Towards Posit & Prove Calculi for Requirements Engineering and Software Design

In Honour of the Memory of Professor Ole–Johan Dahl

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, URL: www.imm.dtu.dk/~db

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 another ("earlier") paper [6] we advocate(d) a strong rôle for domain engineering. We there argued that domain descriptions are far more stable than are requirements prescriptions for support of one or another set of domain activities. In the present paper we shall argue, that once, given extensive domain descriptions, it is comparatively faster to establish trustworthy and stable requirements than it is today. And we shall further, presently, 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. In this contribution we shall thus reason, at a meta-level, about two major phases of software engineering: Requirements engineering, and software design. We shall suggest a number of requirements engineering and software design concerns, stages and steps.
The paper represents work in progress. It is based on presentations of *"topics for discussion"* at the IFIP Working Group 2.3. Such presentations are necessarily of "work in progress" — with the aim of the presentation being to solicit comments. Hence the paper ("necessarily") is not presenting "final" results.

1 Introduction

Our concern, in the present and in most of our work in the last almost 30 years, has been that of trying to come to grips with principles and techniques for software development.

The present paper sketches some such principles and techniques for some of the stages within the phases of requirements engineering and software design.

Our lecture notes, [7], the reader will find a rather comprehensive treatment of these and "most other related" software engineering issues !

1.1 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.
- 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 another ("earlier") paper [6] we advocate(d) a strong rôle for domain engineering.

- We there argued that domain descriptions are far more stable than are requirements prescriptions for support of one or another set of domain activities.
- In the present paper we shall argue, that once, given extensive domain descriptions, it is comparatively faster to establish trustworthy and stable requirements than it is today.
- And we shall further, presently, 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.

In this contribution we shall thus reason, at a meta-level, about two major phases of software engineering:

- Requirements engineering, and
- software design.

We shall suggest a number of requirements engineering and software design concerns, stages and steps, notably, for

- requirements:
  - Domain requirements,
  - interface requirements, and
  - machine requirements.
- Specifically:
  - Domain requirements projection,
  - determination,
  - extension, and
  - initialisation.
- And Software Design:
  - Architecture design, and
  - component determination and design.

1.2 Claimed, 'Preliminary' Contributions

We claim that this paper reports on two kinds of methodological contributions: The "posit & prove calculation" principles of projection, determination, extension and initialisation; and the principle of the stepwise "posit & prove calculation" of software architecture design.

2 Requirements Engineering

2.1 Delineation of Requirements

From [23] 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 [29]."*

We shall, in this paper, not cover the pragmatics of why software is needed, and we shall, 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.

2.2 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 [16, 8, 18, 9, 15, 28, 20, 14, 10, 17, 25, 24, 26, 19, 27]. 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.

2.3 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 shall in software engineering 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.

2.4 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 "belong" to the domain, we first introduce them while creating the domain requirements. We call this *domain extension*. The distinction, thus is purely pragmatic.
– Initialisations:
  In describing a domain, such as we for example described the "space" of all time tables, we must, for each specific time table, designate the "space" of all its points of departures and arrivals. If our requirements involve these departure and arrival points (airports, railway stations, bus depots, harbours), 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 (ie., contextually check) 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. But first we need to tak an aside: To bring "an entire" domain model" ! That is, the next section ("A Domain Intrinsics Model") does not belong to the requirements modelling phase of development, but to the domain modelling phase of development.

A Domain Intrinsics Model :

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.

We now return from our domain modelling detour. In the next four sections we illustrate a number of domain requirements steps. There are other such steps ('fitting', etc.) which we will leave un–explained.

