South African Computer Journal
Number 22
March 1999

Suid-Afrikaanse Rekenaar-tydskrif
Nommer 22
Maart 1999

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarvereniging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

World-Wide Web: `http://www.cs.up.ac.za/sacj/`

Subscriptions

	Annual	Single copy
Southern Africa	R80.00	R40.00
Elsewhere	US$40.00	US$20.00

An additional US$15 per year is charged for airmail outside Southern Africa

to be sent to:

Computer Society of South Africa
Box 1714, Halfway House, 1685
Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

WOFACS 98
IFIP WG2.3 and UNU/IIST
WINTER SCHOOL on PROGRAMMING METHODOLOGY
University of Cape Town, 6-17 July 1998

Chris Brink

Laboratory for Formal Aspects and Complexity in Computer Science, Department of Mathematics and Applied Mathematics, University of Cape Town

Since 1992 there has been a biennial Winter School on Formal and Applied Computer Science (WOFACS) at the University of Cape Town. Each of these have resulted in a *Proceedings* volume: *SACJ* **9**, **13**, **19**, and the volume you hold in your hand right now. All WOFACS events have had more or less the same structure. A group of eminent academics come to Cape Town during the winter vacation, and each of them offers, over a 2-week period, a course of 10 lectures on a particular topic. These short courses are pitched at about Honours level, and have some evaluation mechanism built in: short tests, or exercises, or assignments. At a student's request, and by arrangement with the Head of Department at his/her home institution, these courses can then be offered as part of the student's Honours degree. In this way WOFACS makes a contribution to beginning postgraduate studies on a wide geographical front. Typically such an event would attract students and young staff members not only from across South Africa, but also from many sub-Saharan African countries. Each WOFACS was organised by the UCT Laboratory for Formal Aspects and Complexity in Computer Science (FACCSLab).

WOFACS 98 had a distinctly international flavour. The entire event was, in fact, three things at once (which explains the somewhat complicated title at the top of this page). Besides being, by our reckoning, the 4th WOFACS, it was also the third in a series of outreach offerings of IFIP Working Group 2.3 on Programming Methodology. All the speakers were from WG2.3, and the entire event was built around the topic of programming methodology. Thirdly, the event was also an offering of the International Institute for Software Technology, situated at the United Nations University in Macao. The three role players, FACCSLab, WG2.3 and UNU/IIST, shared an agenda of trying to service specifically possible participants from disadvantaged communities and other African countries, and the UNU/IIST made available some generous grants for this purpose. In keeping with the tradition of these events there were no course fees: except for a fairly modest registration charge WOFACS has always been a free service to the community.

The speakers at WOFACS 98 and their topics were:

- Prof Dines Bjørner, Technical University Denmark: *Domains and Requirements, Software Architectures and Program Organisation.*
- Prof David Gries, Cornell University: *Logic as a Tool.*
- Prof Michael Jackson: *Problem Frames and Principles of Description.*
- Prof Jayadev Misra, University of Texas: *Toward an Applied Theory of Concurrency.*
- Dr Carroll Morgan, Oxford: *Predicate Transformers and Probabilistic Programs.*

Professor Gries' course was regarded as foundational, and recommended to all participants. It was offered during the first week only, at double tempo, thus giving the other four courses the opportunity to make use of concepts and techniques introduced there. Each of the speakers made available a Course Reader of their material. These were printed and bound before the event, and were handed out to participants upon registration. WOFACS 98 was attended by more than 60 participants, inter alia from Angola, Malawi, the Congo, Gabon, Cameroon, Malawi and Uganda.

From South African Universities we had representation, besides UCT, also from the University of Stellenbosch, the Transkei, the Qwa-qwa branch of the University of the North, the Witwatersrand, Pretoria, RAU, the North-West, Vista, UNISA, the Mangosothu Technikon and the Eastern Cape Technikon. Cape Town weather can be pretty stormy in July, but there were sufficiently many beautifully clear winter days to allow participants the opportunity to do some

sightseeing and exploring, after lectures or over the weekend.

We are grateful to our financial sponsors, and pleased to acknowledge their contributions.

- UNU/IIST.
- The Foundation for Research Development.
- The Chairman's Fund of Anglo American.

