Where do Software Architectures come from ?
Systematic Development from Domains and Requirements

A Re-assessment of Software Engineering ?

Dines Bjørner
IT/DTU, Technical University of Denmark

Abstract

In this paper we show how details of a software design emerges in two steps: **software architecture** and **program organisation** and from first having established careful descriptions of the application **domain** and of functional and non-functional **requirements**.

A major aim & objective of this paper is a reassessment of software engineering in the context of extensive use of formal techniques (formal methods: specification and calculi) and programming methodological: to illustrate how software designs partially evolve from careful (formal) requirements descriptions which themselves partially evolve from careful (formal) application domain descriptions.

We believe that this paper covers some new concepts:

- the careful construction of informal as well as formal descriptions of application domains without any reference to software (i.e. computing);
- the systematic (posit/invent & verify/prove) "derivation" of requirements from domain descriptions;
- and the separation of requirements and software design concerns:
  - functional requirements as implemented through software architectures, and
  - non-functional requirements as implemented through program organisations.

Contents

1 Introduction

1.1 A Software Development Paradigm

We cover main notions of a rigorous software development paradigm. We propose the following decomposition:

1. **Software Engineering** ≈
   (a) **Domain Engineering** √
   (b) ⊕ **Requirements Engineering** √
   (c) ⊕ **Software Design**
      i. **Software Design** ≈
      ii. *Software Architecture* √
      iii. ⊕ *Program Organisation* √
      iv. ⊕ (Further) Refinement Steps
      v. ⊕ Coding
2. **Software Development** ≈ All of the above!

We cover √ using:use

- primarily **Formal Specification**. √
- Indicatingand indicate **Design Calculi**.
- Thus the main subject is
  Programming Methodology:
  How to construct software √

Thus we may be contributing to a partial reassessment of software engineering in the two contexts outlined above: the "triptych" paradigm of domain engineering + requirements engineering + software design, and the extensive, to us inescable, use of formal specification and rigorous to formal reasoning.

1.2 Method & Methodology

1.2.1 On Method

- We take **Method** to mean :
  - A set of **Principles**
  - for **Selecting** and **Applying**
  - **Techniques** and **Tools**

  in order efficiently to:
  - **Analyse** and
  - **Synthesise** (i.e. construct)

  efficient artefacts (here: software).

We will identify a number of such
principles, techniques and **tools**.

It is with some resignation that I lament the use of the term 'formal methods'. In the context of our definition of method above basically no method can be formal! One cannot formalise the (analysis and selection) principles. But techniques and tools (like languages) can be formal: based on mathematics, notably algebras and logic.

1.2.2 On Methodology

Since no one realistic piece of software technology can be developed strictly according to one clearly inter-related set of principles (etc.), i.e. according to one method, but possibly according to several such, we need define the notion of methodology:

- We take **Methodology** to mean:
  - The the **Study** and **Knowledge** of **Methods**

Michael Jackson is exploring a concept of *problem frames* [1, 2]. Typical problem frames are such which relate to (i) programming (and specification language) translation, (ii) reactive (i.e. control) systems, (iii) information systems, (iv) workpiece systems, (v) transaction processing, etc. [3].

1.3 Domains, Requirements, Design

We review and relate the three major components of our software engineering "triptych":

- *Software* **Design:** $\mathcal{S}$

  Before we can design (structure, code) the software, i.e. **how** the software (i.e. the *machine*) should operate, we must know **what** is should be doing, i.e. its requirements.

- **Requirements:** $\mathcal{R}$

  Before we can express the requirements we must "understand" the application domain. This is so because functional requirements are expressed solely using professional terms of that domain.

  Requirements descriptions are **validated** by the stake-holders.

- **Domain:** $\mathcal{D}$

  We **narrate**, establish a **terminology** and, here, also **formalise** terms (viz.: nouns and verbs) of the professional sub-language of the **stake-holders** (owners, managers, workers, clients, etc.) of the application domain.

  Domain descriptions are **validated** by the stake-holders.

  Usually we describe far more of the domain than the **environment** needed for software design **verification.**

- **"Derivability":**

  - **Functional Requirements** form an *instantiation,* a *projection* and possibly an *extension* of of a possibly *normative* **domain**.
  - **Software architectures** (grossly speaking) implement *externally observable properties,* i.e. the *functional* **requirements**.

    Program organisations are otherwise "invented"!
  - Software **design** in general proceed by *stepwise transformation*s and/or *verified refinement*s.

- **Correctness Verification:**

  Proofs ($\models$) of correctness of software $\mathcal{S}$ wrt. requirements $\mathcal{R}$ usually proceed by making assumptions about the domain $\mathcal{D}$:

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

That is: Reference is often made to $\mathcal{D}$ when proving the entailment $\mathcal{S} \models \mathcal{R}$. Hence we include it ($\mathcal{D}$) to the left of $\models$.

This fact, that assumptions usually need be made about the environment — outside that which is "immediately" part of the requirements model — further argues for a thorough modelling of the application domain: beyond that of immediate concerns to requirements, and normative rather than (just) instantiated domains.

Paraphrasing Barry W. Boehm:[4]

- Validation: *Getting the Right Product*
- Verification: *Getting the Product Right*

1.4 Aims & Objectives of this Paper

The goal of this paper is to broadly, and in a pedagogical clear and didactically "complete" style, make the reader aware:

1. of the triptych/triple Paradigm:
   Domain + Requirements + Software Design,
2. of Abstraction and
3. Modelling Techniques,
4. of Refinement concepts and
5. of Correctness concerns,
6. that Large–scale Development can be
   tackled in a *Trustworthy, Efficient* manner,
7. of need for *Formal Techniques* and
8. that New Product Ideas can Arise
   when using Formal Specification within the
   Domain / Requirements / Software Design Paradigm

1.5 Message Delivery

- **Question:**
  How will we deliver the message ?
- **Answer:**
  - Through a "major" example.
  - By exemplifying:
    - principles and
    - techniques,
  - By **Reading** the example!
    - Not teaching "how to write".
    - But "how to appreciate".
    - By telling you about Formal Specification.

1.6 Overview of Paper

2 Software Architectures

This section has several parts. First (Sect.2.1) we discuss briefly what an architecture is and give — right-away — an example CSP-like [5, 6] "program" (i.e. specification) and the annotation of an architecture for a "small" system. Then we ask the question: *"where did this architecture come from?"*. Usually we are just presented with the architecture "program". We then propose an answer to the question: *"the architecture came, partially, from a set of requirements, and these came, partially, from a description of the application domain.* We therefore describe the example domain — that of an airline time-table and its users [prospective airline clients and airline staff] — (Sect.2.1.6) and we describe example requirements for a computerised support for using the time-table (Sect.2.1.7). Then we "derive" a software architecture, partially from those requirements (Sect.2.2). Since that software architecture — in keeping with our proposed definition of what an architecture is — does not implement the non-functional requirements, we finally "derive" a program organisation from the non-functional requirements and the software architecture (Sect.3).

The example was first reported, in a rather more detailed (read: cumbersome) manner in [7]. Then it became the subject of Mr. Crilles Jansen's M.Sc. Thesis project [8, 9]. The present version represents a significantly abstracted version of these reports — and should be more accessible!

