74. M. A. McMorran and S. Powell. Z Guide for Beginners. Blackwell Scientific, 1993.
75. J. Mylopoulos. Information Modelling in the Time of revolution. Information Systems, 23(3/4):127–155, 1998.
76. J. Mylopoulos, L. Chung, and B. Nixon. Representing and Using Non–Functional Requirements: A Process–oriented Approach. IEEE Trans. on Software Engineering, 18(6):483–497, June 1992.
77. J. Mylopoulos, L. Chung, and E. Yu. From Object–Oriented to Goal–Oriented Requirements Analysis. CACM: Communications of the ACM, 42(1):31–37, January 1999.
78. B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships between Multiple Views in Requirements Specifications. IEEE Transactions on Software Engineering, 20(10):760–773, October 1994.
79. O. Oest. VDM from research to practice. In H.-J. Kugler, editor, Information Processing '86, pages 527–533. IFIP World Congress Proceedings, North-Holland Publ.Co., Amsterdam, 1986.
80. H. M. Petersen. Agents and Speech Acts: A Semantic Analysis. Master's thesis, Informatics and Mathematical Modelling, Computer Science and Engineering, Bldg. 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs. Lyngby, Denmark, 20 June 2002.

81. B. F. Potter, J. E. Sinclair, and D. Till. An Introduction to Formal Specification and Z. Prentice Hall International Series in Computer Science, 1991.
82. B. F. Potter, J. E. Sinclair, and D. Till. An Introduction to Formal Specification and Z. Prentice Hall International Series in Computer Science, 2nd edition, 1996.
83. D. Rann, J. Turner, and J. Whitworth. Z: A Beginner's Guide. Chapman & Hall, London, 1994.
84. B. Ratcliff. Introducing Specification Using Z: A Practical Case Study Approach. International Series in Software Engineering. McGraw-Hill, 1994.
85. C. Shekaran, D. Garlan, and et al. The role of software architecture in requirements engineering. In First International Conference on Requirements Engineering (Cat. No.94TH0613-0); Colorado Springs, CO, USA, pages 239–245, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.
86. D. Sheppard. An Introduction to Formal Specification with Z and VDM. International Series in Software Engineering. McGraw Hill, 1995.
87. J. M. Spivey. Understanding Z: A Specification Language and its Formal Semantics, volume 3 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Jan. 1988.
88. J. M. Spivey. The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
89. M. Thomas. A Grand Challenge for Computer Science. Presented at the CoLogNET/-FME Industry Day, August 25, 2002, at the FLoC'02 Federated Logic Conference, Copenhagen, Denmark. July 2002.
90. A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In Proceedings 22nd International Conference on Software Engineering, ICSE'2000. IEEE Computer Society Press, 2000.
91. A. van Lamsweerde, R. Darimont, and E. Letier. Managing Conflicts in Goal–Driven Requirements Engineering. IEEE Transaction on Software Engineering, 1998. Special Issue on Inconsistency Management in Software Development.
92. A. van Lamsweerde and E. Letier. Integrating Obstacles in Goal–Driven Requirements Engineering. In Proc. ICSE–98: 20th International Conference on Software Engineereering, Kyoto, Japan, April 1998. IEEE Computer Society Press.
93. A. van Lamsweerde and L. Willemet. Inferring Declarative Requirements Specification from Operational Scenarios. IEEE Transaction on Software Engineering, pages 1089–1114, 1998. Special Issue on Scenario Management.
94. A. van Lamsweerde and L. Willemet. Handling Obstacles in Goal–Driven Requirements Engineering. IEEE Transaction on Software Engineering, 2000. Special Issue on Exception Handling.
95. J. C. P. Woodcock. Using Standard Z. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1993. In preparation.
96. J. C. P. Woodcock and J. Davies. Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science, 1996.
97. J. C. P. Woodcock and M. Loomes. Software Engineering Mathematics. Addison-Wesley Publishing Company, 1989.
98. J. B. Wordsworth. Software Development with Z: A Practical Approach to Formal Methods in Software Engineering. Addison-Wesley Publishing Company, 1993.
99. E. Yu and J. Mylopoulos. Understanding "why" in Software Process Modelling, Analysis and Design. In Proc. 16th ICSE: Intl. Conf. on Software Engineering, Sorrento, Italy, 1994. IEEE Press.
100. P. Zave. Classification of Research Efforts in Requirements Engineering. ACM Computing Surveys, 29(4):315–321, 1997.

101. P. Zave and M. A. Jackson. Techniques for partial specification and specification of switching systems. In S. Prehn and W. Toetenel, editors, VDM'91: Formal Software Development Methods, volume 551 of LNCS, pages 511–525. Springer-Verlag, 1991.
102. P. Zave and M. A. Jackson. Requirements for telecommunications services: an attack on complexity. In Proceedings of the Third IEEE International Symposium on Requirements Engineering (Cat. No.97TB100086), pages 106–117. IEEE Comput. Soc. Press, 1997.