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_4: TT → VAL, client_4(tt) ≡ let q:Query in (Mq(q))(tt) end
        staff_4: TT → TT × RES, staff_4(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_5: TT → Unit
        timtbl_5(tt) ≡
            (let v = client_4(tt) in timtbl_5(tt) end)
            ⌈⌉
            (let (tt′,r) = staff_4(tt) in timtbl_5(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. (The example is, perhaps a bit contrived.)

```
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) ≡
          case q of
            ...
            mk_conn(fa,ta,n) → λtt:TT • ...
          end
    end
```

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

Initialisations :

We remind the reader that this and the immediate three

Initialisation here means: From a given input of flight journies to create an initial time table (ie., an initial database). Ongoing changes to time tables have been provided for through the insert and delete operations — already defined. In their definition, however, we skirted an issue 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 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 TI_TBL_7 =
  extend TI_TBL_6 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 imagination, 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."*[1]

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

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

"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, ie., the 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, ie., the system–to–be" does not honour these contractual relationships, then the "software, ie., the 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 "posit & prove calculus" for deriving domain requirements. So far we have identified four operations in this "posit & prove 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. The "proof" part of the "posit & prove calculus" is a conventional proof of correctness between the two arguments.

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 ?".

2.5 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 air traffic 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)

Observe how the "content" values of icons and curtains are allowed to be any values, as now defined:

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

RDB = Rn $\xrightarrow[m]{}$ TPL-**set**
TPL = An $\xrightarrow[m]{}$ VAL
REF == mk_Ref(rn:Rn,an:An,sel:(An $\xrightarrow[m]{}$ 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"*.

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.

2.6 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 singled 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.

Other Dependabilities :

We omit treatment of the reliability, fault tolerance, robustness, safety, and security "ilities".

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 "posit & prove 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.

Performability :

A computing system satisfies a performability requirements, wrt. change (usually improvement) of "soft" and "hard" performance issues [time, space], when such change only involves "local" changes.

Correctability :

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

Preventability :

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`.

One can distinguish between the computing systems platform on which it may be requirements mandated that development shall take place — in contrast to the computing systems platforms on which it may be requirements mandated that execution and maintenance shall take place. Etcetera.

2.7 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 [30, 31, 29]. 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.

2.8 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 lecture notes [7], 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 have tried, in contrast to todays software engineering (including requirement engineering) text books, to provide some principles and techniques for structuring the requirements documents to be constructed by requirements engineers.

3 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.

3.1 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_4, staff_4, and timtbl_5, are first transformed, and this is just a proposal, as a system of three parallel processes client_arch, staff_arch, and timtbl_arch. Where client_4 and staff_4, embedded within timtbl_5, we now "factor" them out of timtbl_5, and hence we must provide channels that allow client_arch and staff_arch to communicate with timtbl_arch. The communicated values are the denotations, cf. `aplets,` of query and update commands. Whereever client_arch and staff_arch had time tables as arguments they must now communicate the function denotations, that were before applied to time tables, to the timtbl_arch process.