2.1 A Pre–view Architecture

2.1.1 "What is an Architecture ?" — Proposal 1

The concept of 'software architecture' is a pragmatic one. One can define it one way, or in another way, or . . . ! With some background in the CMU 'School' (Garlan et. al) [10] we first propose:

- **Basic Notions:** There are some basic concepts that determine major characteristics of an architecture:
  - Types, Components, Ports, Glue, Connectors, i.e.: *Value spaces, Processes, Channels.*
  - Events, Communication, i.e.: *Input, Output, Rendez-vous.*
- **Configuration:** The above basic concepts enter into configurations:
  - A set of typed components: e.g. processes, parallel, non–deterministic alternatives.
  - Each component with a set of typed ports.
  - A set of typed connectors: e.g. channels, "fixed" at component ports.
  - The ability to observe typed events at ports, that is: typed communication over channels.

Perhaps an explanation is due:

- Since an architecture specification is generally one that describes a computable "thing", and since we wish to have (mental and mechanical) tools that support us in our writing (conceptualisation) process, we naturally expect an architecture specification to evolve around a type space, that is: A set of sets of values. The terms: Components, ports, glue, connectors are mentioned from Garlan — but we "immediately translate" them into notions of process algebras, notably CSP: Processes and channels.

  For the components to interact (via channels) events signal communication, i.e., in the CSP "jargon", the simultaneous "rendez–vous" (instantaneous "meeting") of respective output and input events together with their "transfer" of values from the output (later designated by the "shriek": !) to the input process (later designated by the "question mark": ?).

  An architecture is now the configuration of components (processes) with internal (component to component) and possibly also external channels (coming from, or leading to an "outside").

This characterisation is in terms of "implementation, construction".

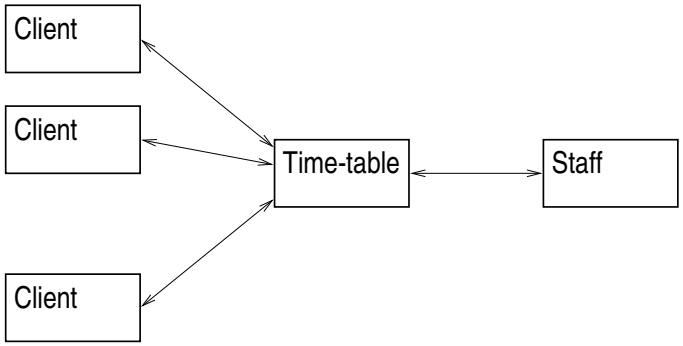
2.1.2 "A Picture is Worth a Thousand Words!"

Let us illustrate, by a diagram, an architecture of $n+2$ components: n client components (processes), a staff component (process), and a time-table component (process). That is: The architecture is apparently about clients, staff and time-tables! Let us assume that the time-table is an airline time-table, that the clients are prospective passengers and that the staff is that of an, or the, airline.

The rectangular boxes shall denote processes and the two-way arrows shall denote two-way channels.

But such diagrams, as well as the first characterisation, leaves something to be desired: *"What is it all about?".*

Figure 1: An Airline Time-table Software Architecture



2.1.3 "What is an Architecture ?" — Proposals 2–3

We therefore "look behind" the mere syntax of boxes and arrows to ask for the meaning of the configuration.

- An 'ideal' Software Architecture allows one to observe 'exactly' that some functional requirements have been satisfied.[1]
- A Software Architecture reveals those, and only those (hence external) interfaces which the user — here clients and staff — can observe (including manipulate).

Now we are getting to where we want to be!

2.1.4 A Formalisation of a Software Architecture

Yet, before we return to a more systematic treatment of the notions of domain and requirements, let us briefly state that the above diagram covers a distributed sofware "system" that allows clients to query a time-table, and staff to operate upon this time-table (potentially changing it). Now we can postulate a formal process (and channel, etc.) description which details the above diagram.

The formal model outlines a type space, channels and values of query and update operations as well as of the client, staff and timetable processes. Without much further a-do:[2]

type
Index, TT, VAL, RES, Φ = TT$\rightarrow$VAL, Ψ = TT$\rightarrow$TT$\times$RES

[1] The two terms 'ideal' and 'exactly' are "balanced": either both are present, or both can be omitted — yielding different meanings.

[2] In this paper we use a rather simplified version of RSL [11], the RAISE method [12] Specification Language. In this simplified version we treat all channel and value definitions as those of objects (in the sense of RSL).

channel
ctt[i:Index]:Φ,ttc[i:Index]:VAL, stt:Ψ, tts:RES
value
ϕ_1,ϕ_2,..,ϕ_q:Φ, ψ_1,ψ_2,..,ψ_u:Ψ
cϕ: **Unit** → Φ **Unit**, cϕ() ≡ $\phi_1 \sqcap \phi_2 \sqcap .. \sqcap \phi_q$
sψ: **Unit** → Ψ **Unit**, sψ() ≡ $\psi_1 \sqcap \psi_2 \sqcap .. \sqcap \psi_u$

client: i:Index → **in** ttc[i] **out** ctt[i] × **Unit**
client(i) ≡ (ctt[i]!cϕ() ; ttc[i]? ; client(i))
staff: **Unit** → **in** tts **out** stt × **Unit**
staff() ≡ (stt!sψ() ; tts? ; staff())

timetable: TT → **in** ctt[i],stt **out** ttc[i],tts × **Unit**
timetable(tt) ≡ **let** tt′ =
[] { **let** ϕ = ctt[i]? **in** ttc[i]!ϕ(tt) ; tt **end** | i:Index }
[] **let** ψ = stt? **in let** (tt′,r) = ψ(tt) **in** tts!r ; tt′ **end end**
in timetable(tt′) **end**

system() ≡ ‖ { client(i) | i:Index } ‖ timetable(tt) ‖ staff()

Annotations:

- Index is a finitely enumerated type. Its elements indexes a finite set of clients.
- TT designates a set of time–tables.
- VAL designates a set of time–table query values.
- RES designates a set of time–table update responses.
- Φ designates a set of time–table query functions (from time-tables to values).
- Ψ designates a set of time–table update operations (from time-tables to new time-tables and responses).
- ctt, ttc, stt, tts designate arrays of client channels to, resp. from time-tables, and a pair of staff to, resp. from time-table channels.
  (The previous two-way arrows have each been replaced by two one-way arrows.)
- $\phi_1, \phi_2, .., \phi_q$ stands for an arbitrary set of (pre-compiled) query functions as internal non-deterministically chosen by the client query selection process **c**ϕ.
- $\psi_1, \psi_2, .., \psi_u$ stands for an arbitrary set of (pre-compiled) update operations as internal non-deterministically chosen by the staff update selection process **s**ψ.
- client(i) designates an i'th client who, wrt. time-tables, can be modelled as issuing query requests (via a client to time-table channel), awaiting results, whereupon the client resumes activities.

- staff designates a staffer who, wrt. time-tables can be modelled as issuing update requests (via the staff to time-table channel), awaiting responses, whereupon the staffer resumes activities.
- timetable designates the time-table which external non-deterministally awaits for action requests from either one of a number of clients or from a staffer.