We thank the University of Cape Town for the use of its premises and facilities. We are particularly grateful to the 5 speakers, who put a lot of thought and preparation into their lectures, and tackled with great success the difficult task of conveying state-of-the-art material to a heterogeneous audience. It is only fair that specific thanks should be given to Carroll Morgan, whose idea it was in the first place to have a combined event, and to Dines Bjorner, who kickstarted the fundraising campaign. Finally, I would like to add my personal thanks to my colleagues and staff who worked so hard behind the scenes to make a success of WOFACS 98.

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, DK-2800 Lynbgy, Denmark; db@it.dtu.dk

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.*

Keywords: *software engineering, domain engineering, requirements engineering, software design, software architecture, program organisation; abstraction, modelling, development validation, correctnes verification; formal specification; RAISE, RSL*
Computing Review Categories: *D.2.1, D.2.4*

1 Introduction

1.1 A Development Paradigm

We cover main notions of a rigorous software development paradigm.[1] We propose the following decomposition:

1. **Software Engineering** ≈ **Domain Engineering** ⊕ **Requirements Engineering** ⊕ **Software Design**
2. **Software Design** ≈ *Software Architecture* ⊕ *Program Organisation* ⊕ (Further) Refinement Steps ⊕ Coding
3. **Software Development** ≈ All of the above!

We use primarily **Formal Specification**. and 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 and 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.**

[1] A version of this paper was presented Thu. 11 September 1998 at the annual JSSST (Japan Society for Software Science and Technology) Conference, Univ.of Electro Communications, nr. Tokyo,Japan. It was included in its otherwise entirely Japanese Conference Proceedings. Those proceedings are not publicly available.

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 **Study** and **Knowledge** of **Methods**

1.3 The Paradigm

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 stakeholders.

- **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 stakeholders.

  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 transformations* and/or *verified refinements*.

- **Correctness Verification:**

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

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

Paraphrasing Barry W. Boehm:[3]

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

1.4 Aims & Objectives

The goal of this paper is to broadly, and in a pedagogical clear and didactically "complete" style, make the reader aware: of the triptych/triple Paradigm: Domain + Requirements + Software Design, of Abstraction and Modelling Techniques, of Refinement concepts and of Correctness concerns, that Large–scale Development can be tackled in a *Trustworthy, Efficient* manner, of need for *Formal Techniques* and that New Product Ideas can Arise when using Formal Specification within the Domain / Requirements / Software Design Paradigm

1.5 Message Delivery

How will we deliver the message ? 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.

2 Example Development

This section has several parts. First (Sect.2.1) we discuss briefly what an architecture is and give — right-away — an example CSP-like [6, 8] "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?"*.

2.1 A Pre–view Architecture

2.1.1 "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) [1] 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.

2.1.2 "Picture Worth a 1000 Words"

Let us illustrate, by a diagram, figure 1, 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.

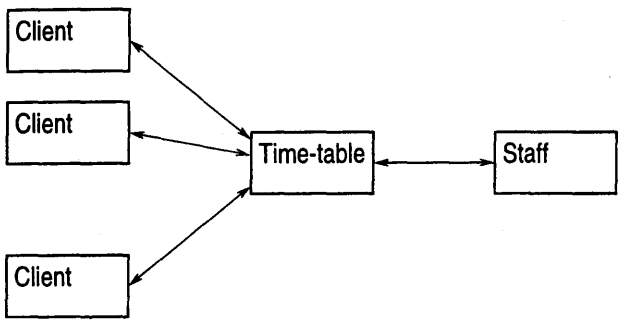

Figure 1: An 'Time-table Software' Architecture

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?"*.

2.1.3 "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.[2]
- 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 Formalisation

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:[3]

```
type
   Index, TT, VAL, RES,
   Φ = TT→VAL,
   Ψ = TT→TT×RES
channel
   ctt[i:Index]:Φ,ttc[i:Index]:VAL, stt:Ψ, tts:RES
value
   φ1,φ2,..,φq:Φ, ψ1,ψ2,..,ψu:Ψ
   cφ: Unit → Φ Unit,
   cφ() ≡ φ1 ⌈⌉ φ2 ⌈⌉ .. ⌈⌉ φq
   sψ: Unit → Ψ Unit,
   sψ() ≡ ψ1 ⌈⌉ ψ2 ⌈⌉ .. ⌈⌉ ψu

   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())

   timtbl: TT → in ctt[i],stt out ttc[i],tts Unit
   timtbl(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 timtbl(tt') end

   system() ≡
      || { client(i) | i:Index }
      || timtbl(tt) || staff()
```

Warning: *We are using a simplified variant of* RSL *[4], one in which we leave out class (and scheme) definitions — thereby "effectively" making all definitions designate objects!*

2.2 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: 2.3, Requirements: 2.4) to the systematic unfolding of an architecture.