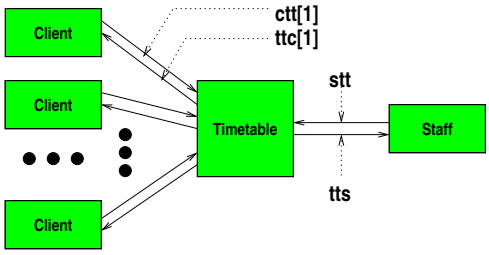
```
scheme ARCH =
  extend ... with
    class
      channel
        ctt QU, ttc VAL, stt UP, tts RES
      value
        system_arch: TT → Unit, system_arch(tt) ≡ client_arch() ‖ staff_arch() ‖ timtbl_arch(tt)

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

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

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

Notice how we have changed the non–deterministic behaviour from being internal ⌈⌉ for timtbl_5 to becoming external [] for timtbl_arch. 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_arch_1 processes, each having its identity as a constant parameter. Figure 3.1[2] illustrates the idea.

The system_arch_1 now consists of n client_arch_1 parallel processes in parallel with a basically unchanged staff_arch_1 process and a slightly modified timtbl_arch_1 process. The slightly modified timtbl_arch_1 process expresses willingness to input from any client_arch_1 process, in an external non–deterministic manner. Etcetera:

[2] Figures 3.1–3.2 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 3.1 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.



Architecture: A Time−table with Clients and Staff
Fig. 1.

value
n:**Nat**
type
CIdx = {| 1..n |}
channel
ctt[1..CIdx] QU, ttc[1..CIdx] VAL, stt UP, tts RES
value
system_arch_1: TT → **Unit**
system_arch_1(tt) ≡ ‖ { client_arch_1(i) | i:CIdx } ‖ staff_arch_1() ‖ timtbl_arch_1(tt)

client_arch_1: CIdx → **out** ctt **in** ttc **Unit**
client_arch_1(i) ≡ **let** q:Query **in** ctt[i] ! Mq(q) ; ttc[i] ? ; client_arch_1(i) **end**

staff_arch_1: **Unit** → **out** stt **in** tts **Unit**
staff_arch_1() ≡ **let** u:Update **in** stt ! Mu(u) ; tts ? ; staff_arch_1() **end**

timtbl_arch_1: TT → **in** { ctt[i],stt[i] i:CIdx } **out** ttc,tts **Unit**
timtbl_arch_1(tt) ≡
[] { **let** q = ctt[i] ? **in** ttc[i] ! q(tt) **end** timtbl_(tt) | i:CIdx }
[] (**let** u = stt ? **in let** (tt′,r) = u(tt) **in** tts ! r ; timtbl_arch_1(tt′) **end end**)

3.2 Component Design

By a component design (as action) we understand a set of transformations, from a software architecture design, that implements the remaining interface requirements and major machine requirements, to the component design (as document). Whereas a software architecture design may have been expressed in terms of rather comprehensive processes, component design, as the name intimates, seeks to further decompose the architecture design into more manageable parts. Object modularisation (ie., module design) goes hand–in–hand with component design, but takes a more fine–grained approach. We are not yet ready, in our research, to relate these "posit & prove transformations" to the refinement calculus of for example Ralph Johan Back [21]. There are (at least) points: First there are too many issues predicating which refinements to choose. These issues represent the judicious prioritisation between a multitude of domain, interface and machine requirements: Which to consider and implement before others ? Secondly the "refinement steps" illustrated next seem rather large. Hence for a proper refinement calculus to be proposed we need express the "large" steps, it seems, in terms of sequences of "smaller" steps. We are far from ready to embark on such an endeavour.

This is why we have used the phrase: *Posit & Prove Calculus* in the title of this communication.

One may say, colloquially speaking, that where component design decomposes a software design (and as guided by (remaining interface and by) machine requirements) into successively smaller

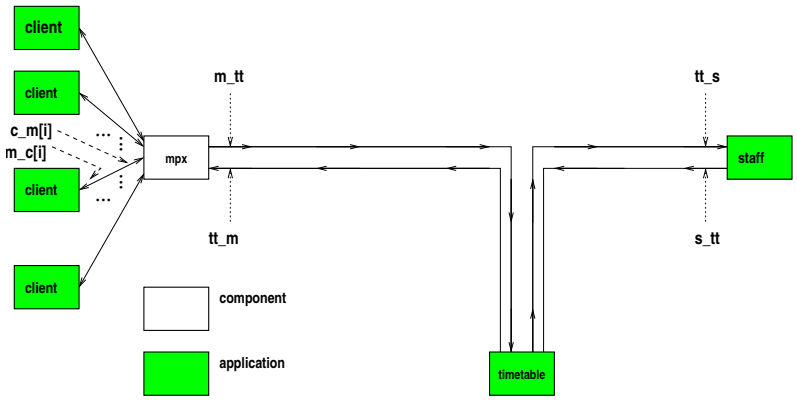
parts, module design composes these parts from initially smallest modules. The former is, so–to–speak "top–down", where the latter seems more "bottom–up"[3].

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–less generic interface between, "to one side" n identical (or at least similar) processes, and, "to the other side" one process.