  A client query is performed on the time-table, the result value communicated back to that cleint, whereupon the unchanged time-table resumes activities.

  A staff update is performed on the time-table, causing a possible change to the time-table with a response sent back to the staffer, wereupon the possibly changed time-table resumes activities.
- system is the (parallel) configuration of a number of clients, and a staffer around a shared time-table.

2.1.5 "Software Architectures: From Where?"

Clearly, how do we know whether a proposed architecture actually solves a, or the, problem as first posed? Not unless we have carefully recorded, written down, systematically developed and analysed, relevant notions of both the application domain and the requirements. So, instead of just postulating an

- **Architecture**

— as is too often done without much serious, recorded development work preceding its presentation — we propose, in line with our "triptych dogma":

- via **Requirements**
- from **Domains**

With this we have completed our initial "analysis" of the problem whose general solution this paper intends to illustrate.

There now follows two pre-cursor sections (Domain: 2.1.6, Requirements: 2.1.7) to the systematic unfolding of an architecture.

2.1.6 A Domain

After having acquired knowledge about the application domain we write it down. The description usually has four parts of which we shall only show three — and then only rather cursorily. The four parts are:

- **A Synopsis:**
  A synopsis is a very brief characterisation of "what it is all about" — where 'it' here refers to a, or the, domain!

- **A Terminology:**
  A terminology defines, in concise, natural (national) language, all the professional terms used in the synopsis, elsewhere in the terminology and in the narratives. For a domain description the terms are those of the stake-holders of the application domain.

  We shall not present the terminology in this paper since the example is drawn from a domain familiar to most readers.

- **Narratives:**
  Since there may be many stake-holders involved in any one software (to be developed) there may be a need for several concordant narratives.

  Each domain narrative is a (the stake-holder) professional language indicative (*as it is* [1, 2]) description of the domain as seen from the side of that, or those, stake-holders. The descriptions must only use such technical terms that are part of the stake-holder's professional language (whose terms are defined in the terminology). (Usually the narratives also define these terms.) The narrative otherwise present all relevant designations, definitions and refutable assertions — basically following Michael Jackson's *Principles of Description* [1, 2]. Basically a narrative narrates all the phenomena of the domain believed to be of relevance to any software that might be required for that domain. The domain narratives, as well as their formal counterparts, are here proposed described normatively. That is: Generically — purportedly relevant to several distinct occurrences of the domain. Thus our normative description of the airline time-table system should be "true" of "conceivably all" airlines' time-tables! This is in contrast to an instantiated domain description. The latter describes a specific instance, say for Japan Airlines.

- **Formal Models:**
  Each narrative is complemented with a formal model. Sometimes one model may capture several narratives. Sometimes parameter instantiations of a parameterised model brings about desired stake-holder perspectives.

We otherwise refer to [13] for a more elaborate presentation of principles and techniques of domain modelling.

Domain Synopsis

- The domain is that of airline time-tables being queried by clients and being updated by staff.

Staff querying the time-table act as clients.

Domain Narrative

- Time–tables are further undefined.
- Operations on time-tables either leave the time-table unchanged (as for users), or updates the time-table (as for staff).
- Client queries extract time-table information

- Responses inform staff on time-table update.
- Client queries (staff updates) are abstracted as functions ϕ (respectively operations ψ).

Comments: Nothing is said about whether there are just one, or several, clients. The narrative is deliberately left under-specified.

No cause for ambiguity seems to arise — yet!

We also do not elaborate on specific query nor on specific update operations. That is: We have deliberately chosen to 'abstract' these — as already hinted in the pre-view example!

Here we see, perhaps an extreme example of a normative domain description. We allow for any range of operations as long as clients cannot change the time-tables. This allows us great freedom — as we shall later see.

Domain Formalisation

type
TT, VAL, RES
Φ = TT $\to$ VAL
Ψ = TT $\to$ TT$\times$RES

value
ϕ_1,ϕ_2,...,ϕ_q:Φ
ψ_1,ψ_2,...,ψ_u:Φ

client: TT $\to$ VAL
client(tt) $\equiv$ **let** ϕ:Φ = $\phi_1 \bigsqcap \phi_2 \bigsqcap ... \bigsqcap \phi_q$ **in** ϕ(tt) **end**

staff: TT $\to$ TT$\times$RES
staff(tt) $\equiv$ **let** ψ:Ψ = $\psi_1 \bigsqcap \psi_2 \bigsqcap ... \bigsqcap \psi_q$ **in** ψ(tt) **end**

Annotations:

- **TT, VAL, RES** designates sets of time-tables, query result values and update responses.
- Φ, Ψ designates sets of query functions, respectively update operations.
- ϕ, ψ designates archetypical query functions, respectively update operations.
- **client** designates that a client can perform (any one of a set of) queries on time-tables.
- **staff** designates that a staffer can perform (any one of a set of) updates on time-tables.

Comment:

- That's all!
- In this domain description, informal as well as formal, we "underspecify": we do not, for example, say whether there are many clients, but speak of an archetypical client, and of archetypical query functions and update operations.

2.1.7 Requirements

Requirements express **what** the desired software should offer — which properties are deemed desirable. Requirements, from partly a pragmatic point, can be "divided" into two sets: the functional, resp. the non-functional requirements.

Functional Requirements

Functional requirements "derive" from, or as Michael Jackson and Pamela Zave [14] expresses it: "resides in", "the domain". Constructing functional requirements, to us, include the following techniques: Projection, instantiation and extension. These will now be illustrated.

We see the following general principles guiding our construction of functional requirements:

- **Domain Projection:**
  In constructing requirements the requirements engineer must — together with other stake-holders — decide on which segment of the (normative) domain will be referred to in the requirements.
- **Domain Instantiation:**
  And if the domain description was normative, it must now be instantiated to the particular properties and desirabilities of the specific customer's domain.
- **Domain Extension:**
  Once the completed software system is inserted in the domain it becomes part of that (now new) domain. But this is not the "extension" that we here mean. Rather we mean: provision of concepts and facilities that could have been part of the old domain — for example if there had otherwise been sufficient supporting technology or human resources available.
- **Shared Phenomena:** **External ↔ Internal States**
  Some of the type space of the domain may be called its state. Determining the state space of a domain is a, sometimes pragmatically determined, art! We are here referring only to those parts of the domain whose values may change independently of what goes on in the machine (i.e. in the sotware described computations)!
  The shared phenomena issues of requirements are now: (i) how to initially input, i.e. initialise the [machine] internal state, (ii) how to make sure that there is a reasonably updated version of the external state in the machine, (iii) how to vet the state input and updates, (iv) ..., etc.

We otherwise refer to [13] for a more elaborate presentation of principles and techniques of requirements modelling.

- **Projection:**

  We decide to maintain "all" of the domain types of time-tables, query & update operations, and values of and responses to such operations.

  type TT, Φ, Ψ, VAL, RES

  The client and the staff functions will, however, be redefined.

- **Instantiation:** Three parts. Part 1:

  We decide, however, to instantiate time-tables, values and responses, to concrete types. In doing so we shall introduce hitherto "hidden" types: Flight numbers, airport names and (departure and arrival) times. And we shall likewise introduce a simple notion of journey: namely that of a one or several "leg" trip with which a flight number is associated in the time-table and which the flight is expected to travel.