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

[3] In this paper we use a rather simplified version of RSL [4], the RAISE method [5] Specification Language. In this simplified version we treat all channel and value definitions as those of objects (in the sense of RSL).

2.3 A Domain

After having acquired knowledge about the application domain we write it down. We otherwise refer to [2] for a more elaborate presentation of principles and techniques of domain modelling.

2.3.1 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.

2.3.2 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.

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.

2.3.3 Domain Formalisation

type
  TT, VAL, RES
  Φ = TT → VAL
  Ψ = TT → TT×RES

value
  $\phi_1,\phi_2,...,\phi_q$:Φ
  $\psi_1,\psi_2,...,\psi_u$:Φ

  client: TT → VAL
  client(tt) ≡
    let ϕ:Φ = $\phi_1 \sqcap \phi_2 \sqcap ... \sqcap \phi_q$ **in**
    ϕ(tt) **end**

  staff: TT → TT×RES
  staff(tt) ≡
    let ψ:Ψ = $\psi_1 \sqcap \psi_2 \sqcap ... \sqcap \psi_q$ **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.

2.4 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.

2.4.1 Functional Requirements

Functional requirements "derive" from, or as Michael Jackson and Pamela Zave [9] expresses it: "resides in", "the domain". Constructing functional requirements, to us, include the following techniques: Projection, instantiation and extension. These will now be illustrated.

- **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 $\xrightarrow{m}$ Jrn
    Jrn′ = An $\xrightarrow{m}$ (arr:T × dept:T)

```
    Jrn = {| j:Jrn′ • wf_Jrn(j) |}
value
    wf_Jrn: Jrn′ → Bool
    wf_Jrn(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!

2.4.2 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. 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".*

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

  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).

```
type
  CHI = Icon × Prompt × Result
  Icon = brws | disp | conn
       | init | add | del | 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.

2.4.3 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 abstractly.

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

In the above model, we have, for simplicity, omitted any description of any temporal sequence that relates the three parts of the CHI interface.

Redefinition of Staff Function

```
value
  staff: CHI → TT → CHI × TT
  staff(,,)(tt) ≡
    let icon = init ⌈⌉ add ⌈⌉ del ⌈⌉ ...
    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,
      del →
        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
```

2.5 A Software Architecture

2.5.1 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 descrip-

tions 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 6, respectively 6, references to the time--table (tt) and "insert" in their place appropriate references to channels and channel communications!

2.5.2 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 figure 2 one can actually automatically generate the basic formalisation which follows.

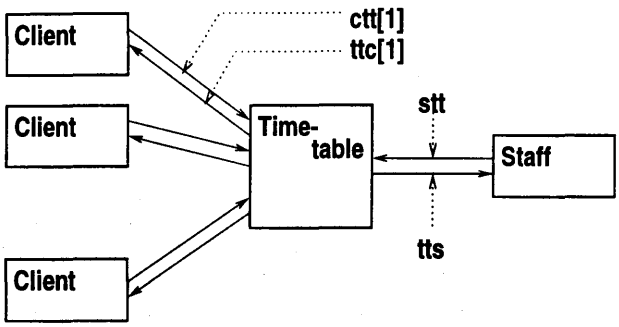

Figure 2: A Time-table Software Architecture

Formalisation

```
system() ≡
    || { client(i) | i:Index }
    || timtbl(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.

2.5.3 Channel 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 6, 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. 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!

2.5.4 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 = brws ⌈⌉ disp ⌈⌉ conn in
        case icon of:
            brws →
                (brws,nil,
                ctt[i]!Mq(mk_brws());tcc[i]?) end,
            disp →
                let fn:Fn • fn ∈ dom tt ∨ ... in
                (disp,mk_disp(fn),
                 ctt[i]!Mq(mk_disp(fn));tcc[i]?) end,
            conn →
                let da,ta:An • {da,ta} ⊆ Ans(tt) ∨ ... in
                (conn,mk_conn(m,fa,ta),
                 ctt[i]!Mq(mk_conn(m,da,ta));tcc[i]?)
        end end
```

Note 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 6.