Figure 3.2 illustrates the idea.



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

Fig. 2.

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.

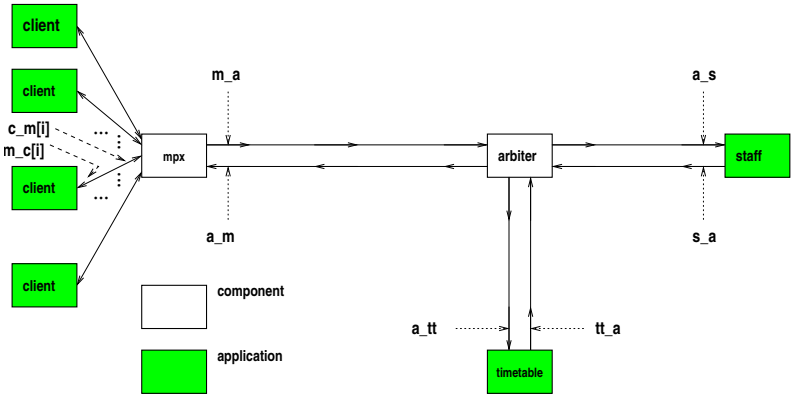
Accessability :

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 3.2 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,

[3] 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.



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

Fig. 3.

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 the multiplexor function.

Letting the arbiter also handle urgency of requests is natural. It would, in our view, be a further 'accessability' requirements.

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 3.2 illustrates the idea.

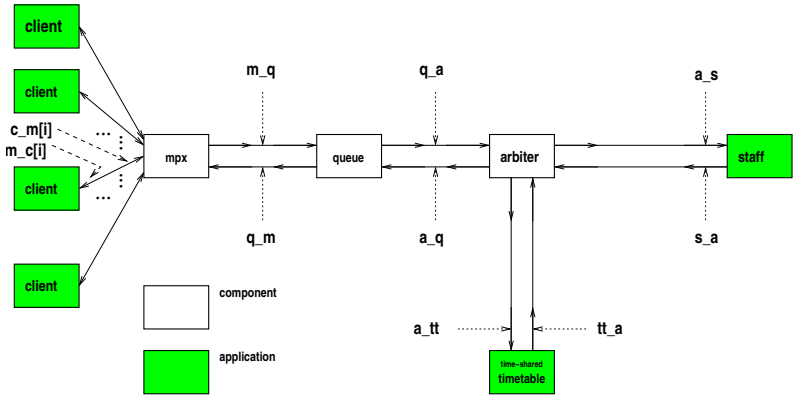
The multiplexor is now to accept successive requests for interaction from multiple clients (or even the same client). And the queueing component is to queue outstanding requests that are, at the same time sent to the time table process. It may respond to previously received requests, "out–of–order". The queueing component will track "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 abstractly developed first. They "set standards" for exchange of information and

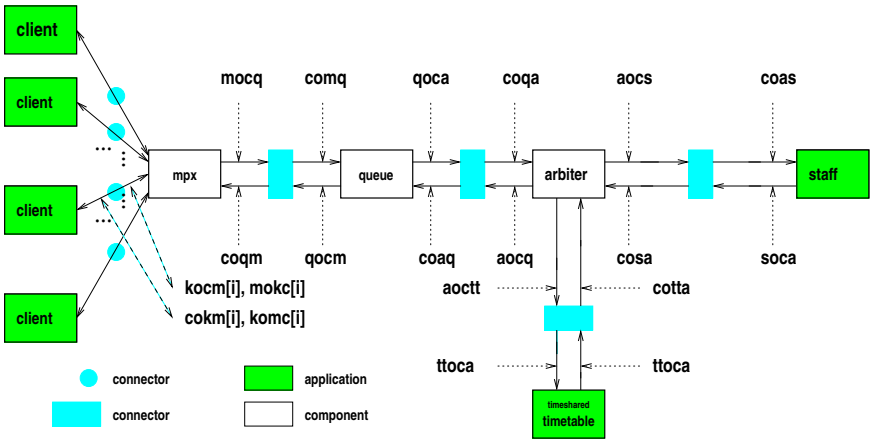


Program Organisation with Clients, Multiplexor, Queue, Arbiter, Staff, time–shared Timetable, and Channels

Fig. 4.

control between components. That is, they define abstract, simple protocols. Once all components have been "inserted" one may refine the protocols to suit these compponents.

Figure 3.2 illustrates the idea.



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

Fig. 5.

We leave further specification to the reader.

Architecture vs. 'Componentry' :

We refer to work by David Garlan and his colleagues, work that relate very specifically to the above [3, 1, 13, 4, 22, 2, 12, 5]. 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, that is of the connectors mentioned above under Adaptability. The CMU (ie., the Garlan et al.) work is much appreciated.

3.3 Towards "Posit & Prove Calculi" for Architecture and Component Structure Derivation

We have sketched a "posit & prove 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.

We have not covered in this paper any "posit & prove calculus" aspects of deriving architectures from domain requirements.

4 Conclusion

4.1 Summary

We have completed a *"tour de force"* of example developments. Stepwise 'refinements' of domain descriptions, here for time tables, 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.

4.2 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 [11].

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 shown how to develop software "light". That is: Formally specifying phases, stages and steps, and, in a few, crucial cases, formulating lemmas and theorems (concerning "this and that"). We have found that developing software "light" seems 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 [23] van Lamsweerde complements the above approximately as follows (our interpretation[4]):

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:

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

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

4.3 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 3's subsection on 'Component and Object Design', "visualised" in Figures 3.2–3.2, 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.

4.4 A Programme of Current Research

We briefly recall that there seems to be interesting research issues in better understanding and providing methodological support for the derivation of domain requirements and the derivation of component structures.

4.5 Acknowledgements