  – **TT, RES, VAL:** Concrete types:

  ```
  type
    Fn, An, T,
    TT       = Fn m→ Journey
    Journey′ = An m→ (arrival:T × departure:T)
    Journey  = {| j:Journey′ • wf_Journey(j) |}
  value
    wf_Journey: Journey′ → Bool
    wf_Journey(j) ≡ ...
    Ans: TT → An-set
    Ans(tt) ≡ ∪ { dom tt(fn) | fn:Fn • fn ∈ dom tt }
  type
    RES == ok | nok | VAL
    VAL  =  TT | Journey
  ```

  So for each flight number a time-table records two or more airport names, and for each of these the gate arrival and departure times. Well-formedness of a journey may, for example, mean that for any given flight (number) no two distinct airport arrival and departure times overlap, and that for a time-table, as a whole, for example, that for any given airport no two flights take-off or land at the same time, or even "within prescribed intervals". The Ans function observes all the airport names mentioned in a time-table. Responses to staff operations allow them to also act as clients.

- **Instantiation** Part 2:

– Φ Specific Operation:
  We now instantiate the query operations to a specific few: (i) browse time–table (see it all, no argument) and (ii) display a specific (fn) journey:

```
type
  Query == mk_brws() | mk_disp(fn:Fn)
value
  type M_q: Query → Φ
  M_q(q) ≡
    case q of
      mk_brws()
        → λtt.tt,
      mk_disp(fn)
        → λtt.if fn ∈ dom tt
                then tt(fn) else [ ] end
    end
```

  $\mathcal{M}_q$ applied to query "commands" yield denotations: functions from time-tables to values.

The above semantics definition, $\mathcal{M}$, is classical.

• **Instantiation** Part 3:

– Ψ Specific Operation:
  Similarly the staff update operations are: (i) initialise (reset) to (pre-loaded) time–table, tt:TT, (ii) add a "flight" (fn,j) to tt and (iii) delete a "flight" fn from tt — with responses ok, or nok — in addition to the client queries.

```
type
  Update == mk_init() | mk_add(fn:Fn,j:Journey)
            | mk_del(fn:Fn) | Query
value
  tt_init : TT
  type M_u: Update → Ψ
  M_u(u) ≡ case u of:
      mk_init() → λtt.(tt_init,ok),
      mk_add(fn,j)
        → λtt.if fn ∈ dom tt
                then (tt,nok) else (tt ∪ [fn↦j],ok) end
      mk_del(fn)
        → λtt.if fn ∈ dom tt
                then (tt \ {fn},ok) else (tt,nok) end
      _ → (tt,M_q(u))
    end
```

$\mathcal{M}_u$ applied to update "commands" yield denotations: functions from time-tables to pairs of time-tables and responses.

- **Extension:**

  We now extend the domain by a query whose calculation usually is too cumbersome for ordinary humans to carry out and with any assurance of having done it right!

  - Additional Φ: inquire of an at most m:**Nat** flight change journey between two given airports (fa,ta):An×An

```
type
  Conn = Fn × (An × Fn)*
  VAL  = TT | Journey | Conn-set
  conn == mk_conn(m:Nat,fa:An,ta:An)
value
  M_q: conn → TT → Conn-set
  M_q(mk_conn(m,fa,ta))(tt) as conns
    pre: ...
    post: ...
```

We leave it to the reader to define appropriate pre/post conditions for this query which is expected to calculate the (possibly empty set of) all the zero, one, two, etc., m stop-over and flight change travels between two given airports. We also leave it to the reader to decide whether circular trips are allowed, and, in general, what properties the result should otherwise possess!

Non-functional Requirements

Constructing non-functional requirements, to us, include the following techniques: Initialisation, system configuration, dependability (accessability, availability, reliability, security and safety), and CHI. Formalised techniques for handling some of these will now be illustrated. An illustrative list of kinds (and examples) of non-functional requirements include:

- **Performance:** *"such and such" a function must execute in at most* t *seconds,* or *the software must require at most* k *kilo bytes of storage.*
- **Dependability:** This term is debatable, anyway:
  - **Availability:** *the software must provide "fair" access amongst multiple users contending for access to shared resources ...*
  - **Reliability** *the software must not give rise to "down-times" of at most a total of* m *minutes total over any* d *day period (interval).*
  - **Accessability:** *provide simultaneity in use of resources where reasonable* — in order to make these accessible to a "largest number" of users concurrently.

  - **Safety:** *the software must "guard against" ("such and such") failures in the (sensor and actuator) hardware technology to which the machine is attached — i.e. loss of property and lives, due to software errors, must remain within legal limits during the life-time of the system in which the software resides.*
  - **Security:** *the software must secure that only "appropriately" authorised personnel and/or "machinery" (agents) can cause software to respond in any way!*[3]

- **Maintainability:** Maintenance has to do with regular or irregular changes to or inspection of the software. There seems to be at least three causes for maintenance:
  - **Perfective:** *improve response time, lower storage consumption, ...*
  - **Adaptive:** *change software so that the machine can be attached to new sensors, actuators or other input/output.*
  - **Corrective:** *the software must be "easy to debug"!*
- **Platform: Hardware/Software:** *the software must "execute" on "such & such" platforms (IBM PC compatibles, Linux with CORBA (!), etc.).*
- **CHI: Computer/Human Interface:** *the resulting machine must be "user friendly".*
- *&c.*

Usually the exression (as well as a later software design [implementation]) of non-functional requirements entail cumbersome, detailed pre-cautions and descriptions.
At the same time it is often difficult, with today's known techniques, to describe sufficiently precisely, including to formalise, many types of non-functional requirements (e.g. "user-friendliness"). Even for some of the types (for example: maintainability and performance) where one can indeed formalise, no techniques are yet known whereby one can argue (e.g. prove) that such properties are indeed satisfied by the resulting sofware — other than what "time will tell!".

We continue or mixture of narrative and formal description.

- **Initialisation:** Provide for *Initial time-table*

  We do not show formalisation — but could!

  The requirements specification of this really amounts to a functional requirements but for the systems facilities management personnel: *"One person's program organisation is another person's software architecture".*

  Describing this aspect amounts to the specification of an entire "little sub-system": with own external data formats, data vetting, input (load) procedures, etc.

- **System Configuration:** Provide for n:Index *Clients*, one *Staff* and one *Time-table*

[3] Our favourite "definition" of what constitutes a secure system goes like: *A system is secure if non-authorised agents while accessing (i.e. in contact with) the system (i) cannot ascertain which functional (or other) properties it has, (ii) cannot ascertain the structure of the system, and (iii) does not know whether it knows (i) and/or (ii)!*

The specific property requirements for data communication cabling, data transfer, etc., form one part of this aspect. We abstract part of this and show the required configuration in the architecture model.

- **Availability:** Provide *"Fair" Service* to *Clients* and *Staff*

  The clients and the staff "share" the same time-table. Repeated requests for query, respectively update operations by clients and staff must be handled such that neither party is excluded from access indefinitely. We will informally suggest a solution to this problem only after we have presented the software architecture.