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 6.

2.5.5 The Staff Process

Without much further a-do:

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

2.5.6 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
  timtbl: TT → in {ctt[i]|i:Index},stt
               out {ttc[i]|i:Index},tts Unit
  timtbl(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 timtbl(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.

2.5.7 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 6
   (conn,mk_conn(m,da,ta),
   $\mathcal{M}_q$(mk_conn(m,da,ta))(tt))
   (c) Three Architecture client lines: page 7
   (conn,mk_conn(m,da,ta),
   ctt[i]!$\mathcal{M}_q$(mk_conn(m,da,ta)));tcc[i]?
   (d) Architecture timtbl lines: page 8
      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) (since it has been "used") cancel its first part ctt[i]!$\mathcal{M}_q$(mk_conn(m,da,ta));
   (e) and you get third part of item 1b.
   (f) QED

2.6 Program Organisation

The concept of program organisation is a pragmatic one. We make the distinction 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.

2.6.1 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.

2.6.2 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.*

2.6.3 The Time–table Example

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

A Problem Analysis

We 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 8.

  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 8).

- *Accessability:*

  The current softare architecture can be said to prescribe strict, mutually exclusive serialisation of client and staff processes wrt. timtbl process — OK for zero time processing, not OK for time-consuming time-table operations (like for example connection queries)!

- *Maintainability:*

  Adaptive: Cf. direct channels between timtbl 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.

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.

- *Accesability:*

  Interleave of user and/or staff processing — implies the passing of user index even to the timtbl 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.

- *Maintainability:*

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

  To secure that the developer does not take "white box" advantage of knowledge of how the client, staff and timtbl processes are implemented, it is suggested to insert *connectors* between these and the (newly introduced) arbiter process.

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

A Process Diagram

Now we piece the design choice "mosaic" elements together. Figure 3 diagrams processes and chanells.

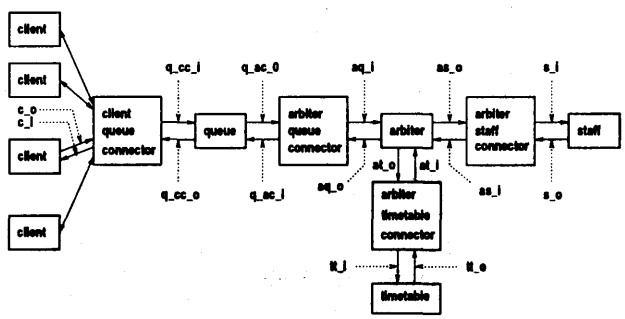

Figure 3: A Time-table Program Organisation

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() || timtblπ(tt)
```

```
    || as_cπ()
    || staffπ()
```

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.

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
```

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
  timtblπ:
    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
```

3 Concluding Remarks

3.1 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.

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

- **a Reassessment of Software Engineering**

3.2 Acknowledgements

I gratefully acknowledge Mr. Crilles Jansen, my M.Sc. Thesis student during the spring of 1998, for our joint work on software architectures [7], my colleagues at UNU/IIST and at my current place of work (since 1976), my Japanese friends, in particular Prof., Dr. Tetsuo Tamai, Tokyo University, for inviting me to present this paper at the JSSST Annual Conference, see footnote 1, and Prof., Dr. Chris Brink for inviting me to publish this paper in this journal.

4 Bibliographical Notes

References

[1] 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. .

[2] 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. .

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

[4] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[5] The RAISE Method Group. *The RAISE Method.* The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

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

[7] Crilles J. Jansen. *Software Architectures and Program Organisations.* M.Sc. Thesis, Dept. of Information Technology, Technical University of Denmark, July 31, 1998.

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

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

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for *review* should be prepared according to the following guidelines:

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
- the title (as brief as possible)
- the author's initials and surname
- the author's affiliation and address
- an abstract of less than 200 words
- an appropriate keyword list
- a list of relevant Computing Review Categories
- Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text according to the Harvard. References should also be according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page,

`http://www.cs.up.ac.za/sacj`

which gives a number of examples.

SACJ is produced using the LaTeX document preparation system, in particular LaTeX 2_ε. Previous versions were produced using a style file for a much older version of LaTeX, which is no longer supported. Please see the web site for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.