The author is tremendously grateful for a very careful review of a referee. I wish to state that many of the very reasonable concerns of the referee are indeed very valid concerns also of mine. Space, however, did not permit me, in a paper as "far sweeping" as this has become, to address each and all of these concerns.

4.6 A Caveat

This paper represents work in progress. It is based on presentations of *topics for discussion* at the IFIP Working Group 2.3. Such presentations are necessarily of "work in progress" — with the aim of the presentation being to solicit comments. As just said above, the anonymous referee has just done that. Thanks.

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

References

1. 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.
2. G.D. 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.
3. 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.
4. 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.
5. 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.
6. Dines Bjørner. Domain Engineering: A "Radical Innovation" for Systems and Software Engineering ? In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, page 54 pages, Heidelberg, October 7–11 2003. Springer–Verlag. The Zohar Manna International Conference, Taormina, Sicily 29 June – 4 July 2003.
7. Dines 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 (March 2003) being negotiated], I: Fall 2003, II: Spring 2004, III: Summer/Fall 2004 2003–2004.
8. A. Dardenne, S. Fikas, and Axel 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.
9. A. Dardenne, Axel van Lamsweerde, and S. Fikas. Goal–Directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.
10. R. Darimont and Axel van Lamsweerde. Formal Refinement Patterns for Goal–Driven Requirements Elaboration. In *Proc. FSE'4, Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 179–190. ACM, October 1996.
11. M. Feather, S. Fikas, Axel 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.
12. 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.
13. D. Garlan and M. Shaw. *An introduction to software architecture*, pages 1–39. World Scientific, Singapore, 1993.
14. Joseph A. Goguen and M. Girotka, editors. *Requirements Engineering: Social and Technical Issues*. Academic Press, 1994.
15. Joseph 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.
16. S. J. Greenspan, John Mylopoulos, and A. Borgida. A Requirements Modelling Language. *Information Systems*, 11(1):9–23, 1986. (About RML).
17. 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.
18. John 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.
19. John 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.
20. 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.
21. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Heidelberg, Germany, 1998.
22. 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.

23. Axel 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.
24. Axel 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.
25. Axel van Lamsweerde and E. Letier. Integrating Obstacles in Goal–Driven Requirements Engineering. In *Proc. ICSE–98: 20th International Conference on Software Enginereering*, Kyoto, Japan, April 1998. IEEE Computer Society Press.
26. Axel 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.
27. Axel van Lamsweerde and L. Willemet. Handling Obstacles in Goal–Driven Requirements Engineering. *IEEE Transaction on Software Engineering*, 2000. Special Issue on Exception Handling.
28. E. Yu and John Mylopoulos. Understanding "why" in Software Process Modelling, Analysis and Design. In *Proc. 16th ICSE: Intl. Conf. on Software Engineering*, Sorrento, Italy, 1994. IEEE Press.
29. Pamela Zave. Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.
30. Pamela Zave and Michael A. Jackson. Techniques for partial specification and specification of switching systems. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 511–525. Springer-Verlag, 1991.
31. Pamela Zave and Michael 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.