- **CHI:** *Icons, Prompt* menu and *Result* windows

  This particular requirement states that the user interface to the computer (i.e. the machine) be effected through a display screen sub-system (for example with an appropriately attached keyboard and mouse). That system can, as suggested here, be decomposed into three parts: icons which are like clickable buttons — one for each of the query and/or update operations (as appropriate), prompt fields where arguments to respective operations (also as appropriate) can be provided by the user, and a result window where query values or update responses are displayed.
  We omit treatment of scrollability.

  type
  CHI = Icon × Prompt × Result
  Icon = browse | display | connection
  | init | add | delete | nil
  Prompt == Query | Update | conn | nil
  Result == VAL | RES

  The seemingly one icon whose value can range over seven enumerated values, including nil (which models "no setting" of the value), can be implemented for example by a curtain icon with six fields, or by six distinct, labelled icons.

  Perhaps this aspect of the computer-human interface (CHI) should be termed a functional requirement. Since the suggested icon, menu and result window concepts are usually part of a perceived solution to general "user-friendliness" we list it here.

Functional Requirements — "Wrap Up"

Till now we have refrained from re-describing the client and the staff functions described in the domain. Now that we have described the CHI "state" — in terms of the icons, prompt fields and the result/response window — we can finalise the requirements.

Redefinition of Client Function

We treat the CHI issue rather abstractly. Primarily we require that whatever that interface was before the issuance of a query (update) operation by a client or staffer (respectively by a staffer), after the requested operation has been obeyed that interface shall have its three components suitably reflect the transaction.


```
value
  client: CHI → TT → CHI
  client(,,)(tt) ≡
    let icon = browse ⌈⌉ display ⌈⌉ connection in
    case icon of:
      browse
        → (browse,nil,M_q(mk_brws())(tt)) end,
      display
        → let fn:Fn • fn ∈ dom tt ∨ ... in
            (display,
              mk_disp(fn),
              M_q(mk_disp(fn))(tt)) end,
      connection
        → let m:Nat,da,ta:An • {da,ta} ⊆ Ans(tt) ∨ ... in
            (connection,
              mk_conn(m,da,ta),
              M_q(mk_conn(m,da,ta))(tt)) end
    end
```

Annotations:

- The client now also focus on the computer screen (CHI) with — as here: an abstraction — of the icons, the prompt menu and the result window, and, as before, the time-table.
- The client internal (⌈⌉) non-deterministically issues either a browse, a display or a connection request for query (by "clicking" an appropriate real, physical button).
- A browse request can be immediately served. The request results in a new 'chi' state in which the icon button for browse is shown (depressed), in which there is no (hence nil) prompt, and in which the result window shows the (possibly scrollable) time-table — which should result from performing the browse ϕ function on the time table!
- A display request causes a prompt for an appropriate flight number. Either it is one which is in the time-table, or (...) it is not such a one (the latter causing an empty journey to be displayed!).[4] Etcetera!
- A connection request causes a prompt for the client to submit a suitable (or, as it may be, less suitable (...)) pair of (from and to) airport names. Etcetera!

In the above model, we have, for simplicity, omitted any description of any temporal sequence that relates the three parts of the CHI interface. It may appear from the formalisation above that the full triple is not presented before the entire operation has executed. One may, however, take the liberty to interpret this specification as follows: first the icon part of the CHI state is set, then the prompt part, and finally the result part.

[4] We do not deal with error diagnostics — the subject of another, proper paper!

Redefinition of Staff Function

Remarks similar to those stated above in connection with, and prior to the formal description of client function can be repeated. So we omit those!

```
value
  staff: CHI → TT → CHI × TT
  staff(,,)(tt) ≡
    let icon = init ⌈⌉add ⌈⌉ delete ⌈⌉ ... in
    case icon of:
      init
        → let (r,tt′) = M_u(mk_init())(tt) in
            ((init,tt′,r),tt′) end,
      add
        → let fn:Fn,j:Journey • fn ∉ dom tt ∨ ... in
            let (r,tt′) = M_u(mk_add(fn,j))(tt) in
            ((add,mk_add(fn,j),r),tt′) end end,
      delete
        → let fn:Fn • fn ∈ dom tt ∨ ... in
            let (r,tt′) = M_u(mk_del(fn))(tt) in
            ((delete,mk_del(fn),r),tt′) end end
      _ → ... see the client function, now using its
            ... alternatives and the M_q function!
    end end
```

Remarks similar to those stated above in connection with, and subsequent to the formal description of client function can be repeated. So we omit thos!

2.2 A Software Architecture

2.2.1 Analysis

So we have most pieces ready to start the design of a software architecture. What is the problem and a possible solution?

- **Problem:**
  No "sharing" of time-table among n users and one staff. The client and the staff function descriptions of the requirements still refer to "own" copies of the time-table. Therefore:
- **A Solution:** System configured into concurrent client and staff processes interleaved wrt. a time-table process

Thus we "lift" from the client and staff function descriptions on pages 19, respectively 20, references to the time-table (tt) and "insert" in their place appropriate references to channels and channel communications!

But before we go in detail, let us review the software architecture design issues confronting us:

- **Software Architecture Design Issues:**
  - * *Requirements Analysis* See above!
    Basically done!
  - * *Overall System Configuration*
    It was decided above to decompose the system into n client process, one staff process and one time-table process. with appropriate channels.
  - * *Functional Requirements Projection*
    A hint was also given above: of "lifting" time-table tt handling of the requirements descriptions of the client and staff function descriptions (on pages 19, respectively 20) into channel declarations and communications.
  - * *Component–by–Component Design*
    By this we mean the "tidying up" of left-over issues: the introduction of appropriate new type spaces if required, etc.

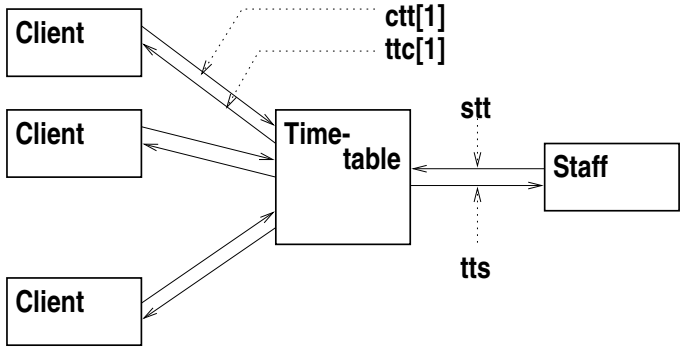
2.2.2 The System Process

The system process is one which "puts" in parallel (||) all the component processes and which contains a number of channels between these.

Process/Channel Diagram

From the below diagram one can actually automatically generate the basic formalisation which follows.

Figure 2: A Time-table Software Architecture



Architecture: A Time-table with clients and staff

Formalisation

system() ≡ ‖ { client(i) | i:Index } ‖ timetable(tt) ‖ staff()

The formalisation describes that all the Index (i.e. n) client processes are put in parallel, and these (in parallel) with the one time-table and the one staff process.

2.2.3 Channels — and some Design Analysis & Choices

But the formalisation above leaves out certain design decisions: Namely as to what to communicate over the channels.

From the "lifting" idea sketched above, we can conclude that when for example:

$\mathcal{M}_q$(mk_conn(m,da,ta))(tt)

is "lifted" from the connection alternative of the description of the client function, page 19, then it must be replaced by an expression that:

- communicates an appropriate request
- to the time-table process,
- and is communicated a query value
- back from that time-table process.

Thus the (tt) argument in the referenced line gives rise to the specific mentioning of client to time-table and time-table to client channels.

We can further ask: what about the input argument tt to the client function, page 19? The answer — given the specificities of the specification language (RSL) used — is that we remove the (tt) argument and replace it — in the type definition of the client function — by appropriate references to the fact that the software architecture client function communicates via specific channels with the timetable function. These channels we now define:

channel
ctt[i:Index]:Φ,
ttc[i:Index]:VAL,
stt:Ψ,
tts:RES

But how did we decide on what values to communicate to the time-table process?

Well there are basically two possibilities, and we naturally have to choose one. Either we could output from the client (or the staff) process the syntactical value of the query (respectively the update) request (i.e. the "raw" commands), or we could send their semantic counterparts, the Φ (respectively Ψ) denotations. We choose the latter!

2.2.4 Functional Requirements Projection

Before we present our choices for the description of the software architecture client, staff and time-table processes, let us review what we have projected from the requirements:

```
type
  Fn, An
  Journey = An ~m~> (T×T),
  TT = Fn ~m~> (An ~m~> Journey)
  Φ = TT → VAL,
  Ψ = TT → TT×RES
  RES == ok | nok | VAL
  VAL = TT | Journey | Conn-set
  Conn = Fn × (An × Fn)*
  Query == mk_brws() | mk_disp(fn:Fn)
             | mk_conn(m:Nat,fa:An,ta:An)
  Update == mk_init(tt:TT) | mk_add(fn:Fn,j:Journey)
             | mk_del(fn:Fn)
  CHI = Icon × Prompt × Result
  Icon = browse | display | connection | init | add | delete | nil
  Prompt == Query | Update | nil
  Result == VAL | RES
value
  ok, nok
  M
```

2.2.5 A Client Process

Based on the design decisions informally recorded above and the summary of the requirements type space projected onto the software architecture description we can now present the three component processes. First the client process:

```
value
  client: i:Index CHI → out ctt[i] in ttc[i] × CHI × Unit
  client(i,(,,)) ≡
    let icon = browse ⌈⌉ display ⌈⌉ connection in
    case icon of:
      browse
        → (browse,
             nil,
               ctt[i]!M_q(mk_brws()) ; tcc[i]?) end,
      display
        → let fn:Fn • fn ∈ dom tt ∨ ... in
          (display,
            mk_disp(fn),
```

```
                ctt[i]!M_q(mk_disp(fn)) ; tcc[i]?) end,
    connection
      → let da,ta:An • {da,ta} ⊆ Ans(tt) ∨ ... in
          (connection,
            mk_conn(m,fa,ta),
              ctt[i]!M_q(mk_conn(m,da,ta)) ; tcc[i]?) end
end
```

Note the fourth line from the connection alternative line in the client function definition. (This is the last expression just above!) The description here expresses: on channel ctt[i] output (!) denotation $\mathcal{M}_q$(mk_conn(m,da,ta)), then (;) input (?) a value from channel tcc[i]. This is the replacement for $\mathcal{M}_q$(mk_conn(m,da,ta))(tt) in the requirements definition of the client function on page 19.

Also note that the type definition specifies: **out** ctt[i] **in** ttc[i]. This is the replacement in the requirements definition of the client function, and of that functions formal parameter (tt) on page 19.

2.2.6 The Staff Process

Without much further a-do:

```
value
  staff: CHI × TT → out stt in tts × CHI × TT
  staff(,,)(tt) ≡
    let icon = init ⌈⌉add ⌈⌉ delete in
    case icon of:
      init
        → (init,nil,(stt!M_u(mk_init()) ; tts?)),
      add
        → let fn:Fn,j:Journey • fn ∉ dom tt ∨ ... in
            (add,
              mk_add(fn,j),
                stt!M_u(mk_add(fn,j)) ; tts?) end,
      delete
        → let fn:Fn • fn ∈ dom tt ∨ ... in
            (delete,
              mk_del(fn),
                stt!M_u(mk_del(fn)) ; tts?) end
      _ → ...
    end end
```

2.2.7 Time–table Process

The time–table process simply "listens" for input either from some client process or from the staff process:

```
value
  tt_init : TT
  timetable: TT → in {ctt[i]|i:Index},stt
                   out {ttc[i]|i:Index},tts × Unit
  timetable(tt) ≡
    let tt′ =
      [] { let φ = ctt[i]? in
           ttc[i]!φ(tt) ;
           tt end | i:Index }
      [] let η = stt? in
         if η:Ψ ["free notation"]
           then
             let (tt″,r) = η(tt) in
             tts!r ;
             tt″ end
           else [assert: η:Φ]
             tts!η(tt);
             tt
         end end
    in timetable(tt′) end
```

An input from any client process is always a query denotation, i.e. of type Φ. An input from a staff process is a denotation, either of an update, i.e. type Ψ, or of a query, i.e. of type Φ.

2.3 A Model–oriented Verification

We can argue, along the lines itemised below, that the client process of the architecture implements the client function of the requirements:

1. **Basis:**
   (a) Case: 'connection'
   (b) Requirements client line: page 19
   (connection,mk_conn(m,da,ta),$\mathcal{M}_q$(mk_conn(m,da,ta))(tt))
   (c) Three Architecture client lines: page 24
   (connection,mk_conn(m,da,ta),ctt[i]!$\mathcal{M}_q$(mk_conn(m,da,ta)));tcc[i]?
   (d) Architecture timetable lines: page 25
      i. **let** ϕ = ctt[i]? **in**
      ii. ttc[i]!ϕ(tt)
2. **Proof Procedure:**
   (a) ϕ, item 1(d)i, is ϕ in item 1(d)ii, and is the $\mathcal{M}_q$(mk_conn(m,da,ta)), item 1c;
   (b) Therefore replace ϕ, item 1(d)ii, with $\mathcal{M}_q$(mk_conn(m,da,ta)), item 1c;
   (c) insert it in lieu of tcc[i]? in item 1c;

(d) cancel its first part ctt[i]!$\mathcal{M}_q$(mk_conn(m,da,ta)) (since it has been "used");

(e) and you get third part of item 1b.

(f) QED

2.4 Conclusion

We have presented a cursory development: Systematic, not rigorous, let alone formal! We have "massaged" three variants of **client** functions ($\mathcal{D}$omain, $\mathcal{R}$equirements, respectively $\mathcal{S}$oftware architecture). We have shown a development of a domain description, of a requirements from that domain description, and of a software architecture from the requirements. And we have shown how all functional and some non-functional (CHI) requirements have been "implemented" — albeit abstractly.

Next we show how a program organisation "implements" remaining non-functional requirements.

3 Program Organisation

The concept of program organisation is a pragmatic one. We make the distinction — which the literature on Software Architecture seemingly does not make — that a *software architecture* exposes those interfaces (hence external) which "primary" users see: can observe and/or manipulate. It is as such that we say the a *software architecture* implements the functional requirements. In contrast, a *program organisation* exposes the internal interfaces between such components which "secondary" users can see. "Primary" users are those for which the functional requirements were first intended. In the example, of sections 2 and 3, those users are the clients and the staff. "Secondary" users are those for which the non-functional requirements were then intended. Examples of such users are those, typically facilities management personnel, who have been charged with the installation and maintenance of the delivered software. Typically internal interface allow these users to observe and tune performance indicators (perfective maintenance), to replace some components with other components for adaptive purposes or for corrective purposes.

Our distinction between software architecture and program organisation has its source in the similar terms: 'computer architecture' and 'machine organisation' as they were more or less explicit in [15]. That paper spoke of the IBM System/360 architecture in terms of what the assembler language level programmer ought to know about the bit, byte, halfword, word, double word and variable field length, as well as the channel data structures, and in terms of the instruction repertoire and channel commands that the IBM/360 "possessed". This architecture was shared across a very large spectrum of machine organisations as these were exemplified in the IBM/360 models 30, 40, 50, 65, 70, etc. The machine organisations of these computers went from 8 bit via 16 and 32 bit to 64 bit data flows, from simple, micro-programmed control (without caches) to highly complex, pipelined and logical circuitry controlled machines (the latter with elaborate caches).

We suggest to follow, wrt. software terminology, 35 years of established hardware terminology practice — when, as here, there are 'corresponding' terms!

3.1 Characterisation

We therefore propose the following "soft" characterisation of what constitutes a program organisation, namely:

- A decomposition of a **Software Architecture**
- into further **Components** with
  - (own) **State Spaces, Types**
  - **Predicates, Functions, Operations**
  - **Channels**, Communication (Events)
- with **Internal Interfaces**
  - to one another
- for purposes of
- satisfying **Non–functional Requirements**
- and usually not observable by the (casual) user.

The characterisation is "soft" since it is hard to draw a precise line between "primary (causal)" and "secondary" users, and similarly — often — between functional and non-functional requirements.
We shall anyway persist!

3.2 Design Decisions

We can structure initial program organisation design work as follows:

- **Major Design Decisions:**
  - Which Non-functional Requirements to Prioritise
- **Consequential Design Decisions:**
  - Decomposition: Components/Interfaces
  - Distribution & Concurrency
  - Connectors, Glue, Ports &c.
  - *&c.*

The above considerations often — and naturally — occur in connection with the software architecture design decision work. Hence a review is needed of non–functional requirements:

- Performance
- Dependability:

* Accessability * Avaliability * Reliability * Security

- CHI
- Maintainability:

  * Adaptive * Perfective * Corrective

- Devt. & Execution Platforms
- *&c.*

We already covered a list of non-functional requirements in section 2.1.7.

3.3 The Time–table Example

For the specific case example at hand we first restate the problem before suggesting a solution.

3.3.1 A Problem Analysis

We re–list some of the non-functional requirements wrt. how the current software architecture presently handles the issue:

- *Availability:* The timetable process does not guarantee "fair" choice between handling input from clients and staff processes. Cf. non-det. choice ([]), page 25.

  The non–deterministic choice that we are referring to above is that between time-table process' handling of the n:Index client process inputs and its handling of the staff process inputs (page 25).

- *Accessability:*

  The current softare architecture can be said to prescribe strict, mutually exclusive serialisation of client and staff processes wrt. timetable process — OK for zero time processing, not OK for time-consuming time-table operations (like for example connection queries)! "Small, quick" query processing, such as journey, could be interleaved with the processing of "large, time-consuming" queries, such as connections.

- *Maintainability:*

  Adaptive:
  Cf. direct channels between timetable and the client and staff processes. These direct connections, if also implemented "ad verbatum", might hinder the development (i.e. refinement) of several distinct implementations of the client process.

3.3.2 Design Decisions

The design decisions include a prioritisation of which non-functional requirements shall first, then subsequently, determine program organisation design decisions. Our example choice is:

- *Availability:*

  Insert an arbiter between client and staff processes.

  This arbiter shall secure a "more fair" choice — perhaps "less non-deterministic" — between the two categories of users.

  A solution could be to have an internal clock (or "bit") secure alternate sampling of client, respectively staff inputs to the time-table process.

- *Accesability:*

  Interleave of user and/or staff processing — implies the passing of user index even to the timetable process, a less than ideal solution when it comes to design for example a time-table process which is as general as possible!

  We decide here to make sure that two or more client processes can be serviced in parallel. This will be effected by a multiplexor process (mpx) inserted between the client processes and the arbiter.

  A journey query handling by the time-table process could thus be interleaved with the handling of a connection query from another client. The former may "arrive" at the timetable process before the latter, but that process may decide to first service the latter.

- *Maintainability:*

  Connectors between client, staff, multiplexor, arbiter and timetable processes.

  To secure that the developer does not take "white box" advantage of knowledge of how the client, staff and timetable processes are implemented, it is suggested to insert *connectors* between these and the (newly introduced) arbiter process. The existence of these connectors force a "standard" ("black box") interface between connected processes.

These were the elements of a "mosaic" of design choices.

3.3.3 A Process Diagram

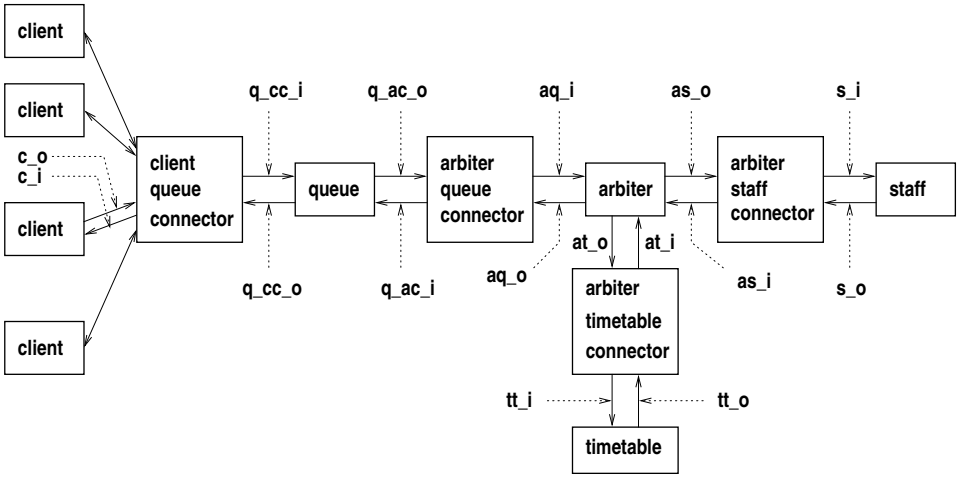
Now we piece the design choice "mosaic" elements together. First diagrammatically:

3.3.4 A Formal Process Model — The System Process

Then we "convert" the diagram prescription into a system process definition:

```
value
  systemπ() ≡
    ‖ { clientπ(i) | i:Index }
    ‖ cm_cπ()
    ‖ mpxπ()
    ‖ mq_cπ()
    ‖ arbiterπ() ‖ at_cπ() ‖ timetableπ(tt)
    ‖ as_cπ()
    ‖ staffπ()
```


Figure 3: A Time-table Program Organisation



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

- **Annotations:**
  - Connectors basically pass information on, in both directions.
  - Arbiter process:
    - interleave requests from clients and staff
    - and re-distributes timetable returned *results* to client, resp. staff processes (depending on types)
  - Multiplexor process multiplexes between users:
    - accepting query requests from users before "earlier" such requests have been concluded,
    - maintains internal records of outstanding requests — for audit purposes (a non-functional benefit only mentioned now)
    - and returns query results to users — but not necessarily in the order of their original issuance.

We leave it to the reader to define the client, mpx, arbiter, staff, timetable and the five connector processes.

We note that the multiplexor/arbiter, the staff/arbiter and the timetable/arbiter connector processes are basically identical modulo names of channels, but that the clients/multiplexor connector process is materially different.

We also observe that the connector processes are rather independent, in their abstracted behaviour, of the 'semantics' of the processes they connect. Thus only two connector processes need be implemented, the $n : 1$ and the $1 : 1$ connectors — with their specification being parameterised, and their instantiation being provided, with the information needed in order to match channel types.

3.3.5 A Formal Process Model — The Channel Declarations

channel

c_o[i:Index]:Φ,	c_i[i:Index]:VAL,
q_cc_i:(Φ×Index),	q_cc_o:(VAL×Index),
q_ac_o:(Φ×Index),	q_ac_i:(VAL×Index),
aq_i:(Φ×Index),	aq_o:(VAL×Index),
at_o:((Φ×Index)|Ψ),	at_i:((VAL×Index)|RES),
as_i:Ψ,	as_o:RES,
tt_i:((Φ×Index)|Ψ),	tt_o:((VAL×Index)|RES),
s_o:Ψ,	s_i:RES

3.3.6 A Formal Process Model — The Process Signatures

value

systemπ: **Unit** → **Unit**
clientπ: i:Index → **out** c_o[i] **in** c_i[i] × **Unit**
cq_cπ: **Unit** → **out** c_i[i],q_cc_i **in** c_o[i],q_cc_o × **Unit**
queueπ: **Unit** → **out** q_ac_o,q_cc_o **in** q_cc_i,q_ac_i × **Unit**
aq_cπ: **Unit** → **out** aq_i,q_ac_i **in** q_ac_o,aq_o × **Unit**
arbiterπ: **Unit** → **out** aq_o,as_o,at_o **in** aq_i,as_i,at_i × **Unit**
at_cπ: **Unit** → **out** tt_i,at_i **in** tt_o,at_o × **Unit**
timetableπ: **Unit** → **out** tt_o **in** tt_i × **Unit**
as_cπ: **Unit** → **out** as_i,s_i **in** s_o,as_o × **Unit**
staffπ: **Unit** → **out** s_o **in** s_i × **Unit**

4 Concluding Remarks

4.1 Summary

We have shown major stages in a development from domains, via functional and non-functional requirements to software architecture and program organisation.

We have shown this development in a style that alternates between design analysis and decisions, on one hand, and writing informal and formal descriptions, on the other hand. At all stages.

The latter: formalising all descriptions, from earliest conception to final code, is, despite 25 years of practice, still not generally appreciated.

We — perhaps not so modestly — suggest that the approach taken here constitutes:

- **a Reassessment of Software Engineering ! (?)**

4.2 Acknowledgements

I gratefully acknowledge Mr. Crilles Jansen, my M.Sc. Thesis student during the spring of 1998, for our joint work on software architectures, my colleagues at UNU/IIST and at my

current place of work (since 1976), and my Hong Kong and Macau colleagues for allowing me to present this paper.

5 Bibliographical Notes

Since this paper was first written its author has written and/or published a few other papers that expand on the general themes of domain and requirements engineering and on software architecture and program organisation: [16, 17, 18, 19]. Currently the author is in the process of wrting up the following more lecture note oriented documents: [20, 21, 22]

References

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

[2] Michael Jackson. *Software Hakubutsushi: Sekai to Kikai no Kijutsu (Software Requirements & Specifications: a lexicon of practice, principles and prejudices).* Toppan Company, Ltd., 2-2-7 Yaesu, Chuo-ku, Tokyo 104, Japan, 1997. In Japanese. Translated by Tetsuo Tamai (Univ. of Tokyo, tamai@graco.c.u-tokyo.ac.jp) and Hiroshi Sako; ISBN 4-8101-8098-0; xxv + 267 pages.

[3] Dines Bjørner, Souleimane Koussobe, Roger Noussi, and Georgui Satchok. Michael Jackson's Problem Frames: . In Li ShaoQi and Michael Hinchley, editors, *ICFEM'97: Intl. Conf. on Formal Engineering Mehtods*, Los Alamitos, CA, USA, 12–14 November 1997. IEEE Computer Society Press.

[4] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ., USA, 1981.

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

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

[7] Dines Bjørner. From Domain Engineering via Requirements to Software. Formal Specification and Design Calculi. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, November 1997. Paper published in SOFSEM'97 Proceedings, Springer-Verlag, Lecture Notes in Computer Science.

[8] Crilles Jansen and Dines Bjørner. Software Architectures and Program Organisation — A Case Study. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, March 1998.

[9] Crilles Jansen. Software Architectures and Program Organisation. M.sc. thesis, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, August 1998.

[10] 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. .

[11] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[12] The RAISE Method Group. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[13] Dines Bjørner. Domains as a Prerequisite for Requirements and Software — Domain Perspectives & Facets, Requirements Aspects and Software Views. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, 1997–1998. This report is included in the Springer-Verlag LNCS published proceedings from a US Office of Army Research sponsored workshop on *'Requirements Targeted Software and Systems Engineering*. The workshop was held at Bernried am Starnberger See, Bavaria, Germany, 12–14 October, and was locally organised by Institute of Informatics, Technical University of Munich, Germany. Editor: Manfred Broy.

[14] P. Zave and M. Jackson. Four dark corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997. .

[15] Gene Amdahl, Gerrit Blaauw, and Frederik Brooks. The IBM System/360 Architecture. *IBM Systems Journal*, 1964.

[16] Dines Bjørner and Jorge R. Cuéllar. Software Engineering Education: Rôles of formal specification and design calculi. *Annals of Software Engineering*, 6:365–410, 1998. Published April 1999.

[17] Dines Bjørner. Domain Modelling: Resource Management Strategics, Tactics & Operations, Decision Support and Algorithmic Software. In J.C.P. Woodcock, editor, *Festschrift to Tony Hoare*. Oxford University and Microsoft, September 13–14 1999.

[18] Dines Bjørner. A Triptych Software Development Paradigm: Domain, Requirements and Software. Towards a Model Development of A Decision Support System for Sustainable Development. In ErnstRüdiger Olderog, editor, *Festschrift to Hans Langmaack*. University of Kiel, Germany, October 1999.

[19] Dines Bjørner. Pinnacles of Software Engineering: 25 Years of Formal Methods. *Annals of Software Engineering*, 1999. Eds. Dilip Patel and Wang ....

[20] Dines Bjørner. Domain Engineering, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [21, 22].

[21] Dines Bjørner. Requirements Engineering, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [20, 22].

[22] Dines Bjørner. Software Design: Architectures and Program Organisation, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [20, 21].

